This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 318353.

D31.4
Test-Generation Methods

Project number:	318353
Project acronym:	EURO-MILS
Project title:	EURO-MILS: Secure European Virtualisation for Trustworthy Applications in Critical Domains
Start date of the project:	1st October, 2012
Duration:	40 months
Programme:	FP7/2007-2013

Deliverable type:	Report
Deliverable reference number:	ICT-318353 / D31.4 / 1.0
Activity and Work package contributing to deliverable:	Activity 3 / WP31
Due date:	January 2016 – M38
Actual submission date:	4th February, 2016

Responsible organisation:	PSud
Editor:	Burkhart Wolff, Yakoub Nemouchi
Dissemination level:	PU
Revision:	1.0 (r1.0)

Abstract:	Methodology, theories, implementation and case-study of a MBT-approach for PikeOS.
Keywords:	PikeOS, Common Criteria, Isabelle/HOL, HOL-TestGen

Editor

Burkhart Wolff, Yakoub Nemouchi (PSud)

Contributors (ordered according to beneficiary numbers)

Abderrahmane Feliachi, Yakoub Nemouchi, Burkhart Wolff (Université Paris Sud)

Sergey Tverdyshev, Oto Havle, Holger Blasum (SYSGO AG)

Bruno Langenstein, Werner Stephan (Deutsches Forschungszentrum für künstliche Intelligenz / DFKI GmbH)

Cyril Proch (Thales Communications & Security SA)

Freek Verbeek (Open University of The Netherlands)

Julien Schmaltz (Technische Universiteit Eindhoven)

Acknowledgment
The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318353.

This document has gone through the consortiums internal review process and is still subject to the review of the European Commission. Updates to the content may be made at a later stage.

Executive Summary

This document consists of four parts:

- Part I gives an overview over formal automated test-generation techniques developed in the EURO-MILS project. This contains a chapter on new methodologies as well as a chapter on implemented techniques in HOL-TestGen.
- Part II develops the main approach of concurrent code-testing supported by HOL-TestGen, presented at an academic example called MyKeOS.
- Part III develops the main case-study on test-case generation for the IPC protocol, which is run against a PikeOS demonstrator.
- Part IV comprises a collection of technical annexes: a) the current HOL-TestGen Reference Manual, b) the Test Theory for the IPC protocol, c) the code for Test Execution Adapters and d) the code for the PiKeOS demonstrator.

Contents

Part I

Introduction and Context

Chapter 1

Overview of the Research Activities in WP31.4

The term “Formal methods” refers to a set of mathematically based techniques and tools for specification, analysis and verification of computer systems. They are mainly used to describe and to verify, in a logically consistent way, some properties of these systems. The formal specification and verification approaches rely usually on some underlying logic. The logical foundation of theorem provers makes them a very convenient basis of any formal development, where the specification and the verification activities can be gathered in one formal environment.

In the context of a certification following the scheme of Common Criteria EAL5-7, formal methods were applied in particular to build descriptions of the security properties that a system considered as the *target of evaluation* (TOE) should achieve. These security properties are are stated in terms of a *security policy model* (SPM) as well as a *formal functional model* (FSP) of the TOE system. For the higher evaluation assurance levels (EAL5-7), these models are required to be formal models, based on a formal method, allowing these descriptions to be unambiguous and machine-checked. In particular the latter is a notable pre-requisite in a collective modeling/proof effort for a complex system model. Beyond a rigourous scheme of documentation, the certification schemes on higher EAL levels are reside on essentially two verification techniques:

1. An (at the highest level) fully-formal refinement proof of the SPM by the FSP, which assures that the security properties are actually established by the functional model, and
2. a rigourous testing techniques wrt. to the real implementation, that allows for establishing confidence in the FSP and a reliable link between the model and the reality in the C-implementation.

In the context of the EURO-MILS project, it was decided from the beginning that the modeling effort from SPM to FSP would be untertaken in Isabelle/HOL[Nip12]. The idea of using a test-generation method based on Isabelle/HOL models is therefore particularly attractive for establishing the link between the FSP and the real implementation in C-code. CC evaluation for aspect of testing (ADV_ATE) is a tedious task especially when couverage is concerned. Currently developers usually do test coverage analysis wihtout any tool support. So the approach presented here could make the task of evaluating test coverage cheaper and less error prone. In recent years, HOL-TestGen [BW13] has been developed for testing models presented in HOL, in particular for operations with complex data-structures, so data-types comprising lists, sets, trees, records, ... Tests were generated in the logical context of a background theory and wrt. to a particularly property (called *test-specification*) formulated in it. At the begin of the project, HOL-TestGen was mostly geared towards the generation of unit-tests and test-specifications of the form:

$$\mathrm{pre}(x) \rightarrow \mathrm{post}(x, \mathrm{SUT}(x))$$

where x is arbitrary input, so possibly also containing an input state, pre and post a pre- and a post-condition, and SUT an uninterpreted constant symbol representing the *system under test*. The test-specification schema covers test scenarios where the initial state of the system is known and the result state is returned by the SUT; it is therefore assumed to be accessible in principle. HOL-TestGen provides automatic procedures for a test-generation process that works in principle as follows: first, a procedure decomposes via data-type splitting rules and a kind of DNF-normalization the initial test-specification into *abstract test cases*, i. e. clauses containing $\mathrm{SUT}(\mathrm{x})$ plus a collection of logical constaints on x. Second, , under the condition that these constraints are satisfiable, a constraint-solver can produce automatically a ground instance for x, say c, and isolate $\mathrm{post}(c, \mathrm{SUT}(c))$ as *concrete test*. If these constraints are unsatisfiable, the abstract test cases are *infeasible*, i. e. represent impossible (empty) test-cases. Eliminating infeasible test cases as early as possible is primordial for effective test generation; it is also the key advantage over random-based testing which tends to be hopelessly inefficient if pre-conditions are non-trivial. Finally, HOL-TestGen offers the possibility to convert concrete test suites via code-generators into test drivers in a variety of target languages.

HOL-TestGen and its methodology is an instance of *model-based testing* (see [ABC+13] for a recent survey over the evolving field, which was pioneered by M.C. Gaudel at the beginning of the 90ies[GB91, Gau95]). However, its methodology coined "proof-based testing" distinguishes itself from main-stream approaches by the following features:

1. rather than residing on small, decidable data-type theories in a propositional or first-order logic setting, HOL-TestGen embraces higher-order logic (HOL) and favors for background theories and test specifications abstract and concise mathematical descriptions rather than indirect problem-encodings;
2. HOL-TestGen allows for *instrumenting* the generation processes of abstract and concrete test cases by *derived rules*, i. e. rules that are short-cuts for the normalization and data selection phases which were justified by formal proof;
3. HOL-TestGen leverages the possibility to "massage" of a *given* model into *testable* one; beyond aforementioned instrumentation of the process, an initial model can be *refined* or *restricted* to a model that is more suited for test-generation and its underlying need for a symbolic execution process;
4. HOL-TestGen offers the possibility of a semantically controlled, clean integration from models to the test driver generation.

However, at the beginning of the EURO MILS project, it was not clear how the approach could be effectively applied to an operating system model involving

1. concurrency and communication, and
2. *very* heavy states and complex data-structures (involving a model on physical memory and policy representations).

Prior work [BBW15] with HOL-TestGen had shown that sequence test scenarios could be treated effectively in principle, if the background theory is geared towards efficient symbolic execution and if the process is decently supported by automated reasoning. However, there is no direct way to generalize the reification technique used in [BBW15] to the PikeOS model.

The research centered around the proof-based test-generation activity (the 31.4 activity in the EURO MILS project) followed therefore two lines of research:

1. Based on prior experience, we attempted to use a "process-algebraic approach" based on the process-algebra Circus[WC02], which models synchronization and concurrency naturally, but which requires a certain effort in instrumentation and "massaging" of the PikeOS FSP into a Circus representation;
2. we attempted to push the "monadic approach" underlying [BBW15], i. e. a specific form of symbolic execution of execution sequences representing Mealy-machines, io-automata or io-lts'es *indirectly*, towards more efficient deductive support for test-sequence generations includinging synchronizations and formally proved reductions of the sequence-space.

1.1 The process-algebraic approach to test-generation

Based on prior work[FGW10, FWG12, FGW12], Abdou Feliachi and Burkhart Wolff pursued the approach further to the two publications, namely: *The Circus Testing Theory revisited in Isabelle/HOL* (by Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel, and Burkhart Wolff) [FGWW13] and the subsequent journal publication comprising a semi-industrial case-study: *Symbolic Test-generation in HOL-TestGen/Cirta* (by Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff) [FGW15].

In order to give an overview on these works, we represent here their content by their abstract:

1.1.1 The Circus Testing Theory Revisited in Isabelle/HOL

Formal specifications provide strong bases for testing and bring powerful techniques and technologies. Expressive formal specification languages combine large data domain and behavior. Thus, symbolic methods have raised particular interest for test generation techniques. Integrating formal testing in proof environments such as Isabelle/HOL is referred to as "theorem-prover based testing". Theorem-prover based testing can be adapted to a specific specification language via a representation of its formal semantics, paving the way for specific support of its constructs. The main challenge of this approach is to reduce the gap between pen-and-paper semantics and formal mechanized theories. In this paper we consider testing based on the Circus specification language. This language integrates the notions of states and of complex data in a Z-like fashion with communicating processes inspired from CSP. We present a machine-checked formalization in Isabelle/HOL of this language and its testing theory. Based on this formal representation of the semantics we revisit the original associated testing theory. We discovered unforeseen simplifications in both definitions and symbolic computations. The approach lends itself to the construction of a tool, that directly uses semantic definitions of the language as well as derived rules of its testing theory, and thus provides some powerful symbolic computation machinery to seamlessly implement them both in a technical environment.

1.1.2 Symbolic Test-generation in HOL-TestGen/Cirta

HOL-TestGen/CirTA is a theorem-prover based test generation environment for specifications written in Circus, a process-algebraic specification language in the tradition of CSP. HOL-TestGen/CirTA is based on a formal embedding of its semantics in Isabelle/HOL, allowing to derive rules over specification constructs in a logically safe way. Beyond the derivation of algebraic laws and calculi for process refinement, the originality of HOL-TestGen/CirTA consists in an entire derived theory for the generation of symbolic test-traces, including optimized rules for test-generation as well as rules for symbolic execution. The deduction process is automated by Isabelle tactics, allowing to protract the state-space explosion resulting from blind enumeration of data. The implementation of test-generation procedures in CirTA is completed by an integrated tool chain that transforms the initial Circus specification of a system into a set of equivalence classes (or "symbolic tests"), which were compiled to conventional JUnit test-drivers. This paper describes the novel tool-chain based on prior theoretical work on semantics and test-theory and attempts an evaluation via a medium-sized case study performed on a component of a real-world safety-critical medical monitoring system written in Java. We provide experimental measurements of the kill-capacity of implementation mutants.

1.1.3 The Process-algebraic Approach: A Summary

Overall, the process-algebraic approach does not seem to be adequate for various reasons. It seems to be a double investment — on the one hand, substantial effort has to be done to develop improved automated support on the shallow embedding of Circus, on the other hand, the distance between the PikeOS FSP developed in the project and a test-theory developed in Circus got larger and larger throughout the project. This became painfully visible when it was decided that system calls in the PikeOS functional model were not only modeled by a sequence of atomic actions (which is perfectly possible in CSP-like languages such as Circus), but that the actions of a system call can also be aborted when, for example, an access-control violation has been detected. This leads to a somewhat non-standard notion of interleaving that gives away the main-advantage of the process-algebraic approach.

1.2 The Monadic Approach to Test-generation

The monadic approach is based on the idea that the transition relation of the system under test can be seen as a monad operation — be it in a state-exception monad in the case of a deterministic transition relation or be it in a Kleisli-Monad in the case of a non-deterministic transition relation. This concept heavily used in purely functional programming languages such as Haskell is a viable approach to model stateful systems in state-less higher-order logics. It can be seen as a kind of abstract reformulation of classical automata concepts, but lends itself via *monad transformers* to modular/aspect-oriented descriptions of complex systems, where the theory of the transformers can be treated as an object of theoretical interest in their own.

Although already sketched in [BW13], its theoretical exploration (reflected in substantial extensions in the HOL-TestGen library) as well as its practical support in the HOL-TestGen system have been greatly improved during the EURO-MILS project and were seen as major contribution to this deliverable.

There had been two major publications along this line of research, namely: *Test Program Generation for a Microprocessor - a Case-Study* (by Achim D. Brucker, Abderrahmane Feliachi, Yakoub Nemouchi, and Burkhart Wolff) [BFNW13] and the subsequent paper: *Testing the IPC Protocol for a Real-Time Operating System* (by Achim D. Brucker, Oto Havle, Yakoub Nemouchi and Burkhart Wolff) [BHNW15].

While the former paper addressed to the question "How to test the requirements of the hardware" underlying an operation system was merely for us a fore-runner to advance the underlying technologies (it was also done at a very early moment of the project where the SPM and FSP were still under heavy development), the latter is right in the focus of our research activities for PikeOS. We will therefore refer to the abstract of the former paper here and refer the reader to Part II which is basically an extended version of the latter.

1.2.1 Test Program Generation for a Microprocessor - a Case-Study

Certifications of critical security or safety system properties are becoming increasingly important for a wide range of products. Certifying large systems like operating systems up to Common Criteria EAL 4 is common practice today, and higher certification levels are at the brink of becoming reality. To reach EAL 7 one has to formally verify properties on the specification as well as test the implementation thoroughly. This includes tests of the used hardware platform underlying a proof architecture to be certified. In this paper, we address the latter problem: we present a case study that uses a formal model of a microprocessor and generate test programs from it. These test programs validate that a microprocessor implements the specified instruction set correctly. We built our case study on an existing model that was, together with an operating system, developed in Isabelle/HOL. We use HOL-TestGen, a model-based testing environment which is an extension of Isabelle/HOL. We developed several conformance test scenarios, where processor models were used to synthesize test programs that were run against real hardware in the loop. Our test case generation approach directly benefits from the existing models and formal proofs in Isabelle/HOL.

Chapter 2

HOL-TestGen: Its Architecture and Methodology

In this chapter, we will describe HOL-TESTGEN and its extensions developed throughout the EURO-MILS project. The system is open-source and the final version 1.8 of the development activities around the system can be uploaded from the HOL-TESTGEN web-page [1].

2.1 Isabelle/HOL

2.1.1 The Isabelle System Architecture

We will describe the layers of the system architecture bottom-up one by one, following the diagram Figure 2.1.

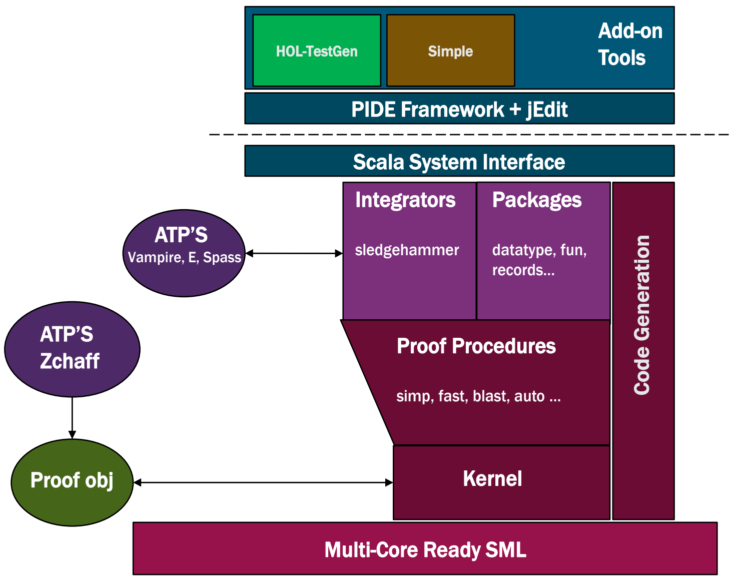


Figure 2.1: The diagram shows the different layers like execution environment, kernel, tactical level and proof-procedures, component level (providing external prover integration like Z3, specification components, and facilities like the code generator, the Scala API to the system bridging to the JVM-World, and the Prover-IDE (PIDE) layer allowing for asynchronous proof and document checking.

[1] https://www.brucker.ch/projects/hol-testgen/

The foundation of system architecture is still the Standard ML (SML,[MTM97]) programming environment; the default PolyML implementation
www.polyml.org supports nowadays multi-core hardware which is heavily used in recent versions for parallel and asynchronous proof checking when editing Isabelle theories.

On top of this, the logical kernel is implemented which comprises type-checking, term-implementations and the management of global contexts (keeping, among many other things, signature information and basic logical axioms). The kernel provides the abstract data-types `thm`, which is essentially the triple (Γ, Θ, ϕ), written $\Gamma \vdash_\Theta \phi$, where Γ is a list of *meta-level assumptions*, Θ the *global context*, containing, for example, the signature and core axioms of `HOL` and the signature of group operators, and a $conclusion$ ϕ, i. e.a formula that is established to be derivable in this context (Γ, Θ). Intuitively, a `thm` of the form $\Gamma \vdash_\Theta \phi$ is stating that the kernel certifies that ϕ has been derived in context Θ from the assumptions Γ.

There are only a few operations in the kernel that can establish `thm`'s, and the system correctness depends *only* on this trusted kernel. On demand, these operations can also log proof-objects that can be checked, in principle, independently from Isabelle; in contrast to systems like Coq, proof objects do play a less central role for proof checking which just resides on the inductive construction of `thm`'s by kernel inferences shown, for example, in [PP10].

On the next layer, proof procedures were implemented - advanced tactical procedures that search for proofs based on higher-order rewriting like `simp`, tableau provers such as `fast`, `blast`, or `metis` , and combined procedures such as `auto`. Constructed proofs were always checked by the inference kernel.

The next layer provides major components — traditionally called *packages* — that implement the *specification constructs* such as *type abbreviations*, *type definitions*, etc., as discussed in subsection 2.1.4 in more details. Packages may also yield connectors to external provers (be it via the **`sledgehammer`** interface or via the `smt` interface to solvers such as Z3), machinery for (semi-trusted) code-generators as well as the Isar-engine that supports structured-declarative and imperative "apply style" *proofs* described in subsection 2.1.7.

The Isar - engine [Wen02] parses specification constructs and proofs and dispatches their treatment via the corresponding packages. Note that the Isar-Parser is configurable; therefore, the syntax for, say, a data-type statement and its translation into a sequence of logically safe constant definitions (constituting a "model" of the data type) can be modified and adapted, as well as the automated proofs that derive from them the characterizing properties of a data-type (distinctness and injectivity of the constructors, as well as induction principles) as `thm`'s available in the global context Θ thereafter. Specification constructs represent the heart of the methodology behind Isabelle: new specification elements were only introduced by "conservative" mechanisms, i. e. mechanisms that maintain the logical consistency of the theory by construction; internally these constructs introduce declarations and axioms of a particular form. Note that some of these specification constructions, for example type definitions, require proofs of methodological side-conditions (like the non-emptyness of the carrier set defining a new type).

We mention the last layer mostly for completeness: Recent Isabelle versions posses also an API written in Scala, which gives a general system interface in the JVM world and allows to hook-up Isabelle with other JVM-based tools or front-ends like the jEdit client. This API, called the "Prover IDE" or "PIDE" framework, provides an own infrastructure for controlling the concurrent tasks of proof checking. The jEdit-client of this framework is meanwhile customized as default editor of formal Isabelle *sessions*, i. e. the default user-interface the user has primarily access to. PIDE and its jEdit client manage collections of theory documents containing sequences of specification constructs, proofs, but also structured text, code, and machine-checked results of code-executions. It is natural to provide such theory documents as part of a certification evaluation documentation.

2.1.2 Isabelle and its Meta-Logic

The Isabelle kernel natively supports minimal higher-order logic called *Pure*. It supports for just one logical type prop the meta-logical primitives for implication $\_ \Longrightarrow \_$ and universal quantification $\bigwedge x.\ P\ x$. The meta-logical primitives can be seen as the constructors of *rules* for various logical systems that can be represented inside Isabelle; a conventional "rule" in a logical textbook:

$$\frac{A_1 \cdots A_m}{C} \tag{2.1}$$

can be directly represented via the built-in quantifiers $\bigwedge$ and the built-in implication $\Longrightarrow$ as follows in the Isabelle core logic Pure:

$$\bigwedge x_1 \ldots x_n \,.\ A_1 \Longrightarrow \ldots \Longrightarrow A_m \ \Longrightarrow \ \mathbf{C} \tag{2.2}$$

…where the variables $x_1, \ldots, x_n$ are called *parameters*, the premises $A_1, \ldots, A_m$ *assumptions* and C the conclusion; note that $\Longrightarrow$ binds to the right. Also more complex forms of rules as occurring in natural deduction style inference systems like:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B} \tag{2.3}$$

can be represented by $(A \Longrightarrow B) \Longrightarrow A {\to} B$. Thus, the built-in logic provided by the Isabelle Kernel is essentially a language to describe (systems of) logical rules and provides primitives to instantiate, combine, and simplify them. Thus, Isabelle is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL. Moreover, Isabelle is also a generic system framework (roughly comparable with Eclipse) which offers editing, modeling, code-generation, document generation and of course theorem proving facilities; to the extent that some users use it just as programming environment for `SML` or to write papers over checked mathematical content to generate LaTeX output. Many users know only the theorem proving language **isar!** for structured proofs and are more or less unaware that this is a particular configuration of the system, that can be easily extended. Note that for all of the aforementioned specification constructs and proofs there are specific syntactic representations in **isar!**.

Higher-order logic (HOL) [Chu40, And86, And02] is a classical logic based on a simple type system. It is represented as an instance in Pure. HOL provides the usual logical connectives like $\_ \wedge \_$, $\_{\to}\_$, $\neg\_$ as well as the object-logical quantifiers $\forall x.\ P\ x$ and $\exists x.\ P\ x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centred around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \mathrm{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on a polymorphically typed λ-calculus, **hol!** can be viewed as a combination of a programming language like `SML` or Haskel, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is the session based on the embedding of HOL into Isabelle/Pure. Note The that simple-type system as conceived by Church for HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskel[WB89, Wen97].

The core of the logic is done via an axiomatization of the core concepts like equality, implication, and the existence of an infinite set, the rest of the library is derived from this core by logically safe ("conservative") extension principles which are syntactically identifiable constructions in Isabelle files. In the following, we will briefly describe the axiomatic foundation of Isabelle/HOL and describe the most common conservative extension principles.

2.1.3 The Logical Core of HOL.

In the entire library (so the Isabelle session "HOL" which is also referred to as "Main" in theory imports), there are only 11 axioms in form of foundational axioms of the HOL-logic:

1. The equality symbol is axiomatized as an equality, i. e. it is reflexive, extensional, and satisfies the Leibniz-property (equals can be replaced by equals in any context P). The Hilbert-Operator is bound to choose the value characterized by equality:

```
axiomatization
where refl        : t = (t::α) and
      subst       : s = t ⟹P s ⟹P t and
      ext         : (⋀ x::α. (f x ::β) = g x) ⟹
                    (λx. f x) = (λx. g x) and
      the_eq_trivial: (THE x. x = a) = (a::'a)
```

2. The following axioms establish a relation between implication and rule formation, and between implication and equality, as well as `True`, `∀ x. P x` and `False` and (which are abbreviations for `((λx::bool. x)= (λx. x))`, `(P = (λx. True))` and `(∀P. P)`, respectively):

```
axiomatization
where impI      : (P ⟹Q) ⟹P ⟶Q and
      mp        : P ⟶Q ⟹P ⟹Q and
      iff       : (P⟶Q) ⟶(Q⟶P) ⟶(P=Q) and
      True_or_False: (P=True) ∨(P=False)
```

3. Finally, a type `ind` is postulated to have an interpretation by an infinite carrier set. Instead of the more common form to state the axiom of infinity: `∃f::ind⇒ind. injective(f)∧¬ surjective(f)` , this axiom comes in two parts over two constants `Zero_Rep` and `Suc_Rep`:

```
axiomatization Zero_Rep :: ind and Suc_Rep :: ind ⇒ind where
 Suc_Rep_inject:  Suc_Rep x = Suc_Rep y ⟹x = y and
 Suc_Rep_not_Zero_Rep: Suc_Rep x ≠ Zero_Rep
```

   On this basis, the type of natural numbers is constructed via an inductive definition, the integer and rational numbers via quotient constructions, etcpp.

4. A further axiom is devoted for another form of the Hilbert-Choice operator:

```
axiomatization Eps :: ('a ⇒bool) ⇒'a
where someI: P x ⟹P (Eps P)
```

2.1.4 The Conservative Extension Methodology

An Isabelle/HOL version coming from a trusted distribution site should *only* have these axioms. Note that in the "src/HOL" folder containing the system libraries, there are many example theories and sub-sessions that actually state their own axioms; a prudent Isabelle theory evaluator should make sure that none of these sessions were included.

Besides the logic, the instance of Isabelle called Isabelle/HOL offers support for specification constructs mapped to conservative extensions schemes, i. e. a combination of type and constant declarations as well as (internal) axioms of a very particular form. We will briefly describe here *type abbreviations*, *type definitions*, *constant definitions*, *datatype definitions*, *primitive recursive definitions*, *well-founded recursive definitions* as well-as Locale constructions. We consider this as the “methodologically safe” core of the Isabelle/HOL system.

Using solely these conservative definition principles, the entire Isabelle/HOL library is built which provides a *logically safe language base* providing a large collection of theories like sets, lists, Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions.

2.1.5 Advanced Specification Constructs — Recursive Function Definitions.

Finally, there is a parser for primitive and well-founded recursive function definition syntax. For example, the sort-operation can be defined by:

$$\begin{array}{lll} \mathrm{fun} & \mathrm{ins} & :: [\alpha :: \mathrm{linorder}, \alpha\ \mathrm{list}] \Rightarrow \alpha\ \mathrm{list} \\ \mathrm{where} & \mathrm{ins}\ x\ [\,] & = [x] \\ & \mathrm{ins}\ x\ (y\#ys) & = \mathrm{if}\ x < y\ \mathrm{then}\ x\#y\#ys\ \mathrm{else}\ y\#(\mathrm{ins}\ x\ ys) \end{array} \tag{2.4}$$

$$\begin{array}{lll} \mathrm{fun} & \mathrm{sort} & :: (\alpha :: \mathrm{linorder})\ \mathrm{list} \Rightarrow \alpha\ \mathrm{list} \\ \mathrm{where} & \mathrm{sort}\ [\,] & = [\,] \\ & \mathrm{sort}(x\#xs) & = \mathrm{ins}\ x\ (\mathrm{sort}\ xs) \end{array} \tag{2.5}$$

which is compiled internally to conservative constant and type definitions by Isabelle. Note that $\alpha :: \mathrm{linorder}$ requires that the type α is a member of the *type class* $\mathrm{linorder}$. Thus, the operation sort works on arbitrary lists of type $(\alpha :: \mathrm{linorder})\mathrm{list}$ on which a linear ordering is defined. The internal (non-recursive) constant definition for the operations ins and sort is quite involved and requires a termination proof with respect to a well-founded ordering constructed by a heuristic. Nevertheless, the logical compiler will finally derive all the equations in the statements above from these definition and makes them available for automated simplification.

2.1.6 Isabelle libraries

Isabelle libraries are predefined theories for users. New theories can be defined using Isabelle specification contructs (i. e. constant definitions) and reasoning around those new definitions can be established using Isabelle lemmas. In general, the predefined libraries implement a known theories like: set theory [NPW14], a theory on natural numbers [NP00], lists [Nip13], functions . . . We have to notice that the cited theories are included in HOL[NWP13], which is an Isabelle instantiation for higher order logic. In this section we will focus on the theories used in the specification of our test theory and the diferent case studies that we will introduce to the reader.

2.1.7 Isabelle Proofs

In addition to types, classes and constants definitions, Isabelle theories can be extended by proving new lemmas and theorems. These lemmas and theorems are derived from other existing theorems in the context of the current theory. Isabelle offers various ways to construct proofs for new theorems, we distinguish two main categories: forward and backward proofs. In addition to Isabelle proofs, some external proofs can be integrated – in a logically safe way – and compiled into an Isabelle proof. We refer the interested reader to the Isabelle Reference Manual.

2.1.8 Isabelle/HOL system features

Finally, Isabelle/HOL manages a set of *executable types and operators*, i. e., types and operators for which a compilation to SML, OCaml, Scala, or Haskel is possible. Setups for arithmetic types such as int have been done allowing for different trade-offs between trust and efficiency. Moreover any datatype and any recursive function are included in this executable set (providing that they only consist of executable operators). Of particular interest for evaluators is the use of the Isar command:

$$\textbf{valid} \quad \mathrm{sort}[1, 7, 3] \tag{2.6}$$

In the context of the definitions Equation 2.4, it will compile them via the code-generator to SML code, execute it, and output:

$$[1, 3, 7] \tag{2.7}$$

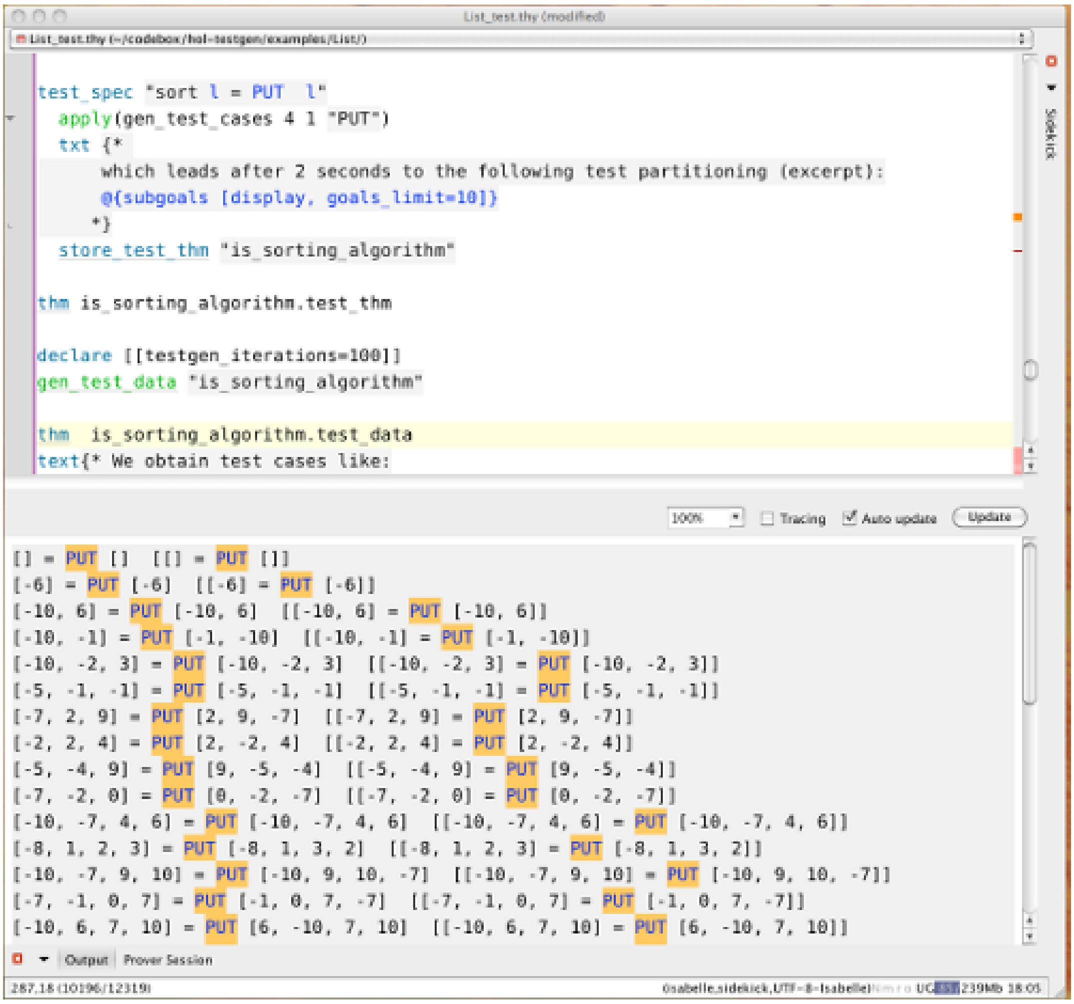


Figure 2.2: A HOL-TESTGEN session: This may be a proof state in a test theorem development, a list of generated test data or a list of test-hypothesis. After test data generation, a test script is generated that drives the test, resulting in a test trace

This provides an easy means to inspect constructive definitions and to get easy feedback for given test examples for them. See the part "Code generation from Isabelle/HOL theories" by Florian Haftmann from the Isabelle system documentation for further details..

2.2 HOL-TestGen

2.2.1 The HOL-TESTGEN workflow and system architecture

Using Isabelle as a symbolic computation environment, i. e., as a framework for implementing HOL-TESTGEN, allows us to profit from the Isabelle infrastructure in many ways. For example, HOL-TESTGEN inherits from Isabelle a document-centric workflow: the user extends existing library-theories by a new *test theory* modeling a specific application domain, by test specifications, by proofs for rules that support the overall process and by test set-ups, while the system provides essentially editing and a stepwise validation/execution functionality for these documents. Overall, these documents can be seen as formal and technically checked test plan of a program under test. Figure 2.2 shows a screenshot of HOL-TESTGEN. Besides processing these documents interactively, the user can also process them in batch mode, e. g., for integrating the test data generation into an automated build process of the program under test.

The HOL-TESTGEN workflow is, conceptually, divided into five distinct phases: first, the *Test Specific-*

ation Phase in which die program under test is modeled and the test specification is written. Second, the *Test Case Generation Phase* in which the abstract test cases are generated. Third, in the *Test Data Generation Phase* (also called Test Data Selection Phase) we chose (at least) one representative, i. e., a concrete test data that is processable by the program under test. Fourth, during the *Test Execution Phase*, the implementation is run with the selected test. Finally, during the *Test Result Verification Phase*, the behavior of the program under test is checked against the specification of the test case.

Recall Figure 2.1, which shows a brief overview over the system architecture supporting this workflow: the first three phases (writing the test specification and the generation of test cases and test data) take place in an environment based on Isabelle/**hol!**. Thus, the user of HOL-TESTGEN can profit from most features (e. g., proving properties over the test specification, transforming the specification into a form that is more suitable for the generation of test cases). After the successful generation of test data, the user can either export a *test script* or a file containing the test data in an **xml!**-like representation. The generated test script is an SML script that, together with a test harness provided by HOL-TESTGEN, can be executed independently from HOL-TESTGEN using an arbitrary SML compiler.[2] By exploiting the various foreign language interfaces of the different SML compilers, this allows for an automated setup for testing implementations in programming languages such as Java, C, SML, any language running on the .net environment, or implementations accessible via Web service calls (e. g., based on widely-used standards such as WSDl). Exporting the test data using an **xml!**-like representation allows for using the test data together with domain-specific test drivers, e. g., for testing the compliance of network firewalls.

2.3 The approach to test case generation and test data selection

As input of the test case generation phase, the *test specification*, one might expect a special format like $\mathrm{pre}\ (x) {\rightarrow} \mathrm{post}\ (x)\ (PUT\ (x))$. This rules out trivial instances such as $3 < PUT\ (x)$ or just $PUT\ (x)$ (meaning that PUT must evaluate to true for x). We do not impose any other restriction on a specification other than the final test statements being executable, i. e., the result of the process can be compiled into a test program.

Processing a test specification, our test case generation procedure (called $\mathrm{gen\_test\_case\_tac}$) can be separated into the following phases which were organized to the conceptual algorithm shown in Figure 2.3. The phases are implemented by tactics that are largely re-configurable.

The Pre-normalizer is an initial phase where definitions of the test specification may be unfolded. Its default is just a simplification tactic.

The Chooser selects (splitting) "redexes," i. e., subterms in the current clause lists on which case-splitting rules will be applied. The default are free variables of a type stemming from a datatype definition such as αlist, if _ then _ else _ expressions as well as matching expressions $\mathrm{case\_of}[] \Rightarrow \_ \mid (\_\#\_) \Rightarrow \_$. The chooser also produces heuristically a ranking among these splitting redexes.

The Splitter executes case-splitting rules for the selected redexes. In the default, this includes the generation of datatype exhaustion theorems as discussed in subsubsection 2.3.1, or splitting rewrites (see 2.1e).

The Normalizer applies the tableaux calculus (see Table 2.1) to split the list of subgoals into *Horn-clause normal form* (**hcnf!**). Finally, by re-ordering the clauses, the calls of the program under test are rearranged such that they occur only in the conclusion, where they must occur at least once. These re-ordered **hcnf!** clauses are called to be in *testing normal form* (**tnf!**), if the conclusion is an executable term.

[2] As the code generator of HOL-TESTGEN is based on the code-generator framework provided by Isabelle/**hol!**, we can quite easily generate test scripts in languages such as Scala, F#, Haskell, or OCaml.

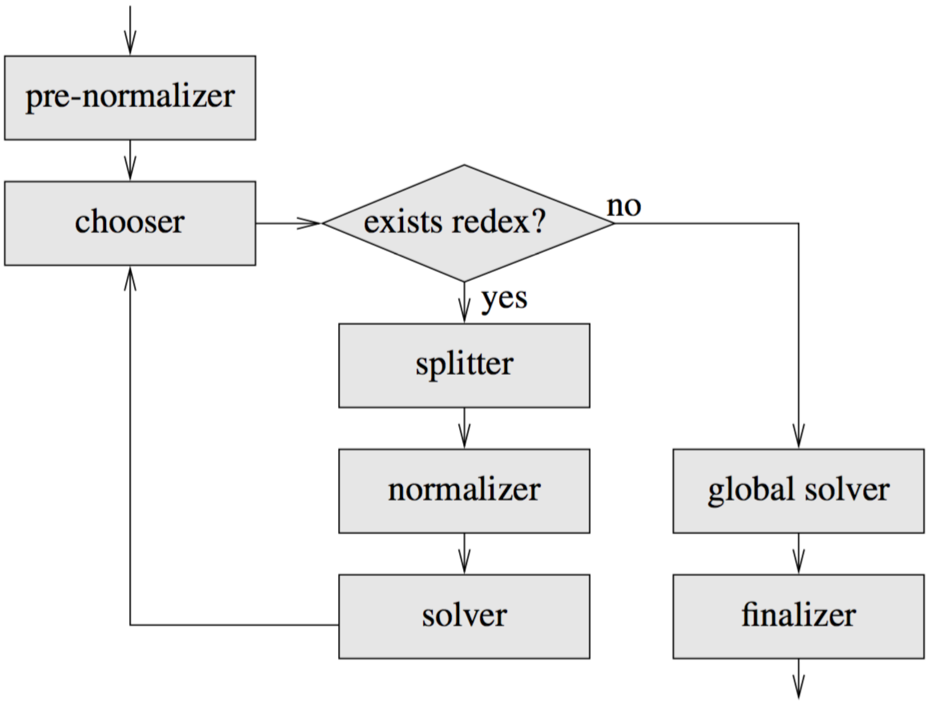


Figure 2.3: A high-level description of the algorithm: after a chooser-phase, where subterms were marked for splitting (default: free variables in the test specification), a splitter introduces case-splits of the clauses in a proof state (default: datatype exhaustion theorem, conditionals), while the normalizer brings the list of clauses into **tnf!** . Several solvers attempt to eliminate clauses with unsatisfiable constraints (representing vacuous test cases), and tries to eliminate redundant (subsumed) cases. The finalizer simplifies again the logical structure of the testing theorem by introducing explicit uniformity-hypothesis

The Solver attempts to eliminate horn-clauses with unsatisfiable constraints. In the default, this is configured as just a rewriter. A finally applied variant of the solver, which applies a more powerful combination of Isabelle decision procedures, is applied when no more redexes have been found. This final solving attempts also tries to eliminate redundant cases.

The Finalizer introduces for all remaining free variables the uniformity-hypothesis (cf. subsubsection 2.3.1).

The gen_test_case_tac procedure performs these steps until no more redexes were found. In the subsequent sections, we discuss two key components of the overall test case generation process, namely two test specific rule-schemata as well as the normalizer constructing the actual test cases. We will briefly sketch the constraint solver used to find concrete instances of a test case, and conclude with a discussion of coverage criteria.

2.3.1 Test cases generation with explicit test-hypothesis

We apply two test specific rule schema that start respectively finalize the normalization process. These rule schema introduce certain subformula which can be seen as a *testing-hypothesis* or proof-obligation and encapsulates them via a constant symbol THYP (which is semantically defined as just an identity) from the rest of the test cases. Following the terminology of Gaudel [Gau95], we distinguish *regularity* and *uniformity*-hypothesis. Note, however, that the explicit use of the hypothesis as proof-obligation inside the logic, even inside the test-theorem is specific to our framework. These two kinds of hypothesis are configured as default into our system, but alternative test-hypothesis are discussed in [BW13].

Using regularity-hypothesis in splitting

In the following, we address the problem of test case generation for universally quantified (or, equivalently, free variables) ranging over recursive datatypes such as lists or trees. For testing recursive data structures, the following form of a *regularity-hypothesis* [Gau95] has been suggested:

$$\frac{\begin{array}{c}[|x| < k] \\ \vdots \\ P\,x\end{array}}{P\,x} \tag{2.8}$$

This rule formalizes the hypothesis: assuming that a predicate P is true for all data x whose *size* (denoted by $|x|$) is less than a given depth k, P is always true. The original rule can be viewed as a meta-notation: In a rule for a concrete datatype, the premises $|x| < k$ can be expanded to several premises enumerating constructor terms.

Instead of this (deliberately) unsound rule, HOL-TestGen *derives* on-the-fly a special datatype exhaustion theorem; its form depends on the *depth* d and the structure of the datatype of x. For the user-defined value $d = 3$ and for the type α list, we have:

$$\frac{\begin{array}{c}[x = [\,]] \\ \vdots \\ P(x)\end{array} \quad \bigwedge a.\ \begin{array}{c}[x = [a]] \\ \vdots \\ P(x)\end{array} \quad \bigwedge a\ b.\ \begin{array}{c}[x = [a, b]] \\ \vdots \\ P(x)\end{array} \quad \bigwedge a\ b\ c.\ \begin{array}{c}[x = [a, b, c]] \\ \vdots \\ P(x)\end{array} \quad \mathrm{THYP}(H)}{P(x)} \tag{2.9}$$

where the explicit test-hypothesis "regularity" has the form $H = (\forall x.\ |x| < 4 \to P(x)) \to \forall x.\ P\,x$.

In the sequel, we will show the effect of the datatype exhaustion theorem on our running example presented in the introduction. The presentation of the testing theory, in our case the definition of the datatype list and the recursive function definitions ins and sort Equation 2.4, is already complete. The test specification "the program under test should be a sorting algorithm" is straight-forward:

$$\textbf{testspec}\ \text{test:}\quad PUT(x) = \text{sort}(x) \tag{2.10}$$

The chooser will detect as redex the free variable x of type list; the splitter will apply the datatype exhaustion theorem accordingly. The resulting proof state reads as follows:

$$\begin{array}{rl} \text{test}: & PUT(x) = \text{sort}(x) \\ 1. & PUT([]) = \text{sort}([]) \\ 2. & \bigwedge a.\ PUT([a]) = \text{sort}([a]) \\ 3. & \bigwedge a\, b.\ PUT([a,b]) = \text{sort}([a,b]) \\ 4. & \bigwedge a\, b\, c.\ PUT([a,b,c]) = \text{sort}([a,b,c]) \\ 5. & \text{THYP}(\forall x.\ |x| < 4 \rightarrow PUT(x) = \text{sort}(x)) \rightarrow \forall x.\ PUT(x) = \text{sort}(x)) \end{array} \tag{2.11}$$

Elementary rewriting by the definitions of sort in Equation 2.4 and the normalization process described in subsection 2.3.2 will turn our test specification into the final test-theorem.

Using uniformity-hypothesis in the finalizer

Uniformity-hypothesis have the form:

$$\text{THYP}(\exists x_1 \ldots x_n.\ P\, x_1, \ldots, x_n \rightarrow \forall x_1 \ldots x_n.\ P\, x_1 \ldots x_n) \tag{2.12}$$

and were used in the finalizer phase of the test-generation procedure. Semantically, this kind of hypothesis expresses the following: whenever a test case is passed successfully for one data of this test case, the program behaves correctly for *all* data of this test case. The derived rule in natural deduction format expressing this kind of test theorem transformation reads as follows:

$$\frac{P\ ?x_1 \ldots ?x_n \qquad \text{THYP}(\exists x_1 \ldots x_n.\ P\, x_1 \ldots x_n \rightarrow \forall x_1 \ldots x_n.\ P\, x_1 \ldots x_n)}{\forall x_1 \ldots x_n.\ P\, x_1 \ldots x_n} \tag{2.13}$$

where the $?x_i$ are just meta variables, i. e., place-holders for arbitrary terms. This rule can also be applied for arbitrary formulae containing free variables since universal quantifiers may be introduced for them.

In contrast to our presentation in introductory examples, we use meta-variables and meta-implications which can be processed by Isabelle's deduction engine directly.

2.3.2 Normal form computations

At the heart of the test case generation, i. e., the generation of the testing theorem, lies a normal form computation process similar to the `DNF`-computation pioneered by Dick and Faivre [DF93]. In contrast to the latter, however, we chose to adopt a Horn-clause normal form (**hcnf!**) used in the usual Isabelle proof states. In a classical logic like **hol!**, Horn-clauses like: $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$ are logically equivalent to $\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1}$. Therefore, the **hcnf!** can be viewed as a conjunctive normal form (**cnf!**). We will interpret the subgoals of a proof state as test cases, and view the assumptions A_i of each subgoal as *constraints* restricting the valid input of a test case.

In the following, we describe the tableaux, rewriting and testing normal form computations in more detail. In Isabelle/**hol!**, the automated proof procedures for **hol!** formulae depend heavily on tableaux calculi [dag96] presented as (derived) natural deduction rules. Table 2.1 presents the core tableaux calculus of **hol!**. With the notable exception of the elimination rule for the universal quantifier (see 2.1c),

Table 2.1: The Standard Tableaux Calculus for **hol!**

$$\frac{P\ ?x}{\exists x.\ P\ x} \qquad \frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}$$

(a) Quantifier Introduction Rules

$$\frac{}{t = t} \qquad \frac{}{\mathrm{true}} \qquad \frac{P \qquad Q}{P \wedge Q} \qquad \frac{\begin{array}{c}[\neg Q]\\ \vdots \\ P\end{array}}{P \vee Q} \qquad \frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q} \qquad \frac{\begin{array}{c}[P]\\ \vdots \\ \mathrm{false}\end{array}}{\neg P} \qquad \frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array} \quad \begin{array}{c}[Q]\\ \vdots \\ P\end{array}}{P = Q}$$

(b) Safe Introduction Rules

$$\frac{\forall x.\ P\ x \qquad \begin{array}{c}[P\ ?x]\\ \vdots \\ R\end{array}}{R} \qquad \frac{\forall x.\ P\ x \qquad \begin{array}{c}[\forall x.\ P\ x,\ P\ ?x]\\ \vdots \\ R\end{array}}{R}$$

(c) Unsafe Elimination Rules

$$\frac{\mathrm{false}}{P} \qquad \frac{P \wedge Q \qquad \begin{array}{c}[P, Q]\\ \vdots \\ R\end{array}}{R} \qquad \frac{P \vee Q \quad \begin{array}{c}[P]\\ \vdots \\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots \\ R\end{array}}{R} \qquad \frac{P \to Q \quad \begin{array}{c}[\neg P]\\ \vdots \\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots \\ R\end{array}}{R} \qquad \frac{\exists x.\ P\ x \qquad \bigwedge x. \quad \begin{array}{c}[P\ x]\\ \vdots \\ Q\end{array}}{Q}$$

$$\frac{P = Q \qquad \begin{array}{c}[P, Q]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[\neg P, \neg Q]\\ \vdots \\ R\end{array}}{R}$$

(d) Safe Elimination Rules

$$P(\text{if } C \text{ then } A \text{ else } B) = (C \to P(A)) \wedge (\neg C \to P(B))$$
$$P(\text{case } x \text{ of } \mathsf{Nil} \Rightarrow F \,|(a\#r) \Rightarrow G\ a\ r) = (x = [] \to P(F)) \wedge (\exists a\, t.\ x = a\#t \to P(G\ a\ t))$$

(e) (Splitting)-Rewrites

any rule application leads to a logically equivalent proof state: therefore, all rules (except $\forall$ elimination) are called *safe*. When applied bottom up in backwards reasoning (which may introduce meta-variables explicitly marked in Table 2.1), the technique leads in a deterministic manner to a **hcnf!**. Note, however, that test cases are not necessarily minimal: there may be test cases that overlap. In practice, however, this occurs seldom in specifications that are based on distinct constructors of data types.

Coming back to our running example sort, the proof-state shown in Equation 2.11 is transformed in the normalization phase as follows. Rewriting the definitions of sort yields:

$$\begin{array}{rl}\text{test}: & PUT(x) = \mathrm{sort}(x)\\ 1. & PUT([]) = []\\ 2. & \bigwedge a.\ PUT([a]) = \mathrm{ins}\ a\ []\\ 3. & \bigwedge a\, b.\ PUT([a,b]) = \mathrm{ins}\ a\ (\mathrm{ins}\ b\ [])\\ 4. & \bigwedge a\, b\, c.\ PUT([a,b,c]) = \mathrm{ins}\ a\ (\mathrm{ins}\ b\ (\mathrm{ins}\ c\ []))\\ 5. & \mathrm{THYP}(\forall x.\ |x| < 4 {\rightarrow} PUT(x) = \mathrm{sort}(x)) {\rightarrow} \forall x.\ PUT(x) = \mathrm{sort}(x))\end{array} \tag{2.14}$$

and further rewrite steps unfolding ins result in:

$$\begin{array}{rl}\text{test}: & PUT(x) = \mathrm{sort}(x)\\ 1. & PUT([]) = []\\ 2. & \bigwedge a.\ PUT([a]) = [a]\\ 3. & \bigwedge a\, b.\ PUT([a,b]) = \text{if } a \le b \text{ then } [a,b] \text{ else } [b,a]\\ 4. & \bigwedge a\, b\, c.\ PUT([a,b,c]) = \mathrm{ins}\ a\ (\text{if } b \le c \text{ then } [b,c] \text{ else } [c,b])\\ 5. & \mathrm{THYP}(\forall x.\ |x| < 4 {\rightarrow} PUT(x) = \mathrm{sort}(x)) {\rightarrow} \forall x.\ PUT(x) = \mathrm{sort}(x))\end{array} \tag{2.15}$$

This proof-state is in normal-form, the overall algorithm continues therefore executing the main loop shown in Figure 2.3. The chooser picks in this iteration the conditionals in subgoals 3 and 4, while the splitter uses the splitting rewrites together 2.1e with the safe introduction rules in 2.1b to compute the following successor proof-state (some elementary rewriting on arithmetic is omitted here):

$$\begin{array}{rl}\text{test}: & PUT(x) = \mathrm{sort}(x)\\ 1. & PUT([]) = []\\ 2. & \bigwedge a.\ PUT([a]) = [a]\\ 3. & \bigwedge a\, b.\ [\![a \le b]\!] \Longrightarrow PUT([a,b]) = [a,b]\\ 4. & \bigwedge a\, b.\ [\![b < a]\!] \Longrightarrow PUT([a,b]) = [b,a]\\ 6. & \bigwedge a\, b\, c.\ [\![b \le c]\!] \Longrightarrow PUT([a,b,c]) = \mathrm{ins}\ a\ [b,c]\\ 7. & \bigwedge a\, b\, c.\ [\![c < b]\!] \Longrightarrow PUT([a,b,c]) = \mathrm{ins}\ a\ [c,b]\\ 8. & \mathrm{THYP}(\forall x.\ |x| < 4 {\rightarrow} PUT(x) = \mathrm{sort}(x)) {\rightarrow} \forall x.\ PUT(x) = \mathrm{sort}(x))\end{array} \tag{2.16}$$

After a few further iterations (and the finalization phase introducing the clauses representing the used uniformity hypothesis), our algorithm will result in the test-theorem shown in Equation 2.16.

Our test specifications may contain higher-order constants and all sorts of bounded quantifiers (e. g., over lists; their elimination is part of the rewrite rule set not discussed in detail here). Moreover, the procedure also works for unbounded quantifiers ranging over datatypes (although in the default setup, only universal quantifiers in positive occurrence and existential quantifier in negative occurrence will be selected in the chooser). However, the procedure leaves quantifiers of other types (such as higher-order function types or sets) unchanged and leaves it to suitable (user-programmed) procedures in the constraint solver.

The chooser also performs an internal bookkeeping of the variables introduced in the process; thus, a splitting of meta-quantified variables a, b, c introduced by the datatype exhaustion theorem is avoided to ensure termination. Finally, observe that the number of test cases that the algorithm constructs is finite, but test cases in itself have usually infinitely many witnesses (test data). This is in sharp contrast to all model-checking related approaches that attempt to approximate infinite datatypes early, and usual in an ad-hoc manner.

A final normalization step brings the proof state in **hcnf!** into a particular variant of it. In particular, this final transformation eliminate subgoals like:

$$\llbracket\neg(PUT\ x = c);\ \neg(PUT\ x = d)\rrbracket \Longrightarrow A_{n+1}\,, \tag{2.17}$$

and transform them into the equivalently clause:

$$\llbracket\neg(A_{n+1})\rrbracket \Longrightarrow PUT\ x = c \vee PUT\ x = d\,. \tag{2.18}$$

We call this form of Horn-clauses *testing normal form* (**tnf!**), if after the normalization the conclusions of all horn-clauses are executable. Not all specifications can be converted to **tnf!**. For example, if the specification does not make a suitably strong constraint over program PUT, in particular if PUT does not occur in the specification. In such cases, gen_test_case stops with an exception.

2.3.3 Test data generation by constraint solving

The test data generator called gen_test_data implements the test data selection phase. It extracts from a given test theorem the constraints of each test case and starts a constraint resolution phase for the latter. Our constraint solver consists of a chain of solvers, filtering smaller constraints from more complex ones. The first level is represented by auto [Pau99] (an Isabelle standard tactic combining a tableaux-prover with a rewrite engine and a linear arithmetic procedure). Remaining unsolved constraints were passed to the second level, an own symbolic random-solver (a number of random ground instances were substituted for the variables occurring in the constraint which is then passed auto). The next level is the compiling random-solver quick_check [BN04] (an Isabelle standard procedure that compiles all constraints to code and searches solutions by test-and-verify random values). Finally, we extended an integration of external **smt!**-solvers available in recent versions of Isabelle, in particular as Z3[BTV09], by constructing from its counter-models substitutions and by verifying them inside Isabelle.

The precise order of solvers and the number of repetitions is user-defined and highly reconfigurable. The choice for the default order sketched above is entirely pragmatic—it turned out to be the fastest for the examples we check, and actually changed over the years according to the availabilities and the increasing power of its components.

Unresolved constraints (marked by RSF in our examples) where still represented in test data statements and thus mark possibly inconclusive tests in the test-execution phase.

The test case generation phase can be very costly in some realistic examples; in others, it is the test data generation which is the bottle neck. Massaging a test theorem into a form that permits the solver to solve all constraints is tantamount for using HOL-TestGen effectively. This form of massage, possibly resulting in new, hand-proven lemmas or axiomatically stated facts to be inserted into the test case generation and test data generation procedure is *the* activity that gives HOL-TestGen an interactive flavor, but also makes the system so powerful.

In our example Equation 2.15, gen_test_data produces the 9 ground instances for the non-trivial test cases in a fraction of a second; in this case, the work is solely done by our symbolic random-solver procedure.

2.3.4 Test-adequacy and theoretical properties

In the following, we discuss the theoretical and practical properties, e. g., the underlying test-adequacy criteria, of our black-box test case generation approach. Obviously, the heart of it is a decomposition of the test specification into a normal form and the construction of test cases for each of its clauses. This is in the tradition of [BGM91] and the work of Dick and Faivre [DF93]. Besides the conceptually minor difference that our basic TNF is essentially a CNF, there is the major difference that we strive to solve a **smt!** (**smt!**) problem for each clause (i. e., test case), and that each clause is also normalized with respect to if _ then _ else _ and datatype induced case-statements.

Definition: Normal Test Specifications $\mathbf{tnf!}_{\mathbf{E/d}}(\mathrm{TS})$. The ***tnf!*** (**tnf!**) modulo a theory E of depth d from a test specification TS has the following properties:

1. all constraints $C_{i,1}, \ldots, C_{i,k}$ do neither contain if _ then _ else _ nor datatype induced case-statements, i. e., they are *fully splitted*,
2. all datatype-generated free variables in TS were splitted at least d times, i. e.. at least d times an exhaustion rule must have been applied to this variable of its descendants,
3. the oracles have the form $P_1(PUT, c_1, \ldots, c_k) \vee \cdots \vee P_m(PUT, c_1, \ldots, c_k)$, where the P_j are closed and executable,
4. all constraints of all clauses (i. e., test cases) are satisfiable modulo E; i. e., $\exists x_1, \ldots, x_k.\ C_{i,1}(x_1, \ldots, x_k) \wedge \cdots \wedge C_{i,k}(x_1, \ldots, x_k)$ are true, and
5. all test-hypothesis are non-redundant, i. e., $\mathrm{THYP}(X) \neq \mathrm{true}$.

$\mathbf{tnf!}_E$ is essentially a **cnf!** (**cnf!**) since existential quantifiers can be eliminated via meta-variables, and the implications into disjunctions.

Definition: $\mathbf{tnf!}_{\mathbf{E/d}}$**-Test-adequacy for TS.** A set of test cases for a test specification TS is $\mathbf{tnf!}_E$-adequate, if a $\mathbf{tnf!}_{E/d}(TS)$ normal form could be computed for TS and the set of test cases contains at least one test case for each clause.

Theorem: HOL-TestGen approximates $\mathbf{tnf!}_{E/d}$-adequacy.

Proofsketch: For the case that we have a complete decision procedure for E, for example, for a Noetherian and convergent set of rewrite rules for the entire theory used in the test specification, the proposition follows by construction. The question arises, what happens if solvers fail (are not a decision procedure). In these cases, there are more clauses with undetected unsatisfiable constraints, or satisfiable constraints for which the constraint solver are unable to construct a solution. These cases are explicitly marked in the resulting test-driver and will result in "inconclusive tests," i. e., tests which require further human inspection.

Theorem: HOL-TestGen is a correct testing procedure, i. e., if a test theorem of the form

$$\frac{\mathrm{TC}[1] \cdots \mathrm{TC}[n] \qquad H_1 \cdots H_m}{\mathrm{TS},} \tag{2.19}$$

is constructed with all $\mathrm{TC}[i]$ in $\mathbf{tnf!}_{E/d}(\mathrm{TS})$, then the implication $\mathrm{TC}[1] \wedge \cdots \wedge \mathrm{TC}[n] \wedge H_1 \wedge \cdots H_m \rightarrow$ TS is logically valid.

Proofsketch: The entire procedure is based on the application of derived rules in **hol!**. (We assume consistency of **hol!** and its correct implementation in Isabelle; However, if one has serious doubts into the latter, it is perfectly possible to generate for the entire derivation of the test-theorem a proof object for **hol!** and check the latter independently from Isabelle).

Theorem: HOL-TestGen is a complete testing procedure, i. e., $\mathrm{TS} \rightarrow \mathrm{TC}[1] \wedge \cdots \wedge \mathrm{TC}[n] \wedge H_1 \wedge \cdots H_m$ holds.

Proofsketch: The construction of the normal form uses only the "safe" (i. e., logically equivalent) rules of Table 2.1, plus rewrite rules for the user-defined operations.

Running our sorting example on standard hardware requires less than a second for depth $d = 3$ (10 cases), less than five seconds for depth $d = 4$ (34 cases). For depth $d = 5$ (154 cases) the generation already requires around 15 minutes. At the first glance, this seems to indicate that the HOL-TestGen approach does not scale well. Especially, as random testing tools like QuickCheck [CH00] promise to check similar properties with several thousands of test cases within 15 minutes. We argue, however, that an in-depth analysis of the situation refutes this conclusion: 1. on average, a purely random testing approach needs to check $4\,000\,000$ test cases[3] to hit the 153 cases of the **tnf!**$_{E/5}$, 2. in many application scenarios of model-based testing, a small number of significant tests is crucial to make testing practically feasible, and 3. **tnf!**$_{E/5}$ is indeed what the sorting problem imposes, that the algorithmic structure of the problem motivates a certain structure of the test cases. While the efficiency of QuickCheck can be improved by *manually* providing specialized test case generators, our approach reveals the underlying problem structure *automatically*.

Finally, the global solver also attempts to eliminate redundant test cases. Since this analysis is costly and in general impossible—subsumption of a test case ϕ in a test case ψ boils down to decide $\psi \to \phi$—we have to live with the fact that test cases are *not* partitions and we will have more test data in practice than needed in a minimal set of test cases in which the classes of solutions are strictly disjoint. Our procedure is well-behaved for medium-size examples shown throughout this paper. The effect of generating redundant test cases can be annoying in very large examples in our experience.

2.4 Summary of new HOL-TestGen Features developed in EURO-MILS

From the version 1.6.0, which was publicly released prior to the EURO-MILS project, to the currently released version 1.8.0 (pre), there are a number of improvements:

1. The Bug-Tracking system of the HOL-TestGen project reports 6 closed bugs, most notable related to performance issues and not sufficient splitting for variables with nested or mutually recursive data-types. (It introduced 2 new performance bugs though.)
2. New command option syntax for configuration settings in Isar.
3. New naming conventions for concrete and abstract test theorems.
4. Substantially extended library: the Monad-theory grew by 2000 loc's together with machinery to generate for Extended-Finite-State-Machine setups for the splitting machinery.
5. Re-organization of the example suite: separation into unit, sequence and reactive sequence test-examples. New reference examples Bank and MyKeOS for sequence testing.

[3] For lists of length n, we generate $n!$ test cases (every permutation). A purely random testing-bases approach that generates lists of length n for integers up to k needs to generate $\binom{k}{n}$ test cases for ensuring the inclusion of all $n!$ permutations.

Part II

Test-Generation for Concurrent OS Code

Chapter 3

Theoretical and Technical Foundations: Testing Concurrent Programs

3.1 Introduction

The verification of systems combining soft- and hardware, such as modern avionics systems, asks for combined efforts in test and proof: In the context of certifications such as EAL5 in Common Criteria, the required formal security models have to be linked to system models via refinement proofs, and system models to code-level implementations via testing techniques. Tests are required for methodological reasons ("Did we get the system model right? Did we adequately model the system environment?") as well as economical reasons (state of the art deductive verification techniques of machine-level code are *practically* limited to systems with ca. 10 kLOC of size, see [KEH$^+$09]).

Our work stands in the context of an EAL5+ certification project [1] of the commercial OS-concurrent operating system called PikeOS, used in avionics applications; PikeOS [SYS, SYS13a, SYS13b] is a virtualizing separation kernel in the tradition of L4-microkernels [HHL$^+$97]. Our work complements the testing initiative by a model-based testing technique linking the formal system model of the PikeOS inter-process communication against the real system. This is a technical challenge for at least the following reasons:

- the system model is a transaction machine over a very rich state,
- system calls were implemented by internal, uninterruptible "atomic actions" reflecting the L4-microkernel concept; atomic actions define the granularity of our concurrency model, and
- the security model is complex and, in case of aborted system calls, leads to non-standard notions of execution trace interleaving.

To meet these challenges, we need to revise conceptual and theoretical foundations.

- We use symbolic execution techniques to cope with the large state-space; their inherent drawback to be limited to relatively short execution traces is outweighed by their expressive power,
- we extend the "monadic test approach" proposed in [BW07, BW13] to a test-method for concurrent code. It combines an IO-automata view [LT89] with extended finite state machines [Gil62] using abstract transitions, and
- we need an adaption of concurrency notions, a "semantic view" on partial-order reduction and its integration into interleaving-based coverage criteria.

This sums up to a novel, tool-supported, integrated test methodology for concurrent OS-system code, ranging from an abstract system model in Isabelle/HOL, complemented embedding of the latter into our monadic sequence testing framework, our setups for symbolic execution down to generation of test-drivers and the code instrumentation.

[1] www.euromils.eu

In this chapter we will introduce a set of technical and theoretical contributions to test concurrent programs. On theoretical side, we present the monadic test approach from an IO-Automata view in section 3.2 then we show how it can be used to express concurrent test scenarios in section 3.4. In section 3.3 we state our refinement relation, which help us to express a familly of conformance relations to link the abstract model with the concrete implementation. On the technical side, we will show how Isabelle is used as an abstract test case generator in section 3.5. Finally, our techniques to build test drivers for concurrent code are presented in section 3.7.

3.2 Monads Theory

The obvious way to model the state transition relation of an automaton A is by a relation of the type $(\sigma \times (\iota \times o) \times \sigma)\,\mathrm{set}$; isomorphically, one can also model it via:

$$\iota \Rightarrow (\sigma \Rightarrow (o \times \sigma)\,\mathrm{set})$$

or for a case of a deterministic transition function:

$$\iota \Rightarrow (\sigma \Rightarrow (o \times \sigma)\,\mathrm{option})$$

In a theoretic framework based on classical higher-order logic (HOL), the distinction between "deterministic" and "non-deterministic" is actually much more subtle than one might think: since the transition function can be underspecified via the Hilbert-choice operator, a transition function can be represented by

$$\mathit{step}\ \iota\,\sigma = \{(o, \sigma')|\,\mathrm{post}(\sigma, o, \sigma')\}$$

or:

$$\mathit{step}\ \iota\,\sigma = \mathit{Some}(\mathrm{SOME}(o, \sigma').\ \mathrm{post}(\sigma, o, \sigma'))$$

for some post-condition post. While in the former "truly non-deterministic" case *step* can and will at run-time choose different results, the latter "underspecified deterministic" version will decide in a given model (so to speak: the implementation) always the same way: a choice that is, however, unknown at specification level and only declaratively described via post. For the system in this paper and our prior work on a processor model [BFNW13], it was possible to opt for an underspecified deterministic stepping function.

We abbreviate functions of type $\sigma \Rightarrow (o\times\sigma)\,\mathrm{set}$ or $\sigma \Rightarrow (o\times\sigma)\,\mathrm{option}$ $\mathrm{MON}_{\mathrm{SBE}}(o, \sigma)$ or $\mathrm{MON}_{\mathrm{SE}}(o, \sigma)$, respectively; thus, the aforementioned state transition functions of io-automata can be typed by $\iota \to \mathrm{MON}_{\mathrm{SBE}}(o, \sigma)$ for the general and $\iota \to \mathrm{MON}_{\mathrm{SE}}(o, \sigma)$ for the deterministic setting. If these function spaces were extended by the two operations *bind* and *unit* satisfying three algebraic properties, they form the algebraic structure of a *monad* that is well known to functional programmers as well as category theorists. Popularized by [Wad92], monads became a kind of standard means to incorporate stateful computations into a purely functional world.

Since we have an underspecified deterministic stepping function in our system model, we will concentrate on the latter monad which is called the *state-exception monad* in the literature.

The operations *bind*, which represent sequential composition with value passing, and *unit*, which represent the embedding of a value into a computation, are defined for the special-case of the state-exception monad as follows:

```
definition bindSE :: ('o,'σ)MON_SE ⇒('o ⇒('o','σ)MON_SE) ⇒
                      ('o','σ)MON_SE
where     bindSE f g = (λσ. case f σof None ⇒None
                               | Some (out, σ') ⇒g out σ')

definition unitSE :: 'o ⇒('o, 'σ)MON_SE ((return _) 8)
where     unitSE e = (λσ. Some(e,σ))
```

We will write $x \leftarrow m_1; m_2$ for the sequential composition of two (monad) computations m_1 and m_2 expressed by $\mathrm{bind}_{\mathrm{SE}}\, m_1(\lambda\, x.m_2)$. Moreover, we will write "return" for $\mathrm{unit}_{\mathrm{SE}}$.
This definition of $\mathrm{bind}_{\mathrm{SE}}$ and $\mathrm{unit}_{\mathrm{SE}}$ satisfy the required monad laws:

```
lemma bind_left_unit_SB : (x := returns a; m x) = m a
  by (rule ext,simp add: unit_SB_def bind_SB_def)

lemma bind_right_unit_SB: (x := m; returns x) = m
  by (rule ext, simp add: unit_SB_def bind_SB_def)

lemma bind_assoc_SB: (y := (x := m; k x); h y) =
                    (x := m; (y := k x; h y))
  by (rule ext, simp add: unit_SB_def bind_SB_def split_def)
```

On this basis, the concept of a *valid monad execution*, written $\sigma \models m$, can be expressed: an execution of a Boolean (monad) computation m of type $(\mathrm{bool}, \sigma)\ \mathrm{MON}_{\mathrm{SE}}$ is valid iff its execution is performed from the initial state σ, no exception occurs and the result of the computation is true.

```
definition validSE ::
 'σ ⇒ (bool,'σ) MONSE ⇒bool (infix ⊨15)
  where (σ ⊨m) = (m σ≠ None ∧fst(the (m σ)))
```

More formally, $\sigma \models m$ holds iff $(m\ \sigma \neq \mathrm{None} \wedge \mathrm{fst}(\mathrm{the}(m\ \sigma)))$, where `fst` and `snd` are the usual *first* and *second* projection into a Cartesian product and `the` is the projection in the `Some` a variant of the option type.
We define a *valid test-sequence* as a valid monad execution of a particular format: it consists of a series of monad computations $m_1 \ldots m_n$ applied to inputs $\iota_1 \ldots \iota_n$ and a post-condition P wrapped in a return depending on observed output. It is formally defined as follows:

$$\sigma \models o_1 \leftarrow m_1\ \iota_1; \ldots ; o_n \leftarrow m_n\ \iota_n; \mathrm{return}(P\ o_1 \cdots o_n)$$

The notion of a valid test-sequence has two facets: On the one hand, it is executable, i. e., a *program*, iff $m_1, \ldots, m_n, P$ are. Thus, a code-generator can map a valid test-sequence statement to code, where the m_i where mapped to operations of the SUT interface. On the other hand, valid test-sequences can be treated by a particular simple family of symbolic executions calculi, characterized by the schema (for all monadic operations m of a system, which can be seen as its step-functions):

$$\frac{}{(\sigma \models \mathrm{return}\, P) = P} \tag{3.1a}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = None}{(\sigma \models ((s \leftarrow m\ \iota; m'\ s))) = False} \tag{3.1b}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\ \iota; m'\ s) = (\sigma' \models m'\ b)} \tag{3.1c}$$

Which corresponds to the following Isabelle/HOL implementation:

```
lemma exec_unitSE [simp]: (σ ⊨(return P)) = (P)
  by(auto simp: validSE_def unitSE_def)

lemma exec_bindSE_failure:
  A σ = None ⟹¬(σ ⊨((s ←A ; M s)))
  by(simp add: validSE_def unitSE_def bindSE_def)
```

```
lemma exec_bindSE_success:
  A σ = Some(b,σ') ⟹(σ ⊨((s ←A ; M s))) = (σ' ⊨(M b))
  by(simp add: validSE_def unitSE_def bindSE_def )
```

This kind of rules is usually specialized for concrete operations m; if they contain pre-conditions C_m (constraints on ι and state), this calculus will just accumulate those and construct a constraint system to be treated by constraint solvers used to generate concrete input data in a test.

3.2.1 An Example: MyKeOS.

To present the effect of the symbolic rules during symbolic execution, we present a toy OS-model. MyKeOS provides only three atomic actions for *allocation* and *release* of a resource (for example a descriptor of a communication channel or a file-descriptor). A *status* operation returns the number of allocated resources. All operations are assigned to a thread (designated by `thread_id`) belonging to a task (designated by `task_id`, a Unix/POSIX-like *process*); each thread has a thread-local counter in which it stores the number (the status) of the allocated resources. The input is modeled by the data-type:

```
datatype in_c = alloc task_id thread_id nat
              | release task_id thread_id nat
              | status task_id thread_id

datatype out_c = alloc_ok | release_ok | status_ok nat
```

where `out_c` captures the return-values. Since `alloc` and `release` do not have a return value, they signalize just the successful termination of their corresponding system steps. The global table `var_tab` (corresponding to our symbolic state σ) of thread-local variables is modeled as partial map assigning to each active thread (characterized by the pair of task and thread id) the current status:

```
type_synonym thread_local_var_tab = (task_id ×thread_id) ⇀int
```

The operation have the precondition that the pair of task and thread id is actually defined and, moreover, that resources can only be released that have been allocated; the initial status of each defined thread is set to 0. The **hol!** representation of the preconditions and postconditions is:

```
fun precond :: thread_local_var_tab ⇒in_c ⇒bool
where
 precond σ(alloc c no m)  = ((c,no) ∈dom σ)
| precond σ(release c no m) = ((c,no) ∈dom σ∧
                             (int m) ≤the(σ(c,no)))
| precond σ(status c no) = ((c,no) ∈dom σ)

fun postcond :: in_c ⇒thread_local_var_tab ⇒
               (out_c ×thread_local_var_tab) set
where
 postcond (alloc c no m) σ=
    { (n,σ'). (n = alloc_ok ∧
               σ'=σ((c,no)↦ the(σ(c,no)) + int m))}
| postcond (release c no m) σ=
   {(n,σ'). (n = release_ok ∧
            σ'=σ((c,no)↦ the(σ(c,no)) - int m))}
| postcond (status c no) σ=
   {(n,σ'). (σ=σ' ∧
           (∃ x. status_ok x = n ∧x = nat(the(σ(c,no)))))}
```

Depicted as an extended finite state-machine (EFSM), the operations of our system model SPEC are specified as shown in Figure 3.1.

A transcription of an EFSM to HOL is reprented by a locale (see section 3.5), which is instantiated by the above definitions of pre-post conditions and:

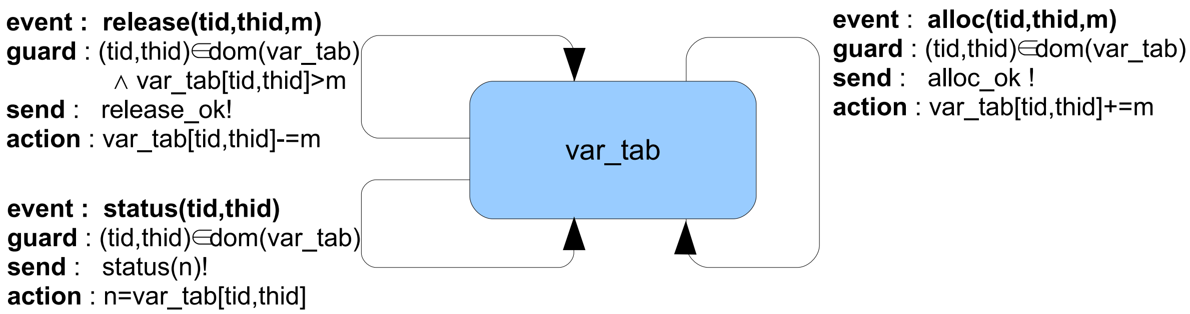


Figure 3.1: SPEC: An Extended Finite State Machine for MyKeOS.

```
definition strong_impl ::
['σ⇒'ι⇒bool, 'ι⇒('o,'σ)MON_SB] ⇒'ι⇒('o, 'σ)MON_SE
where strong_impl pre post ι=
      (λ σ. if pre σι
           then Some(SOME(out,σ'). (out,σ') ∈post ισ)
           else None)

definition SPEC = (strong_impl precond postcond)
```

where `SPEC` represent the instantiation of an EFSM with the semantics of MyKeOS. We show a concrete symbolic execution rule derived from the definitions of the `SPEC` system transition function, e. g., the instance for Equation 4.1:

$$\frac{(\mathit{tid},\mathit{thid}) \in \mathrm{dom}(\sigma) \qquad \mathrm{SPEC}\ (\mathrm{alloc}\ \ \mathit{tid}\ \mathit{thid}\ m)\ \sigma = \mathit{Some}(\mathrm{alloc\_ok},\sigma')}{(\sigma \models s \leftarrow \mathrm{SPEC}\ (\mathrm{alloc}\ \ \mathit{tid}\ \mathit{thid}\ m); m'\ s) = (\sigma' \models m'\ \ \mathrm{alloc\_ok})}$$

where $\sigma = \mathit{var\_tab}$ and $\sigma' = \sigma((\mathit{tid},\mathit{thid}) := (\sigma(\mathit{tid},\mathit{thid}) + m))$. Thus, this rule allows for computing σ, σ' in terms of the free variables $\mathit{var\_tab}$, tid, thid and m. The rules for release and status are similar. For this rule, SPEC $(\mathrm{alloc}\ \ \mathit{tid}\ \mathit{thid}\ m)$ is the concrete stepping function for the input event alloc $\mathit{tid}\ \mathit{thid}\ m$, and the corresponding constraint C_{SPEC} of this transition is $(\mathit{tid},\mathit{thid}) \in \mathrm{dom}(\sigma)$.

3.3 Conformance Relations Revisited

We state a family of test conformance relations that link the specification and abstract test drivers. The trick is done by a coupling variable res that transport the result of the symbolic execution of the specification SPEC to the expected result of the SUT.

$$\begin{aligned} &\sigma \models o_1 \leftarrow \mathrm{SPEC}\ \ \iota_1; \ldots ; o_n \leftarrow \mathrm{SPEC}\ \ \iota_n; \mathrm{return}(\mathit{res} = [o_1 \cdots o_n]) \\ &\qquad \longrightarrow \\ &\qquad\qquad \sigma \models o_1 \leftarrow \mathrm{SUT}\ \ \iota_1; \ldots ; o_n \leftarrow \mathrm{SUT}\ \ \iota_n; \mathrm{return}(\mathit{res} = [o_1 \cdots o_n]) \end{aligned}$$

Successive applications of symbolic execution rules allow to reduce the premise of this implication to $C_{\mathrm{SPEC}}\ \iota_1\ \sigma_1 \longrightarrow \ldots \longrightarrow C_{\mathrm{SPEC}}\ \iota_n\ \sigma_n \longrightarrow \mathit{res} = [a_1 \cdots a_n]$ (where the a_i are concrete terms instantiating the bound output variables o_i), i. e., the constrained equation $\mathit{res} = [a_1 \cdots a_n]$. The latter is substituted into the conclusion of the implication. In our previous example, case-splitting over input-variables ι_1, ι_2 and ι_3 yields (among other instances) $\iota_1 = \mathrm{alloc}\ \ t_1\ \mathit{th}_1\ m$, $\iota_2 = \mathrm{release}\ \ t_2\ \mathit{th}_2\ n$ and $\iota_3 = \mathrm{status}\ \ t_3\ \mathit{th}_3$, which allows us to derive automatically the constraint:

$$\begin{gathered} (t_1, \mathit{th}_1) \in \mathrm{dom}(\sigma) \longrightarrow \\ (t_2, \mathit{th}_2) \in \mathrm{dom}(\sigma') \wedge n < \sigma'(t_2, \mathit{th}_2) \longrightarrow \\ (t_3, \mathit{th}_3) \in \mathrm{dom}(\sigma'') \longrightarrow \mathit{res} = [\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}(\sigma''(t_3, \mathit{th}_3))] \end{gathered}$$

where $\sigma' = \sigma((t_1, th_1) := (\sigma(t_1, th_1) + m)))$ and $\sigma'' = \sigma'((t_2, th_2) := (\sigma(t_2, th_2) - n)))$. In general, the constraint $C_{\mathrm{SPEC}_i}\ \iota_i\ \sigma_i$ can be seen as an *symbolic abstract test execution*; instances of it (produced by a constraint solver such as Z3 integrated into Isabelle) will provide concrete input data for the valid test-sequence statement over SUT, which can therefore be compiled to test driver code. In our example here, the witness $t_1 = t_2 = t_3 = 0$, $th_1 = th_2 = th_3 = 5$, $m = 4$ and $n = 2$ satisfies the constraint and would produce (predict) the output sequence $res = [\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}\,2]$ for SUT according to SUT. Thus, a resulting (abstract) test-driver is:

$$\begin{aligned}&\sigma \models o_1 \leftarrow \mathrm{SUT}\ \ \iota_1; \ldots ; o_3 \leftarrow \mathrm{SUT}\ \ \iota_3;\\ &\qquad\qquad\qquad \mathrm{return}([\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}\,2] = [o_1 \cdots o_3])\end{aligned}$$

This schema of a test-driver synthesis can be refined and optimized. First, for iterations of stepping functions an 'mbind' operator can be defined, which is basically a fold over $\mathrm{bind}_{\mathrm{SE}}$. It takes a list of inputs $\iota s = [i_1, \ldots, i_n]$, feeds it subsequently into SPEC and stops when an error occurs. Using mbind, valid test sequences for a stepping-function (be it from the specification SPEC or the SUT) evaluating an input sequence ιs and satisfying a post-condition P can be reformulated to:

$$\sigma \models os \leftarrow \mathrm{mbind}\ \ \iota s\, \mathrm{SPEC}; \mathrm{return}(P\ os)$$

Second, we can now formally define the concept of a test-conformance notion:

$$\begin{aligned}&(\mathrm{SPEC} \sqsubseteq_{\langle \text{Init,CovCrit,conf}\rangle} \mathrm{SUT}) =\\ &\qquad (\forall \sigma_0 \in \mathit{Init}.\ \forall \iota\ s \in \mathit{CovCrit}.\ \forall \mathit{res}.\\ &\qquad \sigma_0 \models os \leftarrow \mathrm{mbind}\ \ \iota s\ \ \mathrm{SPEC};\ \mathrm{return}(\mathit{conf}\ \ \iota s\ os\ res)\\ &\qquad\qquad \longrightarrow \sigma_0 \models (os \leftarrow \mathrm{mbind}\, \iota s\ \mathit{SUT};\ \mathrm{return}(\mathit{conf}\ \ \iota s\ os\ res)))\end{aligned}$$

For example, if we instantiate the conformance predicate conf by:

$$\mathrm{conf}\ \ \iota s\ os\ res = (\mathrm{length}(\iota s) = \mathrm{length}(os) \wedge res = os)$$

we have a precise characterization of inclusion conformance : We constrain the tests to those test sequences where no exception occurs in the symbolic execution of the model. Symbolic execution fixes possible output-sequence (which must be as long as the input sequence since no exception occurs) in possible symbolic runs with possible inputs, which must be exactly observed in the run of the SUT in the resulting abstract test-driver.

Using pre-and postcondition predicates, it is straight-forward to characterize deadlock conformance or IOCO mentioned earlier (recall that our framework assumes synchronous communication between tester and SUT; so this holds only for a IOCO-version without quiescence). Further, we can characterize a set of initial states or express constraints on the set of input-sequences by the *coverage criteria* $\mathit{CovCrit}$, which we will discuss in the sequel.

3.4 Coverage Criteria for Interleaving

In the following, we consider input sequences ιs which were built as interleaving of one or more inputs for different processes; for the sake of simplicity, we will assume that it is always possible to extract from an input event the thread and task id it belongs to. It is possible to represent this interleaving, for example, by the following definition:

```
fun interleave :: 'a list ⇒'a list ⇒'a list set
where interleave [] [] = {[]}
    |interleave A [] = {A}
    |interleave [] B = {B}
    |interleave (a # A) (b # B) =
      (λx. a # x) `interleave A (b # B) ∪
      (λx. b # x) `interleave (a # A) B
```

and by requiring for the input sequence ιs to belong to the set of interleavings of two processes `P1` and `P2`: ι`s ∈interleave P1 P2`. It is well known that the combinatorial explosion of the interleaving space represents fundamental problem of concurrent program verification. Testing, understood as the art of creating finite, well-chosen subspaces for large input-output spaces, offers solutions based on adapted coverage criteria [SLZ07] of these spaces, which refers to particular instances of `CovCrit` in the previous section. A well-defined coverage criterion [ZHM97, FTW04] can reduce a large set of interleavings to a smaller and manageable one. For example, consider the executions of the two threads in MyKeOS: `T = [alloc 3 1 2, release 3 1 1, status 3 1]` and `T' = [alloc 2 5 3, release 3 1 1, status 2 5]`. Since our simplistic MyKeOS has no shared memory, we simulate the effect by allowing `T'` to execute a `release`-action on the local memory of task 3, thread 1 by using its identity. In general, we are interested in all possible values of a shared program variable x at position l after the execution of a process P. To this end we will define two sets of interleavings under two different known criteria.

- **Criterion1: standard interleaving** (SIN) *the interleaving space of actions sequences gets a complete coverage iff all possible interleavings of the actions of P are covered.*
- **Criterion2: state variable interleaving** (SVI) *the interleaving space of actions sequences gets a complete coverage iff all possible states of x at l in P are covered.*

Under SIN we derive 10 possible actions sequences, which is reduced under SVI to 3 sequences (where one leads to a crash; recall our assumption that the memory is initially 0). Unlike to SIN, SVI has provided a smaller interleaving set that cover all possible states. If we consider `var_tab[3,1]` for x when executing `status 3 1`, the possible results may be undefined, O or 1. While SIN has provided a bigger set, that cover all possible 3 states of x with redundant sequences representing the same value. In model-checking, this reduction technique is also known as partial order reduction [Pel93, GW94]. It is a part of a beauty for our test and proof approach, that we can actually formally prove that the test-sets resulting from the test-refinements:

$$\mathrm{SPEC} \sqsubseteq_{\langle Init,SIN,conf\rangle} \mathrm{SUT} \quad \text{and} \quad \mathrm{SPEC} \sqsubseteq_{\langle Init,SVN,conf\rangle} \mathrm{SUT}$$

are equivalent for a given SPEC. The core of such an equivalence proof is, of course, a proof of commutativity of certain step executions, so properties of the form:

$$o \leftarrow \mathrm{SPEC}\ \iota_i; o' \leftarrow \mathrm{SPEC}\ \iota_j; M\ o\ o' = o' \leftarrow \mathrm{SPEC}\ \iota_j; o \leftarrow \mathrm{SPEC}\ \iota_i; M\ o\ o',$$

which are typically resulting from the fact that these executions depend on disjoint parts of the state. In MyKeOS, for example, such a property can be proven automatically for all $\iota_i = \mathrm{release}\ t\ th$ and $\iota_j = \mathrm{release}\ t'\ th'$ with $t \neq t' \vee th \neq th'$; such reordering theorems justify a partial order on inputs to reduce the test-space. We are implicitly applying the testability hypothesis that SUT is input-output deterministic; if a input-output sequence is possible in SPEC, the assumed input-output determinism gives us that repeating the test by an equivalent one will produce the same result.

3.5 Sequence Test Scenarios for Concurrent Programs

HOL-TESTGEN is a test-generation system based on the Isabelle theorem prover. The main goal of this system is to use the features of Isabelle in order to generate a test set. Using the isar command **`test_spec`** from HOL-TESTGEN framework a test scenario can be represented in form of a test specification. A test specification is an **hol!** formula, i. e. a valid test sequence, that describe the test set to be generated. Two possible schemes for a test scenario can be expressed by a test specification: *unit test scheme, sequence test scheme*. In this section we will focus on sequence test scenarios. In sequence test scenarios, a set of input sequences are generated under a given coverage criteria and symbolically executed (see section 3.6 for more details on our symbolic execution process). Actually, a test specification is a **`lemma`** which contains our refinement relation (see section 3.3) as a proof statement. The representation of a refinement relation, for a scenario related to MyKeOS system, using Isabelle/isar language can be:

```
test_spec test_status:
assumes account_defined: (tid,0) ∈dom σ0∧(tid,1) ∈dom σ0
and    CovCrit : S ∈interleave (syscall tid 0 m m')
                               (syscall tid 1 m'' m''')
and    SPEC :
       σ0 ⊨(s ←mbind S SPEC; return (x = s))
shows  σ0 ⊨(s ←mbind S PUT; return (s = x))
```

In the scenario `test_status` the assumption `account_defined` is used to bound the set of threads in the system to 2 members. The assumption `CovCrit` represent the set of possible input sequences related to the concurrent execution between `syscall tid 0 m m'` and `syscall tid 1 m'' m'''`. Moreover, `SPEC` represent the model of the behaviour of the SUT. Finally, the conclusion `σ0 ⊨(s ← mbind S PUT; return (s = x))` is used to link the model with the real system via the free variable `PUT`. Actually, the free variable `PUT` will be linked to the actual code of SUT during the execution of the *test script* (for more details linkage between a model and a SUT see section 3.7).

In fact, the representation of `test_status` by a **lemma** offers a way to use the symbolic computation engine of Isabelle, usually used for proofs, as a simulation environment for the behaviour of the SUT. Basically, the simulation is done via the application of symbolic execution rules, e. g. an instance for Equation 4.1, on the proof statement, which result with a set of subgoals. Each subgoal represent an abstract test case, and each abstract test case is a representation of a set of possible executions in the SUT. Moreover, during the simulation of the behavior of the SUT, and depending on the generated constraints, other kind of rules called behavioral refinement rules can be applied to optimize the process of symbolic execution:

```
 lemma mbindFSave_vs_mbindFStop :
(σ ⊨(os ←(mbind' ιs SPEC);
        return(length ιs = length os ∧P ιs os))) =
 (σ ⊨(os ←(mbind ιs SPEC); return(P ιs os)))
 proof -
 (...)
```

This rule express the fact that we can reduce the behavior of `SPEC` modeled by `mbind` executions to a behavior of `SPEC` modeled by `mbind'` executions under the constraint `length ιs = length os ∧ P ιs os`. The difference between the two type of executions is:

- With `mbind'` the execution of a given input sequence `ιs` by the operational semantic `SPEC` will fail, i. e. returns `None`, if the execution of one input ι in `ιs` fails.
- On the other hand, the execution of a given input sequence `ιs` by `mbind` will never fail, i. e. it returns always `Some(os, σ)`; if the execution of one input ι in `ιs` fail, `mbind` stop the execution of the sequence, purge the failed input and saves the previous outputs resulting from the execution of the previous inputs.

The difference between executions under `mbind'` and `mbind` can be observed more clearly in their **hol!** specifications:

```
fun   mbind :: 'ι list ⇒('ι ⇒('o,'σ) MON_SE) ⇒('o list,'σ) MON_SE
where mbind [] iostep σ= Some([], σ) |
      mbind (a#H) iostep σ=
              (case iostep a σof
                  None ⇒ Some([], σ)  (* Return Some *)
```

```
            | Some (out, σ') ⇒
              (case mbind H iostep σ' of
                None  ⇒ Some([out],σ') (* Return Some *)
               | Some(outs,σ'') ⇒Some(out#outs,σ'')))

fun  mbind' :: 'ι list ⇒('ι ⇒('o,'σ) MON_SE) ⇒('o list,'σ) MON_SE
where mbind' [] iostep σ= Some([], σ) |
     mbind' (a#S) iostep σ=
           (case iostep a σof
              None ⇒ None (* fail-strict *)
            | Some (out, σ') ⇒
              (case mbind' S iostep σ' of
                None  ⇒ None  (* fail-strict *)
               | Some(outs,σ'') ⇒Some(out#outs,σ'')))
```

The simulation, related to the behavior of MyKeOS specified in the scenario `test_status`, using symbolic execution on Isabelle, is represented by the following:

```
 (...)
(**************************************
 ***Resulting proof statement: ctxt1***
 **************************************)
 1. σ0 ⊨( s ←mbind [alloc tid 1 m'', release tid 0 m',
                    release tid 1 m''', status tid 1]
               SYS; unitSE (x = s)) ⟹
    σ0 ⊨( s ←mbind S PUT; unitSE (s = x))

   (****************************
    ***rules applied on: ctxt1***
    ****************************)
   apply(tactic ematch_tac [@{thm status.exec_mbindFStop_E},
                              @{thm release.exec_mbindFStop_E},
                              @{thm alloc.exec_mbindFStop_E},
                              @{thm H1}] 1)

(**************************************
 ***Resulting proof statement: ctxt2***
 **************************************)
1.(tid, 1) ∈dom σ0⟹
  σ0((tid, 1) ↦the (σ0 (tid, 1)) + int m'') ⊨
  (s ←mbind [release tid 0 m', release tid 1 m''', status tid 1]
          SYS ; unitSE (x = alloc_ok # s)) ⟹
 σ0 ⊨( s ←mbind S PUT; unitSE (s = x))

(****************************
 ***rules applied on: ctxt2***
 ****************************)
apply(tactic ematch_tac [@{thm status.exec_mbindFStop_E},
                           @{thm release.exec_mbindFStop_E},
                           @{thm alloc.exec_mbindFStop_E},
                           @{thm H1}] 1)

(**************************************
 ***Resulting proof statement: ctxt3***
 **************************************)
1. (tid, 1) ∈dom σ0⟹
   (tid, 0) ∈dom (σ0((tid, 1) ↦the (σ0 (tid, 1)) + int m'')) ∧
   int m' ≤
   the ((σ0((tid, 1) ↦the (σ0 (tid, 1)) + int m'')) (tid, 0)) ⟹
   σ0((tid, 1) ↦the (σ0 (tid, 1)) + int m'', (tid, 0) ↦
   the ((σ0((tid, 1) ↦the (σ0 (tid, 1)) + int m'')) (tid, 0)) -
   int m') ⊨
   ( s ←mbind [release tid 1 m''', status tid 1]
              SYS ; unitSE (x = alloc_ok # release_ok # s)) ⟹
  σ0 ⊨( s ←mbind S PUT; unitSE (s = x))
(...)
```

A such proof context *refinement process*, is executed until the input sequence of actions is empty, which let us to get, for the case of a specification of a simple operational semantics, what we call *test normal forms*, represented by subgoals. Of course, the proof statement can be connected to solver constraints with HOL-TestGen command `gen_test_data`, which will instantiate the free variables,e. g. σ_0, `tid` in the different subgoals of the proof statement, by a real data that satisfies the derived constraints.

3.6 Optimized Symbolic Execution Rules

Symbolic execution rules, are logical inference rules used to simulate the behavior of a given system (or a program) by showing the effect of the operational semantics of that system (or program) on the *symbolic variables*. Symbolic variables are a typed syntactic names used to represent a given object (i. e. a passive entity), that may have an infinite set of representations (values), in a system. In general, two kind of variables are distinguished, *global variables* and *local variables*. For instance, in our test specification `test_status` the variable σ_0 can be seen as a global variable(i. e. an object which can be modified by all subjects (threads)), while the arguments, e. g. `m''`, of actions e. g. `alloc tid 1 m''`, in the input sequence can be seen as a local variables (i. e. an object which can be modified only by its owner). In order to provide more informations about the symbolic execution rules used during the simulation of the behavior of MyKeOS we would introduce the generic scheme of their **hol!** representation:

```
lemma exec_mbindFStop_E:
assumes A:(σ ⊨(s ←mbind (in_ev # S) efsm; return (P s)))
and    B:E σ⟹
          ((upd σ) ⊨(s←mbind S efsm;return(P(out_ev σ# s)))) ⟹
          Q
shows   Q
  by(insert A, rule B, simp_all del: mbind'_bind)
```

Code 1: A Generic Elimination Rule For Symbolic Execution

If we observe more closely the previous inference rule, we can figure out that the rule is an elimination rule. An elimination rule is an inference rule that eliminate a given constructor from the premises, i. e. in the rule `exec_mbindFStop_E` we had eliminated `in_ev` from the input sequence `(in_ev # S)`. Actually, the scheme of an elimination rule matches with the scheme of our test specifications, i. e. the free variable `Q` in `exec_mbindFStop_E` will match with σ_0 `⊨(s ←mbind S PUT; return (s = x))` in `test_status`, the assumption `A` in `exec_mbindFStop_E` will match with `SPEC` in `test_status`, and the resulting proof context after the application of this elimination inference rule on the test specification `test_status` will be, the instatiation of the assumption `B` in `exec_mbindFStop_E` by the variables of `SPEC`. We call a such proof context transformation process *ematching*, and it can be expressed in Isabelle by the tactic `ematch_tac`. Moreover note, our symbolic execution process based on proof context transformations, has an enormous performance gain effect on symbolic execution engine of Isabelle. Because, the whole calculation process is reduced technically to a *formal* syntactic transformation of the proof context, instead of calculus based on substitution, rewriting, instantiation, introduction, etc.

3.7 Test Drivers for Concurrent C Programs

The generation of the test-driver is a non-trivial exercise since it is essentially two-staged: Firstly, we chose (from the different options the Isabelle code-generator offers) to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. A test-driver for HOL-TESTGEN consists of four components:

- `main.sml` the global controller (a fixed element in the library),
- `harness.sml` a statistic evaluation library (a fixed element in the library),
- `X_script.sml` the test-script that corresponds merely one-to-one to the generated test-data (generated)
- `X_adapter.sml` a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign function interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing "bit-string-representation-behind-the-scene" enabling relatively efficient symbolic computation. Both representations can be compiled "natively" to data types in the SML level. By an appropriate configuration, the code-generator can map "int" of HOL to three different implementations: the SML standard library `Int.int`, the native-C interfaced by `Int32.int`, and the `IntInf.int` from the multi-precision library `gmp` underneath the polyml-compiler. We do a three-step compilation of data-representations Model-to-Model, Model-to-SML, SML-to-C. A basic preparatory step for the initializing the test-environment to enable test-generation is:

```
test_spec test_status2:
assumes system_def : (c0,no) ∈dom σ0
and    store_finite : σ0= map_of T
and    test_purpose : test_purpose [(c0,no),(c0,no')] S
and    sym_exec_spec :
       σ0 ⊨(s ←mbind' S SYS; return (s = x))
shows σ0 ⊨(s ←mbind' S PUT; return (s = x))
apply(rule rev_mp[OF sym_exec_spec])
apply(rule rev_mp[OF system_def])
apply(rule rev_mp[OF test_purpose])
apply(rule_tac x=x in spec[OF allI])
apply(gen_test_cases 3 1 PUT)
apply(auto intro: P1'' P2'')
store_test_thm mykeos_simple
gen_test_data mykeos_simple
generate_test_script mykeos_simple
```

The tool **`store_test_thm`** is a tool from HOL-TestGen framework. This tool provide the ability to users to store a given proof context of the test specification and refer to this proof context by a label (i. e. `mykeos_simple`). The tool **`gen_test_data`** from HOL-TestGen provide the ability to users to instantiate the symbolic variables inside abstract test cases by concrete data. The latter step is done by sending *proof obligations*, i. e. constraints on the variables generated during the symbolic execution, to constraint solvers in order to instantiate them with satisfiable witnesses. The tool **`gen_test_script`** is provided by HOL-TestGen framework. Basically, the tool provide the ability to users to transform the proof context stored using **`store_test_thm`** to a *code equation*; code equations are rewriting rules used as inputs for Isabelle code generators. For instance, the following code equation is resulting from the application of **`gen_test_script`** on the proof context labeled by the name `mykeos_simple`:

```
   mykeos_simple.test_script ≡
[([], lazy ((λa. Some -1) ⊨
    ( s ←mbind [alloc 3 5 (nat 2), status 3 5]
          PUT; unitSE (s = [alloc_ok, status_ok (nat 1)])))),
 ([], lazy ((λa. if a = (2, 3) then Some 8465 else Some 8) ⊨
     (s ←mbind [release 2 3 (nat 8466), status 2 3] PUT;
      unitSE (s = [])))),
 ([], lazy ((λa. Some 8468) ⊨
     ( s ←mbind [release 2 3 (nat 1), status 2 3]
          PUT; unitSE (s = [release_ok, status_ok (nat 8467)])))),
 ([], lazy ((λa. if a = (2, 3) then Some 8465 else Some 8) ⊨
    ( s ←mbind [release 2 3 (nat 8466), status 2 3] PUT;
      unitSE (s = [])))),
 ([], lazy ((λa. Some -1) ⊨
    ( s ←mbind [alloc 2 3 (nat 1), alloc 2 3 (nat 1), status 2 3]
          PUT;
     unitSE (s = [alloc_ok, alloc_ok, status_ok (nat 1)])))),
```

```
(...)]
```

3.7.1 The adapter

In the following, we describe the interface of the SML-program under test, which is in our scenario an *adapter* to the C code under test. This is the heart of the Model-to-SML translation. Actually, during the execution of the test script, the free variable specified inside the test specification under name `PUT` will be replaced by an adapter. In fact, the adapter is a function defined on the HOL-level, and its semantic is based on constant definitions called *stubs*. The stubs are replaced later on by the semantic of the implementation using code serialisation technic offered by the interface of Isabelle code generator to link the Model-level with SML-level, and then we use MLton compiler to link SML-level to C-level. The HOL-level stubs for the program under test are declared as follows:

```
(*The definition of the stubs*)
consts  status_stub :: task_id ⇒int ⇒(int, 'σ)MON_SE
consts  alloc_stub :: task_id ⇒int ⇒int ⇒(unit, 'σ)MON_SE
consts  release_stub:: task_id ⇒int ⇒int ⇒(unit, 'σ)MON_SE
```

This translation step prepares already the data-adaption; the type `nat` is seen as an predicative constraint on integer (which is actually not tested). On the Model-to-Model level, we provide a global step function that distributes to individual interface functions via stubs (mapped via the code generation to SML . . .). This translation also represents uniformly nat by int's.

```
fun  stepAdapter :: (in_c ⇒(out_c, 'σ)MON_SE)
where
    stepAdapter(status tid thid) =
     (x ←status_stub tid thid; return(status_ok (my_nat_conv x)))
   | stepAdapter(alloc tid thid amount) =
     (_ ←alloc_stub thid thid (int amount); return(alloc_ok))
   | stepAdapter(release tid thid amount)=
     (_ ←release_stub tid thid (int amount); return(release_ok))
```

The `stepAdapter` function links the HOL-world and establishes the logical link to HOL stubs which were mapped by the code-generator to adapter functions in SML, which call internally to C-code inside `X_adapter.sml` via a Foreign Function Interface (FFI).

3.7.2 Code generation and Serialisation

In order to generate concrete code from our theories we will use the code generator [haf15] facilities of Isabelle/HOL. It allows to turn a certain class of HOL specifications into corresponding executable code in a target language (i. e. SML). In this section, we will show how we build a setup to generate SML file containing our test script. In the first place, we will generate 2 SML files. The first one containing all datatypes used in our test specification. The second one containing an adapter for the variable representing the system under test called `PUT` in the test specification `test_status2`. Therefore, both files will be used as libraries for the test script and help to increase its readability. Using Isabelle serialiser, we configure the code-generator to identify the `PUT` with the generated SML code implicitly defined by the above `stepAdapter` definition.

```
 (*Code Setup for Datatypes*)

(* Setup for input actions *)
code_printing
```

```
  type_constructor in_c => (SML) Datatypes.in_c
    |constant alloc => (SML) !(Datatypes.Alloc ( _ , _ , _))
    |constant release => (SML) !(Datatypes.Release ( _ , _ , _ ))
    |constant status => (SML) !(Datatypes.Status ( _ , _ ))

(* Setup for the outputs *)
code_printing
  type_constructor out_c => (SML) Datatypes.out'_c
    |constant alloc_ok => (SML) Datatypes.Alloc'_ok
    |constant release_ok => (SML) Datatypes.Release'_ok
    |constant status_ok => (SML) !(Datatypes.Status'_ok ( _ ))
```

Basically, the link between the stubs in HOL world and the SML functions that calls to the C ones is done by asking Isabelle code generator to replace the stubs by functions inside a given SML file. Technically this step is resumed by:

```
(*Serialisation: replacing the HOL stubs by actual semantics
             represented on SML-level*)
code_printing
constant status_stub (SML MyKeOSAdapter.status)

code_printing
constant alloc_stub (SML MyKeOSAdapter.alloc)

code_printing
constant release_stub (SML MyKeOSAdapter.release)
```

By the same technic we ask the code generator to replace the constant `PUT` by the function `stepAdapter`. The latter function, can be generated automatically, as we will see in the last step, and it contains the calls to the stubs which are now SML functions:

```
(*Serialisation: Linking the free variable PUTwith
             the concrete SML-code via stepAdapter*)
code_printing
constant PUT=> (SML) stepAdapter
```

And there we go and generate the `mykeos_simple`:

```
export_code      stepAdapter mykeos_simple.test_script in SML
module_name TestScript file impl/c/mykeos_simple_test_script.sml
```

3.7.3 Building Test Executables

Inside the SML file containing the module `adapter.sml`, we will use again serialisation technic via the compiler MLton. Actually, MLton provides a foreign function interface to C, this interface is used to call the actual semantic of the program under test. MLton compiler provide a command to build the test executable for our generated TestScript in SML language, containing called function from the implemantation in C language.

3.7.4 GDB and Concurrent Code Testing

Actually the generated build from `MLton` compiler will contain tests for threads executed in concurrency. The problem with executing tests on concurrent code is that, we do not know if the generated tests will be applied to their corresponding executions in the SUT, because of the non-deterministic choices of thread's actions done by a system schedular. In order to deal with this kind of problems, we will execute the test executable inside a GDB session that controls the execution of the concurrent code and make it conform to the generated executions from a test scenario.

3.8 Conclusions

In this chapter we have presented our major contribution during this document. The chapter contains theoretical and technical foundations to test C concurrent program. On the theoretical side, we had presented our test generation framework which relies on a monadic test theory implemented in Isabelle/**hol!**. Our framework is equipped with a specification language based on monads that contains important definitions for testing and symbolic execution activities. First, in order to show the expressive power of our specification language, an isomorphism between the automata world and monads world was presented. Second, in order to provide a generic framework to express state exception behavior, two monad operators were introduced $\mathtt{bind}_{\mathrm{SE}}$ and $\mathtt{unit}_{\mathrm{SE}}$. Based on the latter operators, a new concept called valid test sequence was defined. On the one hand, the notion of a valid test sequence is used to express the behavior of a given system. On the other hand, it is executable and can be treated by a family of symbolic executions calculi. A set of generic symbolic execution rules, for the defined operators, were introduced and in order to show how these concepts are used to model and/or to symbolically execute a given system, a running example on a simple OS called MykeOS was presented. Third, we proposed a generic scheme called test specification, expressed technically by a refinement relation, to link a specification with an implementation, then we had showed how it can be instantiated with a family of test conformance relations. Finally, in order to optimize the symbolic execution process for our test specifications, especially for the case of sequence test scenarios, an approach based on the notion of coverage criteria was proposed.

On the technical side, we had showed how Isabelle/**hol!** easily supports and carries our tools, going from symbolic execution on **hol!** down to test script on code level.

Part III

Test-Generation for the PiKeOS IPC

Chapter 4

Testing PikeOS API

4.1 Introduction

In the following, we will outline the PikeOS model (the full-blown model developed as part of the EUROMILS project is about 20 kLOC of Isabelle/HOL code), and demonstrate how this model is embedded into our monadic testing theory.

As a foundation for our symbolic computing techniques, we refine the theory of monads to embed interleaving executions with abort, synchronization, and shared memory to a general but still optimized behavioral test framework.

This framework is instantiated by a model of PikeOS inter-process communication system-calls. Inheriting a micro-architecture going back to the L4 kernel, the system calls of the IPC-API are internally structured by atomic actions; according to a security model, these actions can fail and must produce error-codes. Thus, our tests reveal errors in the enforcement of the security model.

The chapter proceed as follow: In section 4.2 an informal description of PikeOS IPC is presented. The section 4.3 contains the formalisation of PikeOS IPC in Isabelle/**hol!**. In order to catch the bahavior of the latter a new monad combinator is introduced in subsection 4.3.4. Moreover, a generic memory model is presented in section 4.4, it is used to specify some PikeOS IPC atomic actions. Finally, in order to test PikeOS IPC, our testing approach is extended by new notions, in particular these are:

- a new coverage criteria is defined in subsection 4.5.1,
- a new symbolic execution rules are derived in subsection 4.5.3,
- a new methodology for building test drivers is presented in subsection 4.5.6.

4.2 PikeOS IPC Protocol

The IPC mechanism [SYS13a, SYS13b] is the primary means of thread communication in PikeOS. Historically, its efficient implementation in L4 played a major role in the micro-kernel renaissance after the early 1990s. Microkernels had received a bad reputation, as systems built on top were performing poorly, culminating in the billion-dollar failure of the IBM Workplace OS. A combination of shared memory techniques—the MMU is configured such that parts of virtual memory space are actually represented by identical parts of the physical memory—and a radical redesign of the IPC primitives in L4 resulted in an order-of-magnitude decrease in IPC cost. Also in PikeOS, IPC message transfer can operate between threads which may belong to different tasks. However, the kernel controls the scope of IPC by determining, in each instance, whether the two threads are permitted to communicate with each other. IPC transfer is based on shared memory, which requires an agreement between the sender and receiver of an IPC message. If either the sending or the receiving thread is not ready for message transfer, then the other partner must wait. Both threads can specify a timeout for the maximum time they are prepared to wait and have appropriate access-control rights. Our IPC model includes eight *atomic actions*, corresponding more-or-less to code sections in the API system calls `p4_ipc_buf_send()` and `p4_ipc_buf_recv()` protected by a global system lock. If errors in these actions occur—for example for lacking access-rights—the system call is *aborted*, which means that all atomic actions belonging to the running system call as well as the call of the communication partner were skipped and execution after the system calls on both sides is continuing as normal. It is the responsibility of the application to act appropriately on error-codes reported as a result of a call. In our sequence test scenarios, and using our symbolic execution process running on the top of HOL-TESTGEN, we show how we generate tests from our formal model of the IPC mechanism, we build a *test driver* and show how we can run the generated tests against the PikeOS IPC implementation defined in C-level.

4.3 PikeOS Model

We model the protocol as composition of several operational semantics; this composition is represented by monad-transformers adding, for example, to the basic transition semantics the semantics for abort behavior.

4.3.1 State

In our model, the system state is an abstraction of the VMIT (which is immutable) and mutable task specific resources. It is presented by the (polymorphic) record type:

```
record
('memory,'thread_id,'thread,'sp_th_th,'sp_th_res,'errors)kstate=
  resource              :: 'memory
  current_thread        :: 'thread_id
  thread_list           :: 'thread list
  communication_rights :: 'sp_th_th
  access_rights         :: 'sp_th_res
  error_codes           :: 'errors
  errors_tab            :: 'thread_id ⇀'errors
```

Note that the syntax is very close to functional programming languages such as SML or OCaml or F#. The parameterization is motivated by the need of having different abstraction layers throughout the entire theory; thus, for example, the *resource* field will be instantiated at different places by abstract shared memory, physical memory, physical memory and devices, etc.—from the viewpoint of an operating system, devices are just another implementation of memory. In the entire theory, these different instantiations of `kstate` were linked by abstraction relations establishing formal refinements. Similarly, the field *current_thread* will be instantiated by the model of the *ID* of the thread in the execution context and more refined versions thereof. *thread_list* represents information on threads and there executions. The *communication_rights* field represent the communication policy defined between the active entities (i. e., threads and tasks). The field *access_rights* represent the access policy defined between active entities and passive entities (i. e., system resources).

For the purpose of test-case generation, we favor instances of `kstate` which are as abstract as possible and for which we derived suitable rules for fast symbolic execution.

4.3.2 Actions

As mentioned earlier, the execution of the system call can be interrupted or *aborted* at the border-line of code-segments protected by a lock. To avoid the complex representation of interruption points, we model the effect of these lock-protected code-segments as atomic actions. Thus, we will split any system call into a sequence of atomic actions (the problem of addressing these code-segments and influencing their execution order in a test is addressed in the next section). Atomic actions are specified by datatype as follows:

```
datatype ('ipc_stage,'ipc_dir)action__ipc = IPC 'ipc_stage 'ipc_dir
datatype p4_stage__ipc = PREP | WAIT | BUF | MAP | DONE
```

```
datatype ('thread_id ,'adresses) p4_direct__ipc =
                 SEND 'thread_id 'thread_id 'adresses
               | RECV 'thread_id 'thread_id 'adresses

type_synonym ACTION__ipc =
(p4_stage__ipc,(nat×nat×nat,nat list)p4_direct__ipc)action__ipc
```

Where `ACTION__ipc` is type abbreviation for IPC actions instantiated by `p4_direct__ipc`. The type `ACTION__ipc` models exactly the input events of our monadic testing framework. Thread IDs are triples of natural numbers that specify the resource partition the thread belongs to as well as the task and the individual id. The stepping function as a whole is too complex to be presented here; we refrain on the presentation of a portion of an auxilliary function of it that models just the $\mathtt{PREP\_}_{\mathtt{SE}}\mathtt{ND}$ stage of the IPC protocol; it must check if the task and thread id of the communication partner is allowed in the VMIT, if the memory is shared to this partner, if the sending thread has in fact writing permission to the shared memory, etc. The VMIT is part of the resource, so the memory configuration, and auxiliary functions like `is_part_mem_th` allow for extracting the relevant information from it. The semantic of the different stages is described using a total functions:

```
definition
 PREPSEND ::ACTION_ipc state_id⇒ ACTION_ipc ⇒ACTION_ipc state_id
where PREPSEND σact =
   (case act of (IPC PREP (SEND caller partner msg)) ⇒
     ...
     if is_part_mem_th (get_thread_by_id'' partner σ) (resource σ)
     then
       if IPC_params_c1 (get_thread_by_id'' partner σ)
       then ...)
```

Where $\mathtt{PREP}_{\mathtt{SE}}\mathtt{ND}$, $\mathtt{WAIT}_{\mathtt{SE}}\mathtt{ND}$, $\mathtt{BUF}_{\mathtt{SE}}\mathtt{ND}$, and $\mathtt{DONE}_{\mathtt{SE}}\mathtt{ND}$ define an operational semantic for the atomic actions of the PikeOS IPC protocol.

4.3.3 Traces, executions and input sequences

During our experiments, we will generate *input sequences* rather than traces. An input sequence is a list of a datatype capturing atomic action input syntactically. An *execution* is the application of a transition function over a given input sequence. Using mbind, the execution over a given input sequence is can be immediately constructed.

```
definition execution = (λis ioprog σ. mbind is ioprog σ)
```

4.3.4 Aborted executions

Our model support the notion of abort. An abort is an action done by the system to stop the execution of a given system call. A system call can be aborted for different reasons:

- timeouts: a system call can not finish its execution because a timeout happened. For instance, a caller tried to access to a given resource and run out of the specified waiting time without success, i. e. the resource was not available at that moment. Or the caller run out of the specified waiting time when he was about to wait for a given input from another call.
- other error codes: a system call can not finish its execution because of a returned error code during its execution, i. e. on of the call conditions was not satisfied, e. g. wrong communication partner. Thus, the system stops the execution of the call.

In all cases, when an abort happens to a given PikeOS call, the remaining atomic actions of the call are canceled (not executed). For the case of the IPC protocol both calls, the one coming from the caller and

if *executing DONE stage* **then**
  if *an error happened* **then**
    Update error table by removing the error flag of the current thread and don't execute the DONE action and return the error code.
  else
    Execute the DONE action.
  end
else
  if *Executing a different IPC stage from DONE* **then**
    if *an error happened* **then**
      Update the error table by putting an error flag on both threads in the IPC communication, the caller and his partner, and purge the executed action.
    else
      Execute the action.
    end
  end
end

Algorithm 1: A pseudo code for the Abort operator

the one coming from his communication partner, are canceled. To express the behavior of the abort in our model we will add to our specification language a new monad combinator. The behavior expressed by this combinator is abstracted by the pseudo code in algorithm 1.

In the case of an aborted system call, the semantic of our combinator express the same behavior as stutter steps in automata models, i. e. we stay in the same state, only the *error table* will change. The error table is modeled by the field `errors_tab` of the record `(...)kstate` representing the system state, the field is instantiated by a partial function with type error_tab:: thid ⇀ error, and it is used to save (i. e. marks by a flag) the threads in *error state*, i. e. threads who cause errors during the execution of their system call. Every thread inside the error table is considered as a thread in an error state, when a given system call executed by a given thread is aborted, i. e. the executed action provide an output error code, we update the thread table by adding the thread and its error. Before executing any atomic action (stage) we will check the error table, if a given thread executing an action different from DONE is in the domain of the function that specify the error table, then we purge his executed action (we do nothing to the state of the system) else we will execute the action. During every DONE action execution, if the thread is in the error table then, we remove it from the domain of the function that specify the error table else, we execute the DONE action.

The **hol!** representation of the new monad operator is `abort__lift`, the latter express the explained behavior and will be wrapped around our transition function for PikeOS IPC protocol. The wrapper transforms the behavior of the basic transition function related to IPC protocol presented in subsection 4.3.5, to a the behavior abstracted by algorithm 1.

```
fun abort__lift ::
    (ACTION__ipc ⇒(errors, (ACTION__ipc,'a) state__id_scheme)Mon_SE) ⇒
     (ACTION__ipc ⇒(errors, (ACTION__ipc,'a) state__id_scheme)Mon_SE)
where abort__lift ioprog a σ=
 (case a of
   (IPC DONE (SEND caller partner msg)) ⇒
     if caller ∈dom (act_info (th_flag σ))
     then unit_SE (fst (the((act_info (th_flag σ)) caller))
                (*shoud be: my error*)
             (σ(|th_flag := (th_flag σ)
                         (|act_info := ((act_info (th_flag σ))
                                        (caller := None))|) |))
     else unit_SE (NO_ERRORS) (σ) (*execute done*)
```

```
 (...)
| (IPC _ (SEND caller partner msg)) ⇒
  if caller ∈dom (act_info (th_flag σ))
  then unit_SE(get_caller_error caller σ(*should be: my error*)) σ
      (* purge and add error flag*)
  else (case ioprog a σof
        None ⇒None (*never happens in our exec fun*)
      | Some(NO_ERRORS, σ') ⇒unit_SE(NO_ERRORS) (σ')
      | Some(out', σ') ⇒unit_SE(out')
          (set_caller_partner_error caller partner σσ' out' a))
  (*both caller and partner were 'informed' to be in error-state.*)
( ...))
```

In subsection 4.5.3 we will derive generic symbolic execution rules related to the combination of a given monad that specify an input output program `ioprog` with the abort operator, then we refine these rules for the specific case when the operational semantic is related to PikeOS IPC.

4.3.5 IPC Execution Function

To combine the different semantics of IPC atomic actions we can use two ways of modeling:

- An isabelle function *fun*: Express the semantic with explicit case splitting on actions type in a single function. Useful for the automation of the process of symbolic execution. Used for experimental purposes.
- The composition operator *Fun.comp*: equipped with the syntax $f\ o\ g$. It helps to express the semantic by a set of compositions between different Isabelle constant definitions, these definitions are wrapped around a monad function that express a transition function. Useful to express proofs on our coverage criteria, proofs on refinement and abstractions, but do not help for the automation process of symbolic execution.

Using the compositional way of modeling, the execution semantic for IPC protocol is represented on Isabelle as following:

```
definition IPC_protocol =
 PREP_SEND_lift o PREP_RECV_lift o
 WAIT_SEND_lift o WAIT_RECV_lift o
 BUF_SEND_lift o BUF_RECV_lift o
 DONE_SEND_lift o DONE_RECV_lift
```

The second way of modeling the transition function is the following total function:

```
fun exec_action
::ACTION_ipc state_id⇒ ACTION_ipc ⇒ACTION_ipc state_id
where
  PREP_SEND_run:
  exec_action σ(IPC PREP (SEND caller partner msg)) =
   PREP_SEND σ(IPC PREP (SEND caller partner msg))|
  (...)
```

The function `exec_action` is adapted to the monads using the following definition:

```
definition exec_action_Mon
where   exec_action_Mon =
  (λact σ. Some (error_codes(exec_action σact),
   exec_action σact))
```

The latter function represent the basic operational semantic for PikeOS IPC and it will be combined with the semantic of the abort operator presented in subsection 4.3.4. For instance we wrap around the function `exec_action_Mon` the operator `abort__lift` in order to get, `abort__lift (exec_action_Mon act σ)`. Also we can compose `abort__lift` with `IPC_protocol` to get `abort__lift o IPC_protocol` which is similar to `abort__lift (exec_action_Mon act σ)`. In subsection 4.5.3 we will derive specific symbolic execution related to `abort__lift (exec_action_Mon act σ)`.

4.3.6 System calls

As mentioned earlier, PikeOS system calls are seen as sequence of atomic actions that respect a given ordering. Actually, each system call can perform a set of *operations*. On system-level, the execution of some operations can be ignored by specifying the corresponding parameters in the call by null. PikeOS IPC API provides seven different calls, the most general one is the call *P4_ ipc()*. Using *P4_ ipc()*, five operations can be performed:

1. Send a copied message,
2. Receive a copied message,
3. Receive an event (not modeled),
4. Send a mapped message, and
5. Receive a mapped message.

The corresponding Isabelle model for the call is:

```
datatype ('thread_id, 'msg) P4_IPC_call =
  P4_IPC_call  'thread_id 'thread_id 'msg
| P4_IPC_BUF_call 'thread_id 'thread_id 'msg
| P4_IPC_MAP_call 'thread_id 'thread_id 'msg
  (...)
```

4.4 A Generic Shared Memory Model

Shared memory is the key for the L4-like IPC implementations: while the MMU is usually configured to provide a separation of memory spaces for different tasks (a separation that does not exist on the level of physical memory with its physical memory pages, page tables, ...), there is an important exception: physical pages may be attributed to two different tasks allowing to transfer memory content directly from one task to another.

In order to model a such memory implementation, we will use an abstract memory model with a sharing relation between addresses. The sharing relation is used to model the IPC map operation, which establishes that memory spaces of different tasks were actually shared, such that writes in one memory space were directly accessed in the other. Under the sharing relation, our memory operations respect two properties:

1. Read memory on shared addresses returns the same value.

2. All shared addresses has the same value after writing.

In formal methods, the latter two properties are called *invariants*. An invariant is a property preserved by a class of mathematical object when a certain updates (changes) are performed on that class. The notion of invariants will be used in our model of shared memory. In our memory model, the two listed invariants will be preserved on a tuple type consisting of a pair of two elements: a partial function and an equivalence relation. While the partial function will specify the memory, i. e. the function represent a mapping from its domain consisting of a set of adresses to its range consisting of their corresponding data, the equivalence relation determines the different equivalent classes for addresses. Actually, these equivalent classes are resulting from the different map operations performed by processes of a system.

In order to implement this model on top of Isabelle/**hol!** we will use the specification construct **typedef**, and this for two reasons:

1. It offers a way to define an abstract type that can be equipped with invariants.
2. A defined operation on that abstract type, can be easily used for code generation and this, only by providing a soundness proof which express that the operation preserve the invariants on the defined type.

The **hol!** specification for our memory abstract type is done by the following

```
typedef ('α, 'β) memory =
{(σ::'α ⇀'β, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
 proof
   show (Map.empty, (op =)) ∈?memory
     by (auto simp: identity_equivp)
 qed
```

This type definition defines an isomorphism between the set on the right hand side that contains pairs of the type `('a ⇀'b)×('a ⇒'a ⇒bool)` and the set defined by the new type `('α, 'β)memory`; the first element of a pair is a partial function representing a mapping from adresses to data, the second element is an equivalence relation. The type `('α, 'β)memory` is introduced by two fresh constant symbols, the function `Abs_memory` for abstracting the pairs, and `Rep_memory` the concretization function that refer to the pairs. The application of a given operation `op` on the pairs is isomorphically the same as the application of `Abs_op` on the type `('α, 'β)memory` with the only difference: the use of the type `('α, 'β)memory` for the definition of the different operations assure that the latter talk about representatives which preserve the invariant. Because the set of tuples of type `('a ⇀'b)×('a ⇒'a ⇒bool)` is infinite and may contain tuples that does not preserve the desired invariant, thus the direct use of `op` is not consistent. That is why we will always define a function on representatives in the following, and this in order to get the desired effects on the pairs. Afterwards we implement and use its corresponding abstraction that refers implecitly to representatives preserving the invariant.

Implecitely, five theorems are generated by Isabelle for the functions `Abs_memory` and `Rep_memory`, where `Rep_memory_inverse, ...` are names for the generated theorems:

```
 Rep_memory_inverse:
 Abs_memory (Rep_memory x) = x

Abs_memory_inverse:
?y ∈{(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σy)} ⟹
 Rep_memory (Abs_memory ?y) = ?y

 Rep_memory_inject:
(Rep_memory ?x = Rep_memory ?y) = (?x = ?y)

 Rep_memory:
 Rep_memory ?x ∈{(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
```

These theorems will help in the proof of the different lemmas used for reasoning on a defined constant based on the type `('α, 'β)memory`. Using this new defined abstract type we will now specify three main memory operations, which are *write* denoted by `_ :=$ _` *read* by `_ $ _` and *map* by `_ (_ ⋈_)`. The **hol!** specification of these memory operations is represented for instance, for the case of the map operation by:

```
fun transfer_rep ::('a ⇀'b) ×('a ⇒'a ⇒bool) ⇒'a ⇒'a ⇒
                  ('a ⇀'b) ×('a ⇒'a ⇒bool)
 where transfer_rep (m, r) src dst =
 (m o (id (dst := src)),
 (λ x y . r ((id (dst := src)) x) ((id (dst := src)) y)))

lift_definition
add_e :: ('a,'b)memory ⇒'a ⇒'a ⇒('a, 'b)memory (_ '(_ ⋈_'))
  is transfer_rep using transfer_rep_sound
  by simp
```

The function `transfer_rep` is an update function on representatives, i. e. on the pairs of type `('a ⇀'b)×('a ⇒'a ⇒bool)`, and the function `add_e` is its abstraction defined on the type `('α, 'β)memory`.

Basically, the function `transfer_rep` takes a memory represented by the pair `('a ⇀'b)×('a ⇒'a ⇒bool)`, a source address `src`, a destination adress `dst` and update the pair, in order to express the effect of a memory map on that pair, as follow:

1. the first element of the pair, which is a partial function representing a mapping from adresses to data, is updated by assigning the data of the source address to the destination adress
2. the second element of the pair, which is an equivalent relation between adresses, is updated by adding the destination adress to the same equivalent class of the source adress, and at the same time the relation between the destination and its old equivalent class is destroyed. This definition was validated by PikOS kernel engineers

Actually, we will not directly use `transfer_rep`, the function will be abstracted by `add_e`, and this is advantageous for the following reasons; on one hand we make sure that, on model level, `add_e` will always return pairs that preserve the invariant. On the other hand, the specification constraint `lift_definition` provide automatically a code generation setup for memory operations based on the type `('α, 'β)memory`, i. e. the generated implementation will contain implicitly only pairs that preserve the invariant.

If we look closely, we can observe that a little proof was mandatory to get the definition of `add_e`. In fact, in order to preserve the consistency of its global context, Isabelle forces a such proof. This proof is used to make sure that the invariant defined in the abstract type is preserved by the definition of `add_e`. In other words, we have to make sure that the added definition is sound and its use does not break the invariant, a such soundess proof was provided by the following lemma:

```
lemma transfer_rep_sound:
assumes σ∈{(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
shows transfer_rep σsrc dst ∈
       {(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
proof -
  obtain mem and R
    where P: (mem, R) = σand
          E: equivp R and
          M: ∀x y . R x y ⟶mem x = mem y
      using assms equivpE by auto
  obtain mem' and R'
    where P': (mem', R') = transfer_rep σsrc dst
      by (metis surj_pair)
  have D1: mem' = (mem o (id (dst := src)))
   and D2: R' = (λ x y . R ((id (dst := src)) x)
              ((id (dst := src)) y))
    using P P' by auto
  have equivp R'
```

```
    using E unfolding D2 equivp_def by metis
   moreover have ∀y z . R' y z ⟶mem' y = mem' z
    using M unfolding D1 D2 by auto
   ultimately show ?thesis
    using P' by auto
  qed
```

In order to simplify the use of these abstract memory operations by constraint solvers, and also in order to simplify the proof of symbolic execution rules related to these operations, lemmas expressing the key properties of our shared memory model were introduced, we will present only the most important lemmas:

```
definition sharing ::
        'a ⇒('a, 'b)memory ⇒'a ⇒bool ((_ shares()_/ _)
where (x shares (σ) y) ≡(snd(Rep_memory σ) x y)

definition Domain :: ('α, 'β)memory ⇒'α set
where   Domain σ= dom (fst (Rep_memory σ))

lemma shares_result:
  assumes 1: (x shares (σ) y)
  shows     σ $ x = σ$ y
  using assms lookup_def shares_result
  by metis
```

Sharing is modulo equivalence relation:

```
lemma sharing_refl [simp]: (x shares (σ) x)
 using insert Rep_memory[of σ]
 by (auto simp: sharing_def elim: equivp_reflp)

lemma sharing_sym [sym]:
  assumes x shares (σ) y
  shows y shares (σ) x
  using assms Rep_memory[of σ]
  by (auto simp: sharing_def elim: equivp_symp)

lemma sharing_trans [trans]:
   assumes x shares (σ) y
   and    y shares (σ) z
   shows x shares (σ) z
   using assms insert Rep_memory[of σ]
   by(auto simp: sharing_def elim: equivp_transp)
```

Sharing relates to memory write as follows:

```
lemma sharing_upd: x shares (σ(a :=$b)) y = x shares (σ) y (*$*)
  using insert Rep_memory[of σ]
  by(auto simp: sharing_def update_def
     Abs_memory_inverse[OF update_sound])

lemma update_idem :
   assumes 1: x shares (σ) y
   and    2: x ∈Domain σ
   and    3: σ $ x = z
   shows σ(x:=$ z) = σ
proof -
  have * : y ∈Domain σ
   by(simp add: shares_dom[OF 1, symmetric] 2)
  have σ(x :=$ (σ $ y)) = σ
```

```
    using 1 2 * by (simp add: update_triv)
  also have (σ $ y) = σ$ x
    by (simp only: lookup_def shares_result [OF 1])
  also note 3
  finally show ?thesis .
qed

lemma update_share:
   assumes z shares (σ) x
   shows σ(x :=$ a) $ z = a
   using assms
   by (simp only: update_apply if_true)

lemma update_other:
   assumes ¬(z shares (σ) x)
   shows σ(x :=$ a) $ z = σ$ z (*$*)
   using assms
   by (simp only: update_apply if_False)

theorem update_cancel:
  assumes x shares (σ) x'
  shows σ(x :=$ y)(x' :=$ z) = (σ(x' :=$ z)) (*$*)
proof -
   (...)

theorem update_commute:
   assumes 1:¬ (x shares (σ) x')
   shows   (σ(x :=$ y))(x' :=$ z) =
            (σ(x':=$ z)(x :=$ y))
proof -
    (...)
```

Sharing relates to domain as follows:

```
lemma Domain_mono:
  assumes 1: x ∈Domain σ
  and    2: (x shares (σ) y)
  shows     y ∈Domain σ
  using 1 2 Rep_memory[of σ]
  by (auto simp add: sharing_def Domain_def )
```

```
lemma update_triv:
  assumes 1: x shares (σ) y
    and  2: y ∈Domain σ
  shows    σ (x :=$ (σ $ y)) = σ
proof -
  {
    fix z
    assume zx: z shares (σ) x
    then have zy: z shares (σ) y
      using 1 by (rule sharing_trans)
    have F: y ∈Domain σ⟹x shares (σ) y ⟹
             Some (the (fst (Rep_memory σ) x)) = fst (Rep_memory σ) y
      by(auto simp: Domain_def dest: shares_result)
    have Some (the (fst (Rep_memory σ) y)) = fst (Rep_memory σ) z
      using zx and shares_result [OF zy] shares_result [OF zx]
      using F [OF 2 1]
      by simp
  } note 3 = this
  show ?thesis
    unfolding update'' lookup_def fun_upd_equivp_def
    by (simp add: 3 Rep_memory_inverse if_cong)
qed
```

Similarly, we prove other rules for memory map and memory read which represent a memory theory modulo sharing. The defined memory operations are used actually to implement the MAP and BUF actions of PikeOS IPC. For more details on our **hol!** model for shared memory see section 4.9.

4.5 Testing PikeOS IPC

4.5.1 Coverage Criteria for IPC

An IPC call defines a *communication* relation between two threads. In PikeOS, IPC communications can be symmetric, transitive but can not be reflexive (a thread can not send or receive an IPC message for himself). The transitivity or intransitivity of IPC communications depends mainly on the defined communication rights table and access rights table. In this section, we will define input sequences for ipc calls. The defined input sequences express IPC communications between threads. Other definitions, which are almost the same as the ones used for input sequences, will be used to derive the possible communications between threads after the execution of an IPC call. The IPC input sequences will be used in scenarios for testing information flow policy via IPC error codes, and also scenarios on access control policy implemented via the two tables cited before.

The definition of an input sequence of type IPC communication is based on a new coverage criterion. The criterion is based on the functional model of PikeOS IPC (see section 4.2), and also on our technique to reduce the set of interleaving if two actions can commute (see section 3.4).

- **Criterion3: IPC communications** (IPC_{comm}) *the interleaving space of input sequences gets a complete coverage iff all IPC communications of a given SUT are covered.*

IPC communications are input sequences derived under $\mathbf{IPC}_{comm}$. They have the form:

```
[IPC PREP (SEND th_id th_id' msg),
 IPC PREP (RECV th_id' th_id msg),
 IPC WAIT (SEND th_id th_id' msg),
 IPC WAIT (RECV th_id' th_id msg),
 IPC BUF (RECV th_id' th_id msg),
 IPC DONE (RECV th_id' th_id msg),
 IPC DONE (SEND th_id th_id' msg)]
```

4.5.2 Test Case Generation Process

In our model, a test case generation process is applied on the test scenario to generate concrete tests. To apply a such process we will implicitly benefit from implemented tools, proofs and tactics of Isabelle. As explained in section 3.5, a test scenario is specified by a test specification which is actually a lemma. The goal is not to provide a proof for the lemma, the goal is just to normalize this HOL formula until we get a *test normal form (TNF)* [BW13], and then we generate concrete test from the TNF. In our approach, the process of test generation is composed of:

The Symbolic State.

In our model a symbolic state is the Isabelle lemma proof statement, i. e. a proof context.

The Symbolic Execution Process.

Our symbolic execution process can be seen as an exploration of the proof tree resulting from the application of symbolic execution rules to a given test specification. Symbolic execution rules are Isabelle proved lemmas. Those rules are inference rules derived from a given operational semantics. They are used to simulate the execution of a given transition function, which specify the behavior of the system under test. The application of a such rules allows for going from a symbolic state, i. e. a proof statement, to another symbolic state. In sequence test scenarios this step is applied until the input sequence is empty.

The Normalization Process.

Normalization rules are Isabelle proved lemmas. Two main goal are distinguished for the normalization process

1. First, normalization rules are used to simplify the abstract test cases generated after the application of symbolic execution rules, in order to get a proof statement containing a set of TNFs that can be easily treated by constraint-solvers.

2. Second, normalization rules are used to eliminate as much as possible *unfeasible executions* in the proof tree, i. e. proof statements that lead to true, (see subsection 4.5.4 for further explanation).

In our model, the outputs from this step are *abstract test cases*. Abstract test cases are a normalized proof goals generated from symbolic execution process. Proof goals are normalized, i. e., reduced to clauses over linear arithmetic, list, and map-theories in a format that can be treated by the subsequent constraint solver. Outputs from the normalization process are also called TNFs. In our approach, the step of normalization takes most of the generation time.

The Test Theorem.

After the normalization process we generate the test theorem. Actually HOL-TESTGEN provides a tactic for the generation of a test theorem of the form:

$$\frac{C_1(a_1) \;\Rightarrow\; P(a_1,\; PUT\; a_1) \;\ldots\; C_n(a_n) \;\Rightarrow\; P(a_n,\; PUT\; a_n)\; THYP(H_1 \wedge \cdots \wedge H_n)}{TS}$$

The test theorem decompose each abstract test case in the local proof context generated from a test specification to 3 parts:

1. **Proof Obligations**: are the premises of a given abstract test case. e. g. in the previous formula a proof obligation is $C_i(a_i)$.
2. **Testing Hypotheses**: In addition to testing hypotheses expressed as assumptions of a given test specification, HOL-TESTGEN offer a way to introduce testing hypotheses, e. g. regularity and uniformity hypotheses, to a test specification. In the previous formula testing hypothesis are H_i. $THYP$ is a constant definition used as markup for the testing hypothesis during the generation of the test theorem.
3. **Abstract Test Cases**: also called TNFs, they are represented in the test theorem by $C_i(a_1) \;\Rightarrow\; P(a_i,\; PUT\; a_i)$, where P is the oracle, and a_i is a concrete instance that must satisfy the constraint C_i.

A test theorem state that a concrete test case passes if the application of a program under test PUT on a concrete instance a_i satisfies the oracle P.

Test Data Generation.

The proof obligations of each abstract test case are sent to constraint-solvers such as Z3[dMB08], in order to construct a concrete (“ground”) data for the variables. These instantiated abstract test cases represent actually execution paths in a program under test; they are used as test cases for this system.

4.5.3 Symbolic Execution Rules

Symbolic execution rules are inference rules for the elimination of the inputs in the test specification. In our model we distinguish two categories of these rules:

1. The generic ones: they are related to operators of our specification language, i. e. the proposed monad operators in our theory like:$\texttt{bind}_{\texttt{SE}}$ `abort__lift`, etc. These rules are fixed element in the theory, and they talk in general about any state exception monad `ioprog`, of type $\texttt{('}\iota \Rightarrow \texttt{('o,'}\sigma\texttt{)MON}_{\texttt{SE}}\texttt{)}$, that represent any transition function with state exception.
2. The specific ones: they are a refinement of the generic ones. These rules talk about an intantiation of `ioprog` by a given operational sematic.

The Generic Rules.

Generic rules are elimination rules derived for the generic operational semantics expressed by the different monads operator introduced by our specification language. This kind of rules has the following form:

$$\frac{(\sigma \models outs \leftarrow ioprog\;(\iota\#\iota s); P\; s) \qquad \begin{array}{c}\left[\begin{array}{l} ioprog\;\iota\;\sigma \;=\; Some\;(o_\iota, \sigma') \\ (\sigma' \models outs \leftarrow ioprog\;\iota s; P\;(o_\iota\#s)) \end{array}\right]_{o_\iota,\sigma'} \\ \vdots \\ Q \end{array}}{Q} \tag{4.1}$$

Where σ is a symbolic variable that denote the state of a given system, $outs$ is a sequence of outputs resulting from the execution of the transition function $ioprog$, $\iota\#\iota s$ is a list of inputs and P is a post condition on the sequence of outputs. A concrete example of generic symbolic execution rules is the rule 1 presented in section 3.2. In order to catch the behavior of PikeOS, our specification language was extended by a new state exception monad operator called `abort__lift`, an example of a generic symbolic execution rule related to this operator is:

```
lemma abort_wait_send_mbindFSave_E:
  assumes valid_exec:
   (σ ⊨(outs ←(mbind ((IPC WAIT (SEND caller partner msg))#S)
                     (abort__lift ioprog));P outs))
  and in_err_state:
   caller ∈dom (act_info (th_flag σ)) ⟹
    (σ ⊨(outs ←(mbind S (abort__lift ioprog));
         P (get_caller_error caller σ# outs))) ⟹Q
         (...)
  and not_in_err_state_Some3:
   ⋀σ' error_IPC.
    (caller ∉dom (act_info (th_flag σ))) ⟹
    ioprog (IPC WAIT (SEND caller partner msg)) σ= Some(ERROR_IPC error_IPC, σ') ⟹
    ((set_error_ipc_waitr caller partner σσ' error_IPC msg) ⊨
    (outs ←(mbind S(abort__lift ioprog));
    P ( ERROR_IPC error_IPC# outs))) ⟹Q
  and not_in_err_state_None:
   (caller ∉dom (act_info (th_flag σ))) ⟹
    ioprog (IPC WAIT (SEND caller partner msg)) σ= None ⟹
    (σ ⊨(P [])) ⟹Q
  shows Q
proof (cases caller ∈dom (act_info (th_flag σ)))
(...)
```

In order to motivate the use of elimination rules for symbolic execution, we will explain the process of their application on a given proof context. The use of the rule `abort_wait_send_mbindFSave_E` on a given test specification `Test_Scenario` is conditioned by the existence of a given assumption in `Test_Scenario` that have the same scheme of the assmuption `valid_exec` and the existence of a conclusion. For the case of a valid test specification the conclusion will have the same scheme of `valid_exec`, the only difference will be the FREE variable that represent the model? e. g. `ioprog`. Actually, it is replaced by a variable, e. g. `SUT`, that represent the system under test. Once these conditions are brought together for a given test specification `Test_Scenario` the application of the rule will be performed using the tactic `ematch_tac` (see section 3.5 for further explanations). The process of the application of rules , such as `abort_wait_send_mbindFSave_E`, on a valid representation of `Test_Scenario` is:

1. Each time the input action `(IPC WAIT (SEND caller partner msg))` is in the header of a sequence of inputs ιs specified in a test specification `Test_Scenario`, a matching is established between the assumption `valid_exec` and the assumption that specify a model of a tested system in `Test_Scenario`, e. g. an assumption that specify a model for a test specification `Test_Scenario` related to PikeOS can be `σ ⊨(outs ←mbind is(abort__lift exec_action_Mon);ret = x)`. The same thing will happen for the conclusion of the rule, which is by the way a free variable `Q` that can be instantiated by any boolean formula, of course for the case of a valid test specification the scheme of the conclusion specify a valid test execution for a system under test, e. g. `σ ⊨(outs ←mbind is SUT;return(outs = x)`.
2. After the establishment of the ematching, the proof statement provided by `Test_Scenario` is

transformed to a new proof statement. The latter will contain a set of proof goals, each goal has is a "not matched" assumption specified in the rule, e. g. if `Test_Scenario` contain only an assumption in the form of `valid_exec` then the new proof context, after the application of the rule with ematching tactic, will contain the other assumptions of the rule like `in_err_state` and `not_in_err_state_Some3`, etc.

3. We repeat the same process with different rules related to different input actions until we got an empty input sequence. The resulting proof statement will receive a normalization process in order to get abstract test cases for `Test_Scenario`.

A such process, actually based on ematching technic, has an enormous performance gain effect on symbolic execution engine of Isabelle. Because, the whole calculation process is reduced technically to a formal syntactic transformation of the proof context, instead of calculus based on substitution, rewriting, instantiation, introduction, etc. From another side, the execution of a such process on a sequence of inputs specified in a given test specification can be easily automated by an algorithm. The algorithm basically is represented by an Isabelle tactic, the latter takes the different symbolic execution rules related to the different actions of the specified system and execute the rules on the proof context until no rules can be applied. For instance, a tactic for symbolic execution related to the actions of PikeOS IPC is:

```
val abort_ipc_mbind_TestGen_PureE21_ematch =
 (ALLGOALS o TestGen.REPEAT') (CHANGED o TRY o FIRST'
 [ematch_tac
  [@{thm abort_prep_send_HOL_elim21},
   @{thm abort_prep_recv_HOL_elim21},
   @{thm abort_wait_send_HOL_elim21},
   @{thm abort_wait_recv_HOL_elim21},
   @{thm abort_buf_send_HOL_elim21},
   @{thm abort_buf_recv_HOL_elim21},
   @{thm abort_map_send_HOL_elim2},
   @{thm abort_map_recv_HOL_elim2},
   @{thm abort_done_send_HOL_elim1'},
   @{thm abort_done_recv_HOL_elim1'}]]);
```

The tactic `abort_ipc_mbind_testGen_PureE21_ematch` is implemented on SML level using the different Isabelle SML libraries, the elements of the tactic are:

- `ALLGOALS`: a tactic combinator of type `tactic * tactic -> tactic` from the module `Tactical` of Isabelle/ML. It applies the tactic on all goals of a proof statement. A proof statement is usually called a proof context.

- `TestGen.REPEAT'`: a tactic combinator of type `(int -> tactic)-> int -> tactic`. It is an adaptation of `REPEAT_ALL_NEW`, from the module `Tactical` of Isabelle/ML for HOL-TestGen and it is used to repeat the same tactic on a given subgoal.

- `CHANGED`: a tactic combinator of type `tactic -> tactic`. Its apply the tactic on a given goal, and if it fails (i. e.the goal is not changed), an Isabelle fail error is raised.

- `TRY`: a tactic combinator of type `tactic -> tactic`. its apply the tactic on a given goal, and if it fails, it let the goal unchanged.

- `FIRST'`: a tactic combinator of type `('a -> tactic)list -> 'a -> tactic`. Tries a number of tactics, specified actually inside a list, on a given goal.

- `@{thm _}`: an antiquotation that refers to a given Isabelle theorem. Antiquotations are used as links to the object specified using Isabelle's specification constructs. The objects can be Isabelle theorems, types, theories, etc. Each object has its own type of antiquotation, e. g. in order to refer to a given Isabelle theory we use `@{theory theory_name}`, another antiquotation can be `@{context}`, it is used to refer to a given local context(proof statement) of a proof. Antiquotations are useful for many activities, e. g. they are useful in order to get formal links of the different objects in a given document generated from Isabelle theories, which helps for instance in the review of the document. Also they are useful for development, e. g. in the developement automated tactics.

- `abort_prep_send_HOL_elim21`: is a symbolic execution rules related to PikeOS IPC model.

For more details on Isabelle tactic development we would refer to [Urb13]. Moreover note, for more details on rules related to `abort__lift` see subsection 4.3.5.

The Specific Rules.

These rules are instantiations for the generic ones by a given operational semantics. For the case of PikeOS system, its operational semantics is expressed by a transition function (presented in subsection 4.3.5) over 10 atomic actions which are:

1. PREP SEND/RECV: in this stage some checks related to PikeOS message descriptor, i. e. a file containing informations about the communicating threads, are done.

2. WAIT SEND/RECV: The wait stage is mainly used for synchronisation.

3. BUF SEND/RECV : The stage BUF represent data transfer via memory copy.

4. MAP SEND/RECV : The stage MAP data transfer via memory mapping.

5. DONE SEND/RECV: The stage DONE used to finish the IPC communication between the threads.

As mentioned in the presvious section and in section 3.5, the role of the symbolic execution rule is to update the proof context according to the execution semantics of the different atomic actions of the IPC protocol. An example of a symbolic execution rule derived from the operational semantics of PikeOS IPC is:

```
lemma abort_wait_send_HOL_elim21:
  assumes
  valid_exec:
  (σ ⊨(outs ←(mbind ((IPC WAIT (SEND caller partner msg))#S)
                (abort__lift exec_action__id_Mon)); P outs))
  and in_err_exec:
    caller ∈dom (act_info (th_flag σ)) ⟹
     (σ ⊨(outs ←(mbind S(abort__lift exec_action__id_Mon));
                 P (get_caller_error caller σ# outs))) ⟹Q
  and
not_in_err_exec1:
 caller ∉dom (act_info (th_flag σ)) ⟹
   IPC_send_comm_check_st__id caller partner σ⟹
   IPC_params_c4 caller partner ⟹
   IPC_params_c5 partner σ⟹
  (σ⦇current_thread := caller,
     thread_list := update_th_waiting caller (thread_list σ),
     error_codes := NO_ERRORS,
     th_flag     := th_flag σ⦈
       ⊨(outs ←(mbind S(abort__lift exec_action__id_Mon));
                  P (NO_ERRORS # outs))) ⟹Q
(...)
not_in_err_exec24:
 caller ∉dom (act_info (th_flag σ)) ⟹
  IPC_send_comm_check_st__id caller partner σ⟹
  IPC_params_c4 caller partner ⟹
 ¬IPC_params_c5 partner σ⟹
  ∃th. (thread_list σ) caller = Some th ⟹
 (σ⦇current_thread := caller ,
    thread_list := update_th_current caller (thread_list σ),
    error_codes := ERROR_IPC error_IPC_5_in_WAIT_SEND,
    th_flag     := th_flag σ
    ⦇act_info := act_info (th_flag σ)
    (caller ↦(ERROR_IPC error_IPC_5_in_WAIT_SEND),
     partner ↦(ERROR_IPC error_IPC_5_in_WAIT_SEND))⦈⦈ ⊨
    (outs ←(mbind S(abort__lift exec_action__id_Mon));
           P (ERROR_IPC error_IPC_5_in_WAIT_SEND# outs)))⟹Q
  shows Q
```

Other Rules.

In order to simplify the proof of the symbolic execution rules presented earlier, other rules related to the execution semantics of PikeOS were derived:

```
lemma abort_prep_send_obvious10':
 (σ ⊨(outs ←(mbind ((IPC PREP (SEND caller partner msg))#S)
                  (abort__lift exec_action__id_Mon)); P outs)) =
  ((caller ∈dom ((act_info o th_flag)σ) ⟶
  (σ ⊨(outs ←(mbind S(abort__lift exec_action__id_Mon));
             P (get_caller_error caller σ# outs)))) ∧
 (caller ∉dom ((act_info o th_flag)σ) ⟶
 (∀a b. (a = NO_ERRORS ⟶
  exec_action_id_Mon (IPC PREP (SEND caller partner msg)) σ=
  Some (NO_ERRORS, b) ⟶
   (σ⦇current_thread := caller,
     thread_list := update_th_ready caller (thread_list σ),
     error_codes := NO_ERRORS,
     th_flag := th_flag σ⦈⊨
     (outs ←(mbind S(abort__lift exec_action__id_Mon));
            P (NO_ERRORS # outs)))) ∧
  (∀error_memory. a = ERROR_MEM error_memory ⟶
   exec_action__id_Mon (IPC PREP (SEND caller partner msg)) σ=
   Some (ERROR_MEM error_memory, b) ⟶
    (σ⦇current_thread := caller,
      thread_list := update_th_current caller (thread_list σ),
      error_codes := ERROR_MEM error_memory,
      th_flag    :=
      th_flag σ
      ⦇act_info := ((act_info o th_flag)σ)
      (caller ↦(ERROR_MEM error_memory),
       partner ↦(ERROR_MEM error_memory))⦈⦈
  (...)
```

Moreover, in order to optimize the process, some rules called behavioral refinement rules are derived:

```
lemma abort_prep_send_obvious0:
assumes not_in_err :
       caller ∉dom (act_info (th_flag σ))
and   ioprog_success:
      ioprog (IPC PREP (SEND caller partner msg)) σ=
       Some(NO_ERRORS, σ')
shows abort__lift ioprog (IPC PREP (SEND caller partner msg)) σ=
       Some(NO_ERRORS, (error_tab_transfer caller σσ'))
using assms
by simp
```

For more details on these rules we would refer to section 4.19.

4.5.4 Abstract Test Cases

Abstract test cases are proof goals resulting from the application of symbolic execution and the normalization processes on a given test specification. Abstract test cases represent a *possible* execution path in the system under test. In our approach, having n number of abstract test cases does not necessarily mean that all n paths are *feasible*. An abstract test case is feasible if and only if there exist a model, i. e. an instatiation of the free variables by a witness, that satisfy the premises of the abstract test case. In our approach, the number of feasible test cases is always less than or equal to the number of abstract test cases resulting from symbolic execution and normalization processes. The number of feasible abstract test cases is not necessarily equal to the number of concrete tests. A concrete test is a witness used to justify that a given abstract test case is feasible. Many witnesses can exist and used for the justification. Actually, in some cases the number of witnesses can be infinite. Of course, if no witnesses can be derived for an abstract test case this means that the abstract test case is *infeasible*. Thus, in our approach we can clearly end with 0 concrete tests for a given test scenario and this can happen if the constraint-solver can not provide a model that satisfies the *proof obligations* of the formula that represent an abstract test case. The problems related to detecting feasible abstract test cases, and the elimination of infeasible ones before the test generation, is not tackled during this thesis. An example of an abstract test case is:

```
⋀z za y.
(...) =
 [e, f, g] ⟹
 (...) =
 [a, b, C] ⟹
 IPC_send_comm_check_st__id thID2 thID1 σ_1 ⟹
 IPC_params_c4 thID2 thID1 ⟹
 IPC_params_c5 thID1 σ_1 ⟹
 act_info (th_flag σ_1) thID2 = None ⟹
 ¬ IPC_buf_check_st__id thID2 thID1
    (σ_1⦇current_thread := thID2,
        thread_list :=
          if thID2 ∈dom (thread_list σ_1)
          then thread_list σ_1(thID2 ↦(the ∘thread_list σ_1) thID2
              ⦇th_state := WAITING⦈)
          else thread_list σ_1,
        error_codes := NO_ERRORS⦈) ⟹
 thID1 ≠ thID2 ⟹
 act_info (th_flag σ_1) thID1 = Some y ⟹
 σ_1 ⊨
 ( outs ←mbind
         [IPC WAIT (RECV thID1 thID2 [z, za]),
          IPC WAIT (SEND thID2 thID1 [z, za]),
          IPC BUF (SEND thID2 thID1 [z, za]),
          IPC MAP (SEND thID2 thID1 [z, za]),
          IPC DONE (SEND thID2 thID1 [z, za]),
          IPC DONE (RECV thID1 thID2 [z, za])]
          PUT2; unit_SE
         (outs =
          [y, NO_ERRORS,
           ERROR_IPC error_IPC_1_in_BUF_SEND,
           ERROR_IPC error_IPC_1_in_BUF_SEND,
           ERROR_IPC error_IPC_1_in_BUF_SEND,
           ERROR_IPC error_IPC_1_in_BUF_SEND]))
```

In order to get a concrete test case we have to instantiate this abstract test case with witnesses for the variables `z, za, y`. The instantiation process is done by sending the formula that contains the conjunction of the premises, e. g. `IPC_params_c4 thID2 thID1`, to constraint-solvers via an interface provided by HOL-TestGen. In our terminology, the conjunction between the premises of an abstract test case is called Proof Obligation (PO).

Most of the time, a configuration is needed in order to help the constraint solver to reason about proof obligations. The configuration of the constraint solver is basically done by a set of Isabelle lemmas that help in the solving process of the PO. For technical reasons, the lemmas of the configuration must be written in **hol!** language, and not in `isar` or `pure` language. For example in order to allow the constraint-solver `smt` to reason about properties related to our abstract memory model, we use the rule:

```
lemma adde_share_charn [simp, code_unfold]:
 assumes 1: ¬(i shares (σ) k')
 and    2: ¬(k shares (σ) k')
 shows i shares(σ(i' ⋈k')) k = i shares (σ) k
 using assms fun_upd_apply id_def mem_adde_E sharing_def sharing_refl
 by metis
```

In its current form this rule will be refused by the solver `smt`. The following adaptation is needed:

```
lemma adde_share_charn_smt :
 ¬(i shares (σ) k') ∧
  ¬(k shares (σ) k') ⟶
  i shares (σ(i' ⋈k')) k = i shares (σ) k
  using adde_share_charn
  by simp
```

In our framework, and in order to feed the solver `smt` with the rule `adde_share _charn_smt` we use the command:

```
declare adde_share_charn_smt [testgen_smt_facts]
```

We have to notice that we experienced several problems related to solving a PO containing constraints around an abstract type,e. g. the type of our memory model. For example, in some cases the `smt` solver fails to provide a solution to a PO containing a constraint of the form `(i shares k)`, and this of course because we do not have yet a perfect lemmas configuration that help the solver to reason about the `shares` relation correctly.

4.5.5 Test Data For Sequence-based Test Scenarios

A test scenario is represented by a test specification and can have two main schemes: unit test or sequence test. The specification `TS_simple_example2` is an example of a sequence test scenario for PikeOS IPC.

```
test_spec TS_simple_example2:
  is ∈IPC_communication ⟹
  σ₁ ⊨(outs ←mbind is(abort__lift exec_action_Mon);return(outs = x)
⟶σ₁ ⊨(outs ←mbind is SUT; return(outs = x))
```

For a σ_1 definition that contains a suitable VMIT configuration, possible generated values for `is` are, e. g.:

```
[IPC PREP (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC PREP (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC WAIT (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC WAIT (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC BUF (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (RECV (0,0,1) (0,0,2) [0,4,5,8])]
```

The sequence is an abstraction of an IPC communication between the thread with the $ID = (0, 0, 1)$ and the thread with $ID = (0, 0, 2)$ via a message $msg = [0, 4, 5, 8]$. Natural numbers inside the message are abstractions on memory addresses. In `TS_simple_example2` the execution semantic of the input sequence is represented by our execution function *exec_action_Mon*. We wrapped around our execution function a monad transformer $abort_{lift}$ that express the behavior of an abort. The equality in `return(outs = x)` specify our conformance relation between SUT outputs and the model outputs. After using our symbolic execution process the `out` of this test case is:

```
[NO_ERRORS,
 NO_ERRORS,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV]
```

The error-codes observed in the sequence is related to IPC. The error-codes was returned in the stage *WAIT_ RECV*. The interpretation of this error-codes is that the thread has not the rights to communicate with his partner. We can observe the behavior of our abort operator in this sequence of error-codes; All stages following `WAIT_RECV` are purged (not executed), and the same error is returned instead. We focus on error-codes in our scenarios, since error-codes represent a potential for undesired information flow: for example, un-masked error-messages may reveal the structure of tasks and threads of a foreign partition in the system; a revelation that the operating system as separation kernel should prevent.

4.5.6 Test Drivers

In this section we address the problem to compile "abstract test-drivers" as described in the previous sections into concrete code and code instrumentations that actually execute these tests.

HOL-TestGen can generate test scripts in SML, Haskell, Scala and F#. For our application, we generate SML test scripts and use MLton (www.mlton.org) for building the test executable: MLton 1. provides a foreign function interface to C and 2. is easily portable to small POSIX system (it mainly requires a C compiler, libc, and libm).[1]

In more detail, we generate two SML structures *automatically* from the Isabelle theories. The first structure, called `Datatypes`, contains the datatypes that are used by the interface of the SUT. In our example, this includes, e. g., `IPC_protocol` and `P4_IPC_call`. The second structure, called `TestScript`, contains a list of all generated test cases as well the *test oracle*, i. e., the algorithms necessary to decide if a test result complies to the specification or not. In addition, HOL-TestGen provides a test harness (as SML structure `TestHarness`) that 1. takes the list of test cases (from `TestScript`) and executes them on the SUT, 2. uses the test oracle (also from `TestScript`) to decide if the actual test results complies to the specification, and 3. provides statistics about the number of successful and failed tests as well as errors (e. g., unexpected exceptions) during test execution.

In addition, for testing C code, we need to provide a small SML structure (ca. 20 lines of code), called `Adapter`, that serves two purposes: 1. the configuration of the foreign function, e. g., the mapping from SML datatypes to C datatypes and 2. the concretization of abstractions to bridge the gap between an abstract test model and the concrete SUT.

An example for a concretization would be a test specification using an an enumeration to encode error states while the implementation uses an efficient encoding as bit vector. The `Adapter` structure only needs to be updated after significant changes to either the system specification or the system under test.

For testing concurrent, i. e., multi-threaded, programs we need to solve a particular challenge: *enforcing certain thread execution orders* (a certain scheduling) during test execution. There are, in principle, three different options available to control the scheduler during test execution: 1. instrumenting the SUT to make the thread switching deterministic and controllable, 2. using a deterministic scheduler that can be controlled by test driver, or 3. using the features of debuggers, such as the GNU debugger (gdb), for

[1] In our code generation setup, we avoid the use of the SML datatype `Int.Inf` and, by this, we can remove the dependency on the GNU multi-precision library (libgmp).

multi-threaded programs.

In our prototype for POSIX compliant systems, we have chosen the third option: we execute the SUT within a gdb session and we use the gdb to switch between the different threads in a controlled way. We rely on two features of gdb (thus, out approach can be applied to any other debugger with similar features), namely: 1. the possibility to attach to break points in the object code scripting code that is executed if a break point is reached and 2. the complete control of the threading, i. e., gdb allows to switch explicitly between threads while ensuring that only the currently active thread is executed (using the option `set scheduler-locking on`).

This approach has the advantage that we neither need to modify the SUT nor do we need to develop a custom scheduler. We only need to generate a configuration for controlling the debugger. The necessary gdb command file is generated automatically by HOL-Testgen based on a mapping of the abstract thread switching points to break points in the object code. The break points at the entry points allows us to control the thread creation, while the remaining break points allow us to control the switching between threads. Thus, we only need the SUT compiled in debugging mode and this mapping. In this sense, we still have a "black-box" testing approach.

Moreover, Using gdb together with `taskset`, we ensure that all threads are executed on the same core; in our application, we can accept that the actual execution in gdb changes the timing behavior. Moreover, we assume a sequential memory model, so our approach does not cover TLB-related race conditions occurring in multi-core CPU's.

A note on testing small embedded systems and low-level operating system code. This setup works well for mid-size embedded systems to large systems using standard desktop or server operating systems. It does not work for small embedded systems or for testing small operating system kernels or hypervisors. Such system often to neither provide a rich enough `libc` (or `libm`) nor enough system resources that allows to run the complete test driver on the system under test. For such systems, we envision a host-target setup, where only a very small target library needs to be ported to the target system. This target library serves mainly two purposes: 1. stimulate, remotely controlled from the host system, the functions under test and 2. collect the test result and report it back to the host system. All expensive computation such as comparing test results, creating statistics are executed on the host system.

Finally, for small systems it might be necessary to develop a custom scheduler, e. g., similar to [MQB07], to control the execution order of multi-threaded programs.

4.5.7 Experimental Results

In this section we will discuss our test experiences, the obtained results and the different problems encountered. The table Table 4.1 represent 5[2] different test specifications related to PikeOS IPC, i. e. test scenarios for PikeOS IPC API, and also the statistics related to the application of the different steps of our test generation process on these scenarios. Four columns are distinguished in Table 4.1:

1. **SE**: is the step related to the symbolic execution process. During this step the derived symbolic execution rules related to PikeOS IPC are applied on the scenario.

2. **Norm**: represent the step of our normalization process. During this step we apply tactics like `simp` and other derived rules from the model in order to eliminate contradictory proof goals resulting from the SE step.

3. **TT**: is the step of the generation of the test theorem. During this step we use a HOL-TestGen tactic to determine the PO and to introduce uniformity testing hypotheses on the different proof goals resulting from the Norm step. This step can be seen as a preparatory step for the data selection process.

[2] actually we designed 38 scenario, we did not finish all the experiments at submission time, further explanation are presented in the sequel.

4. **TD**: represent the step of test data selection. During this step we send the POs in the test theorem to constraint-solvers. Also, after that a given solver choose a model for the POs an Isabelle proof is mandatory in order to make sure that the chosen model satisfies the PO. We have to notice that, for simple models, the process of proving the satisfaction of the PO by the chosen model, is done automatically by an Isabelle tactic but, for complicated models such as PikeOS model, where its symbolic execution results with complicated predicated defined around abstract types, e. g. predicate around our memory model, the proofs need to be done manually. This does not mean that the process can not be automated, but at the moment, we do not have the set of lemmas and the corresponding tactics that help to get a such automatic setup.

Each column in Table 4.1 is composed of two other columns. The columns named *Num* contain the number of outputs from each step of the generation process, and the columns *Time* contain the duration of the step by minutes. The scenarios Sc1 and Sc2 contain the value $undet$ in their columns, it means that we did not manage to finish the steps of the generation and the experience is done for these scenarios. The judgement $undet$ is different from the judgement represented by the symbol $-$, also contained in the table. The judgement $undet$ is applied to an experience where our process of test generation had failed in a given step, and we are not trying to fix the failed part because, the fixes depends on major changes in the various levels of the tool-chain. The judgement $-$ is applied on an experience which is not finished yet, i. e. we do not have the results of all the steps of the process but, finishing the experience depends on manageable technical problems [3].

Note that the execution of the steps related to the test generation process is sequential. Thus, if the current step fails the next one can not be executed. For example during the scenario Sc1, we had derived actually 69984 symbolic test cases in 2 hours for 1 input sequence that represent an IPC communication (recall subsection 4.5.1) but, we did not manage to normalize a such proof context with a such size, which means that all remaining steps of the process can not be performed because they all depend of the outputs from the Norm step.

As explained in subsection 4.5.4, the generation of 69984 symbolic test cases does not necessarily mean that all the cases, represented by proof goals, are feasible. We have to normalize the proof goals in order to eliminate the contradictory ones. Even if we have managed to normalize a proof context with a such size, we still need to find models for the different normalized goals and prove that, the chosen models satisfy the POs. While the fact of generating almost 70000 goals using our symbolic approach in only 2 hours can be seen as an impressive result, we have failed during the normalization process, and this come back to:

1. **The model.** the model of PikeOS IPC is heavy, and this because of the branching in the atomic actions, especially the PREP action.

2. **The way of modeling.** it is the main influential factor. We believe that some changes on the way of modeling can help to make the normalization process lighter. e. g. the definition of meta-predicates that characterize feasible paths only, or at least the elimination of the most of infeasible paths, and accordingly, the definition of the corresponding symbolic execution rules, can actually result with an optimized proof context after the SE step.

In order to execute our tool-chain from top to bottom we have tried other test scenarios to avoid the previous cited problems. For example, the scenario Sc2 is similar to the scenario Sc1 but, without including the PREP stage in the input sequence that represent 1 IPC communication. From Sc2, we had derived 1973 symbolic test cases in 2 minutes (which is another impressive result). After 6 hours of normalization process, 27 abstract test case remained. But still we did not manage to get automatically models for the 27 abstract test cases, and this because of a failure from the constraint-solvers, such `smt`, to provide a solution for complicated POs. The failure come back mainly to missing lemmas used

[3] At submission time of this document, we had managed to finish only 4 experiences.

as a configuration (recall last paragraphs in subsection 4.5.4) for the constraint solvers and not to the constraints-solver design.

For the scenarios Sc3 to Sc5, we have tried another approach in order to deal with the previous cited problems and also to generate test cases that cover communications with PREP action. Basically the approach is based on a technique that, allows to force a given execution path from the possible ones, resulting from the execution of the PREP action. Actually, after the execution of a PREP action, 6 execution paths are possible (see the symbolic execution rule for PREP action in section 4.22). Since we have 2 PREP actions in the head of a sequence that represent 1 IPC communication, all possible execution paths related to the 2 PREP actions is equal to 6×6. Actually, the 2 PREP actions are derived from: the ipc send system call for the PREP SEND action, and the ipc receive system call for PREP RECV. Each system call is executed by a thread. Instead to opt for a standard execution of the 2 PREP actions with rules that simulate all possible executions paths like we did in Sc1, we had opted for rules that force one execution path inside a test scenario. In order to cover all paths, we had designed 36 scenarios, each scenario force a given execution path during the PREP stage. Because we do not have any problems for executing the other actions which are different from PREP, we used the standard rules for their symbolic execution.

In order to apply this new tchnique to our scenarios, new symbolic execution rules were designed to cope with the explosion in the number of the abstract test cases, which influence negatively our normalization process. For example, in the scenario Sc3 we had derived 2 new symbolic execution rules for PREP actions. Each rule characterize one execution path by assuming that the path-predicate that describe the execution path is true. The symbolic execution rules used to simulate the the behavior of the actions PREP_SEND and PREP_RECV in the scenario Sc3 are:

```
lemma abort_prep_send_HOL_elim21'_factor:
  assumes valid_exec:
   (σ ⊨(outs ←(mbind ((IPC PREP (SEND caller partner msg))#S)
        (abort__lift exec_action_id_Mon)); P outs))
  and in_err_exec1:
     caller ∈dom (act_info (th_flag σ))
  and in_err_exec:
     caller ∈dom (act_info (th_flag σ)) ⟹
      (σ ⊨(outs ←(mbind S(abort__lift exec_action_id_Mon));
           P (get_caller_error caller σ# outs))) ⟹Q

  shows Q
  apply (insert valid_exec)
  apply (elim abort_prep_send_mbindFSave_E')
  apply (simp add: in_err_exec)
  apply (simp add: in_err_exec1)+
  done
```

```
lemma abort_prep_recv_HOL_elim21'_factor:
  assumes valid_exec:
  (σ ⊨(outs ←(mbind ((IPC PREP (RECV caller partner msg))#S)
        (abort__lift exec_action_id_Mon)); P outs))
  and in_err_exec1:
   caller ∈dom (act_info (th_flag σ))
  and in_err_exec:
    caller ∈dom (act_info (th_flag σ)) ⟹
        (σ ⊨(outs ←(mbind S(abort__lift exec_action_id_Mon));
                          P (get_caller_error caller σ# outs))) ⟹Q

   shows Q
  apply (insert valid_exec)
  apply (elim abort_prep_recv_mbindFSave_E')
  apply (simp add: in_err_exec)
  apply (simp add: in_err_exec1)+
  done
```

Of course the path-predicate `in_err_exec1` must be expressed also in the test specification Sc3. This predicate express the fact that the caller of the action (the caller of PREP SEND and also the caller of PREP RECV), was in an error-state (recall subsection 4.3.4).

Scenarios	SE		Norm		TT		TD	
	Num	Time	Num	Time	Num	Time	Num	Time
Sc1	69984	120	*undet*	*undet*	*undet*	*undet*	*undet*	*undet*
Sc2	1973	2	27	360	1	162	*undet*	*undet*
Sc3	1973	2	2	0.01	1	120	2080	0.23
Sc4	1973	2	-	-	-	-	-	-
Sc5	1973	2	-	-	-	-	-	-

Table 4.1: Statistics for our TestGen Process

From another side, we did not manage to execute the generated tests on PikeOS sources, for confidentiality reasons. In order to evaluate our approach we had implemented a PikeOS IPC-like environment using POSIX implementation. We had managed to execute 2 scenarios on this PikeOS demonstrator. Of course, when the state of the PikeOS demonstrator is initialised correctly our tests did not found any bugs. If the state is not initialised correctly our generated tests detect the bugs. Finally, we still have problems to define a program that initialise automatically the state of the demonstrate and bring it to the same value generated by the model. At the moment this step is done manually, and this due to some technical chanllenges like, how to export or import the values of a static array defined on C-level to the sml-level. Finally, another technical challenge is that GDB can not run an executable containing a `Main.sml` function defined in `sml` language. In order to deal with this problem, we have to define a `Main.c` function on C-level and call our `harness.sml` inside the `Main.c`, and this using the foreign function interface of `MLton`.

For the highest level the developer also needs to provide some formal representation of the high level design. In addition to couverage also test depth needs to be analysed, which means that the possible interactions between subsystems are to be sufficiently covered by tests. It could be an interesting to extend the HOL-TestGen approach into this direction. Due to the lack of a formal representation of the high level design (FSP), this could not be done in EURO-MILS.

In the EUROMILS SYSGO evaluated how to integrate generated test data into existing requirement engineering and testing processes. The steps of the test sequences are at a granularity of preemption points. This is a granularity that is smaller than interface-based test cases, which are targeting at function invocations, but not at preemption points.

4.6 Conclusion

4.6.1 Related Work.

There is a wealth of approaches for tests of behavioral models; they differ in the underlying modeling technique, the testability and test hypothesis', the test conformance relation etc.; in section 3.2 we mention a few. Unfortunately, many works make the underlying testability hypothesis' not explicit which makes a direct comparison difficult and somewhat vague. For the space of testability assumptions used here (the system is input-output deterministic, is adequately modeled as underspecified deterministic system, synchronous coupling between tester and SUT suffices), to the best of our knowledge, our approach is unique in its integrated process from theory, modeling, symbolic execution down to test-driver generation.

With respect to the test-driver approach, this work undeniably owes a lot Microsoft's CHESS project [MQB07], which promoted the idea to actually control the scheduler of real systems and use partial-order reduction techniques to test systematically concurrent executions for races in applications of realistic size (e. g., IE, Firefox, Apache). For our approach, controlling the scheduler is the key to justify the presentation of the system as underspecified-deterministic transition function.

4.6.2 Conclusion and Future Work.

We see several conceptual and practical advantages of a *monadic approach* to sequence testing:

1. a monadic approach resists the tendency to surrender to finitism and constructivism at the first-best opportunity; a tendency that is understandably wide-spread in model-checking communities,
2. it provides a sensible shift from syntax to semantics: instead of a first-order, intentional view in *nodes* and *events* in automata, the heart of the calculus is on *computations* and their *compositions*,
3. the monadic theory models explicitly the difference between input and output, between data under control of the tester and results under control of the SUT,
4. the theory lends itself for a theoretical and practical framework of numerous conformance notions, even non-standard ones, and which gives
5. ways to new calculi of symbolic evaluation enabling symbolic states (via invariants) and input events (via constraints) as well as a seamless, theoretically founded transition from system models to test-drivers.

We see several directions for future work: On the model level, the formal theory of sequence testing (as given in the HOL-TestGen library theories `Monad.thy` and `TestRefinements.thy`) providing connections between monads, rules for test-driver optimization, different test refinements, etc., is worth further development. On a test-theoretical level, our approach provides the basis for a comparison on test-methods, in particular ones based on different testability hypothesis'.

Pragmatically, our test driver setup needs to be modified to be executable on the PikeOS system level. For this end, we will need to develop a host-target setup (see subsection 4.5.6). Finally, we are interested in extending our techniques to actually test information flow properties; since error-codes in applications may reveal internal information of partitions (as, for example, the number of its tasks and threads), this seems to be a rewarding target. For this purpose, not only action sequences need to be generated during the constraint solving process, but also (abstract) VMITs.

Part IV

Annexes

HOL-TestGen 1.7.0-dev (svn. rev. 11222:11225M) User Guide

http://www.brucker.ch/projects/hol-testgen/

update authors

Achim D. Brucker
a.brucker@sheffield.ac.uk
The University of Sheffield, Sheffield, UK

Lukas Brügger
lukas.a.bruegger@gmail.com
ETH, Zürich, Switzerland

Matthias P. Krieger
Matthias.Krieger@lri.fr
LRI, Orsay, France

Burkhart Wolff
wolff@lri.fr
LRI, Orsay, France

February 4, 2016

Laboratoire en Recherche en Informatique (LRI)
Université Paris-Sud 11
91405 Orsay Cedex
France

Note:
This manual describes HOL-TestGen version 1.7.0-dev (svn. rev. 11222:11225M). The manual of version **1.8.0** is also available as technical report number **TR number to be requested** from the Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France.

Contents

1. Introduction

Today, essentially two validation techniques for software are used: *software verification* and *software testing*. Whereas verification is rarely used in "real" software development, testing is widely-used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Dijkstra's verdict* [18, p.6]:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

Abstraction Techniques: model-checking raised interest in techniques to abstract infinite to finite models. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [9, 17].

Systematic Testing: the discussion over *test adequacy criteria* [25], i. e. criteria solving the question "when did we test enough to meet a given test hypothesis," led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [21, 19].

Specification Animation: constructing counter-examples has raised interest also in the theorem proving community, since combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [8, 22, 16].

The first two areas are motivated by the question "are we building the program right?" the latter is focused on the question "are we specifying the right program?" While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows, that it simply does not apply in practically important situations. In particular, if a formal model of the environment of a software system (e. g. based among others on the operation system, middleware or external libraries) must be reverse-engineered, testing ("experimenting") is without alternative (see [12]).

Following standard terminology [25], our approach is a *specification-based unit test*. In general, a test procedure for such an approach can be divided into:

Test Case Generation: for each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test Data Generation: (also: Test Data Selection) for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test Execution: the implementation is run with the selected test input data in order to determine the test output data.

Test Result Verification: the pair of input/output data is checked against the specification of the test case.

The development of HOL-TestGen [13] has been inspired by [20], which follows the line of specification animation works. In contrast, we see our contribution in the development of techniques mostly on the first and to a minor extent on the second phase.

Building on QuickCheck [16], the work presented in [20] performs essentially random test, potentially improved by hand-programmed external test data generators. Nevertheless, this work also inspired the development of a random testing tool for Isabelle [8]. It is well-known that random test can be ineffective in many cases; in particular, if preconditions of a program based on recursive predicates like "input tree must be balanced" or "input must be a typable abstract syntax tree" rule out most of randomly generated data. HOL-TestGen exploits these predicates and other specification data in order to produce adequate data, combining automatic data splitting, automatic constraint solving, and manual deduction.

As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis is valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification, see [11, 14] for details.

2. Preliminary Notes on Isabelle/HOL

2.1. Higher-order logic — HOL

Higher-order logic(HOL) [15, 7] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e. g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML (SML) or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

2.2. Isabelle

Isabelle [23, 1] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic (constructive and classical), Zermelo-Fränkel set theory and HOL, which we chose as the basis for the development of HOL-TestGen.

Isabelle consists of a logical engine encapsulated in an abstract data type *thm* in Standard ML; any *thm* object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (*tactics*) have been developed; namely a simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

We use the possibility to build on top of the logical core engine own programs performing symbolic computations over formulae in a logically safe (conservative) way: this is what HOL-TestGen technically is.

[1]to be more specific: *parametric polymorphism*

3. Installation

3.1. Prerequisites

HOL-TestGen is built on top of Isabelle/HOL, version 2013-2, thus you need a working installation of *Isabelle 2013-2*. To install Isabelle, follow the instructions on the Isabelle web-site:

`http://isabelle.in.tum.de/website-Isabelle2013-2/index.html`

If you use the pre-compiled binaries from this website, please ensure that you install both the `Pure` heap and `HOL` heap.

3.2. Installing HOL-TestGen

In the following we assume that you have a running Isabelle 2013-2 environment including the jEdit based front-end. The installation of HOL-TestGen requires the following steps:

1. Unpack the HOL-TestGen distribution, e. g.:

   ```
   tar zxvf hol-testgen-1.7.0-dev.tar.gz
   ```

   This will create a directory `hol-testgen-1.7.0-dev` containing the HOL-TestGen distribution.

2. Check the settings in the configuration file `hol-testgen-1.7.0-dev/make.config`. If you can use the `isabelle` tool from Isabelle on the command line to start Isabelle 2013-2, the default settings should work. The `ISABELLE` variable in `make.config` needs to point to the 2013-2 version of Isabelle. For this, it can be necessary to configure an absolute path, e.g.,

   ```
   ISABELLE=/usr/local/Isabelle2013-2/bin/isabelle
   ```

3. Change into the `top` directory

   ```
   cd hol-testgen-1.7.0-dev
   ```

   and build the HOL-TestGen heap image for Isabelle by calling

   ```
   isabelle build -d . -b HOL-TestGen
   ```

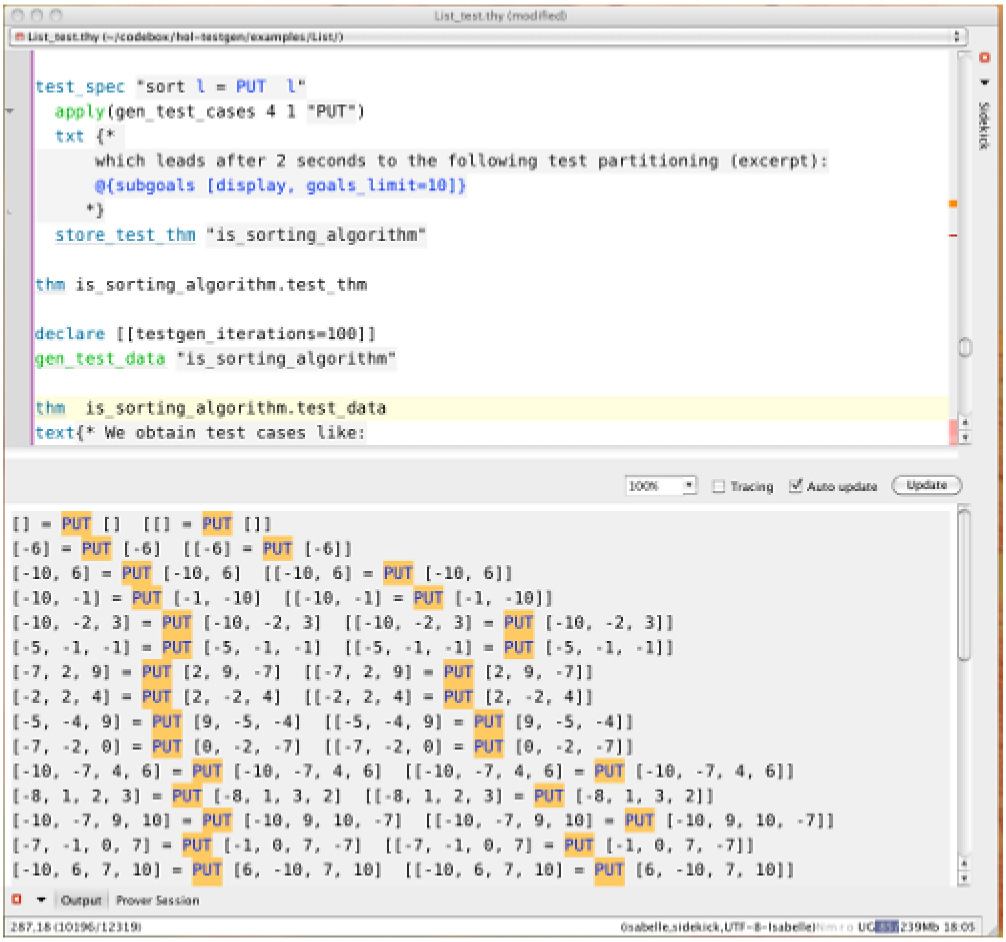


Figure 3.1.: A HOL-TestGen session Using the jEdit Interface of Isabelle

3.3. Starting HOL-TestGen

HOL-TestGen can now be started using the `isabelle` command:[1]

```
cd <hol-testgen-home>examples/unit/List
<isabelle2013-2-home> jedit -d ../../.. -l HOL-TestGen List_test.thy
```

After a few seconds you should see an jEdit window similar to the one shown in Figure 3.1.

Alternatively, it is possible to compile many examples, for example the above `List_test` without the `-l HOL-TestGen` option. This has the consequence that the HOL-TestGen components (libraries, SML files, ...) are included in the session and can be run or

[1]If, during the installation of HOL-TestGen, a working HOLCF heap was found, then HOL-TestGen's logic is called `HOLCF-TestGen`; thus you need to replace `HOL-TestGen` by `HOLCF-TestGen`, e. g. the interactive HOL-TestGen environment is started via `isabelle jedit -l HOLCF-TestGen`.

modified with the example together. This is particularly useful for debugging purposes. However, the paths to theories in the theory imports must then be expanded to their relative position.

Note that in some environments, jEdit is known to crash for unknown reasons when called the first time (this is not an Isabelle error). Just restarting should resolve the problem. In general, we strongly recommend to use the jEdit client as user-interface (instead of Proof General).[2] Use the system manual (see `http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/system.pdf`) as a high-level description of jEdit's system options; another source of information is the built-in README-facility inside the jEdit client.

[2]Still, in case you are using an non re-parenting window manager, you might want to stick to Proof General as jEdit has some problems with such window managers.

4. Using HOL-TestGen

4.1. HOL-TestGen: An Overview

HOL-TestGen allows one to automate the interactive development of test cases, refine them to concrete test data, and generate a test script that can be used for test execution and test result verification. The test case generation and test data generation (selection) is done in an Isar-based [24] environment (see Figure 4.1 for details). The test executable (and the generated test script) can be built with any SML-system.

4.2. Test Case and Test Data Generation

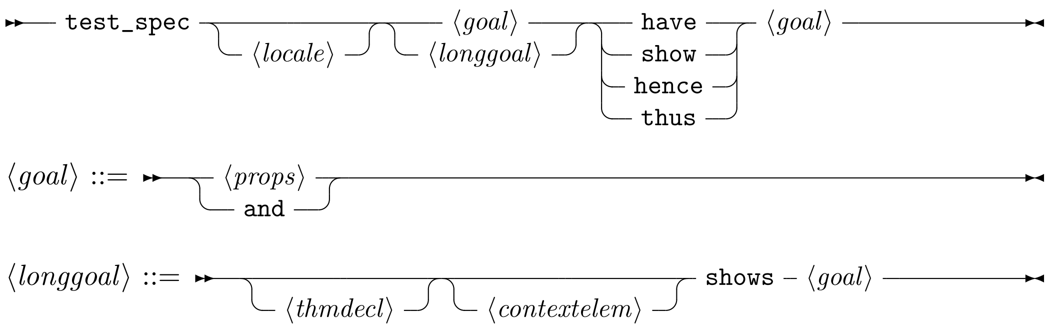
In this section we give a brief overview of HOL-TestGen related extension of the Isar [24] proof language. We use a presentation similar to the one in the *Isar Reference Manual* [24], e. g. "missing" non-terminals of our syntax diagrams are defined in [24]. We introduce the HOL-TestGen syntax by a (very small) running example: assume we want to test a function that computes the maximum of two integers.

Starting your own theory for testing: For using HOL-TestGen you have to build your Isabelle theories (i. e. test specifications) on top of the theory Testing instead of Main. A sample theory is shown in Table 4.1.

Defining a test specification: Test specifications are defined similar to theorems in Isabelle, e. g.,

> **test_spec** "prog a b = max a b"

would be the test specification for testing a simple program computing the maximum value of two integers. The syntax of the keyword `test_spec` : *theory* → *proof*(*prove*) is given by:



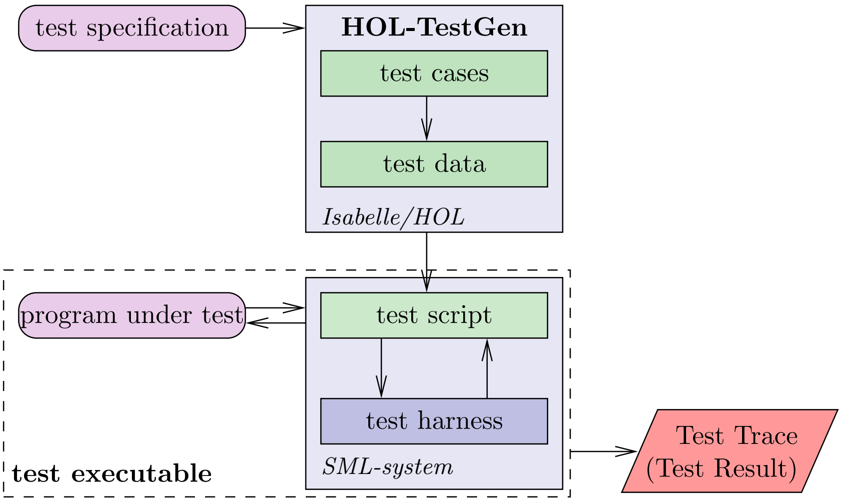


Figure 4.1.: Overview of the system architecture of HOL-TestGen

```
theory max_test
imports Testing
begin

test_spec "prog a b = max a b"
  apply(gen_test_cases "prog" simp: max_def)
  mk_test_suite "max_test"

gen_test_data "max_test"

thm max_test.concrete_tests

generate_test_script "max_test"
thm max_test.test_script

text {* Testing an SML implementation: *}
export_code max_test.test_script in SML   module_name TestScript file "impl/sml/max_test_script.sml"

text {* Finally, we export the raw test data in an XML-like format: *}
export_test_data "impl/data/max_data.dat" max_test

end
```

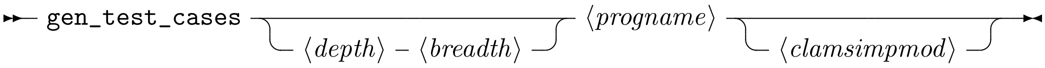
Table 4.1.: A simple Testing Theory

Please look into the Isar Reference Manual [24] for the remaining details, e. g. a description of ⟨*contextelem*⟩.

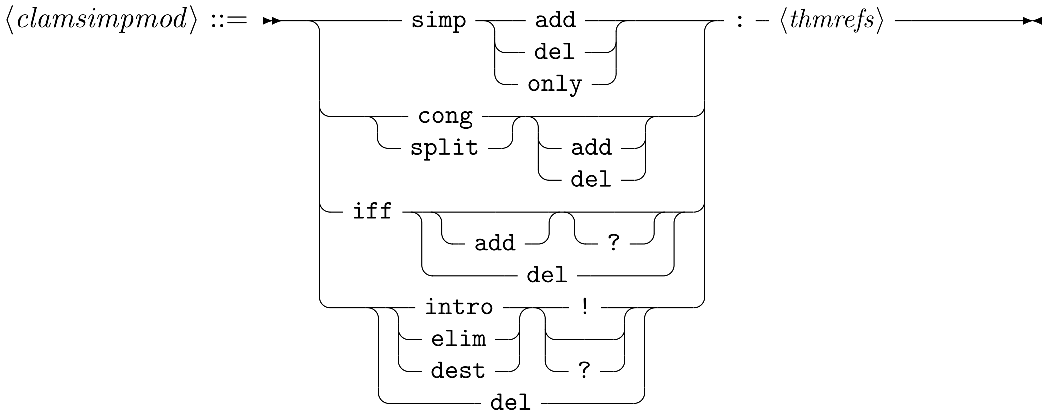
Generating symbolic test cases: Now, abstract test cases for our test specification can (automatically) be generated, e. g. by issuing

apply(gen_test_cases "prog" simp: max_def)

The `gen_test_cases` : *method* tactic allows to control the test case generation in a fine-granular manner:



where ⟨*depth*⟩ is a natural number describing the depth of the generated test cases and ⟨*breadth*⟩ is a natural number describing their breadth. Roughly speaking, the ⟨*depth*⟩ controls the term size in data separation lemmas in order to establish a regularity hypothesis (see [11] for details), while the ⟨*breadth*⟩ controls the number of variables occurring in the test specification for which regularity hypotheses are generated. The default for ⟨*depth*⟩ and ⟨*breadth*⟩ is 3 resp. 1. ⟨*progname*⟩ denotes the name of the program under test. Further, one can control the classifier and simplifier sets used internally in the `gen_test_cases` tactic using the optional ⟨*clasimpmod*⟩ option:



The generated test cases can be further processed, e. g., simplified using the usual Isabelle/HOL tactics.

Creating a test suite: HOL-TestGen provides a kind of container, called *test-suites*, which store all relevant logical and configuration information related to a particular test-scenario. Test-suites were initially created after generating the test cases (and test hypotheses); you should store your result of the derivation, usually the test-theorem which is the output of the test-generation phase, in a test suite by:

mk_test_suite "max_test"

for further processing. This is done using the `mk_test_suite` : *proof* (*prove*) $\rightarrow$ *proof* (*prove*) | *theory* command which also closes the actual "proof state" (or *test state*. Its syntax is given by:

$\rightarrowtail$ `mk_test_suite` – $\langle name \rangle$ $\rightarrowtail$

where $\langle name \rangle$ is a fresh identifier which is later used to refer to this test state. This name is even used at the very end of the test driver generation phase, when test-executions are performed (externally to HOL-TestGen in a shell). Isabelle/HOL can access the corresponding test theorem using the identifier $\langle name \rangle$.test_thm, e. g.:

thm max_test.test_thm

Generating test data: In a next step, the test cases can be refined to concrete test data:

gen_test_data "max_test"

The *gen_test_data* : *theory*|*proof* $\rightarrow$ *theory*|*proof* command takes only one parameter, the name of the test suite for which the test data should be generated:

$\rightarrowtail$ `gen_test_data` – $\langle name \rangle$ $\rightarrowtail$

After the successful execution of this command Isabelle can access the test hypothesis using the identifier $\langle name \rangle$.test_hyps and the test data using the identifier $\langle name \rangle$.test_data

thm max_test.test_hyps
thm max_test.concrete_test

In our concrete example, we get the output:

THYP ((∃x xa. x ≤xa ∧prog x xa = xa) ⟶ (∀x xa. x ≤xa ⟶ prog x xa = xa))
THYP ((∃x xa. ¬x ≤xa ∧ prog x xa = x) ⟶ (∀x xa. ¬ x ≤xa ⟶ prog x xa = x))

as well as :

prog −9 −3 = −3
prog −5 −8 = −5

By default, generating test data is done by calling the *random solver*. This is fine for such a simple example, but as explained in the introduction, this is far incomplete when the involved data-structures become more complex. To handle them, HOL-TestGen also comes with a more advanced data generator based on *SMT solvers* (using their integration in Isabelle, see e. g. [10]).

To turn on SMT-based data generation, use the following option:

declare [[testgen_SMT]]

(which is thus set to false by default). It is also recommenced to turn off the random solver:

declare [[testgen_iterations =0]]

In order for the SMT solver to know about constant definitions and properties, one needs to feed it with these definitions and lemmas. For instance, if the test case involves some inductive function foo, you can provide its definition to the solver using:

declare foo.simps [testgen_smt_facts]

as well as related properties (if needed).

A complete description of the configuration options can be found below.

Exporting test data: After the test data generation, HOL-TestGen is able to export the test data into an external file, e. g.:

export_test_data "test_max.dat" "max_test"

exports the generated test data into a file `text_max.dat`. The generation of a test data file is done using the *export_test_data* : *theory*|*proof* → *theory*|*proof* command:

`export_test_data` – ⟨*filename*⟩ – ⟨*name*⟩ [⟨*smlprogname*⟩]

where ⟨*filename*⟩ is the name of the file in which the test data is stored and ⟨*name*⟩ is the name of a collection of test data in the test environment.

Generating test scripts: After the test data generation, HOL-TestGen is able to generate a test script, e. g.:

gen_test_script "test_max.sml" "max_test" "prog"
"myMax.max"

produces the test script shown in Table 4.2 that (together with the provided test harness) can be used to test real implementations. The generation of test scripts is done using the *generate_test_script* : *theory*|*proof* → *theory*|*proof* command:

`gen_test_script` – ⟨*filename*⟩ – ⟨*name*⟩ – ⟨*progname*⟩ [⟨*smlprogname*⟩]

where ⟨*filename*⟩ is the name of the file in which the test script is stored, and ⟨*name*⟩ is the name of a collection of test data in the test environment, and ⟨*progname*⟩ the name of the program under test. The optional parameter ⟨*smlprogname*⟩ allows for the configuration of different names of the program under test that is used within the test script for calling the implementation.

Alternatively, the code-generator can be configured to generate test-driver code in other progamming languages, see below.

Configure HOL-TestGen: The overall behavior of test data and test script generation can be configured, e. g.

```
structure TestDriver : sig end = struct
  val return     = ref  ~63;
  fun eval x2 x1 = let
                       val ret = myMax.max x2 x1
                   in
                      ((return := ret);ret)
                   end
  fun retval ()  = SOME(!return);
  fun toString a = Int.toString a;
  val testres    = [];

  val pre_0   = [];
  val post_0  = fn () => ( (eval ~23 69 = 69));
  val res_0   = TestHarness.check retval pre_0 post_0;
  val testres = testres@[res_0];

  val pre_1   = [];
  val post_1  = fn () => ( (eval ~11 ~15 = ~11));
  val res_1   = TestHarness.check retval pre_1 post_1;
  val testres = testres@[res_1];

  val _ = TestHarness.printList toString testres;
end
```

Table 4.2.: Test Script

declare [[testgen_iterations =15]]

The parameters (all prefixed with testgen_) have the following meaning:

depth:	Test-case generation depth. Default: 3.
breadth:	Test-case generation breadth. Default: 1.
bound:	Global bound for data statements. Default: 200.
case_breadth:	Number of test data per case, weakening uniformity. Default: 1.
iterations:	Number of attempts during random solving phase. Default: 25. Set to 0 to turn off the random solver.
gen_prelude:	Generate datatype specific prelude. Default: true.
gen_wrapper:	Generate wrapper/logging-facility (increases verbosity of the generated test script). Default: true.
SMT:	If set to "true" external SMT solvers (e.g., Z3) are used during test-case generation. Default: false.
smt_facts:	Add a theorem to the SMT-based data generator basis.
toString:	Type-specific SML-function for converting literals into strings (e.g., `Int.toString`), used for generating verbose output while executing the generated test script. Default: `""`.
setup_code:	Customized setup/initialization code (copied verbatim to generated test script). Default: `""`.
dataconv_code:	Customized code for converting datatypes (copied verbatim to generated test script). Default: `""`.
type_range_bound:	Bound for choosing type instantiation (effectively used elements type grounding list). Default: 1.
type_candidates:	List of types that are used, during test script generation, for instantiating type variables (e.g., α list). The ordering of the types determines their likelihood of being used for instantiating a polymorphic type. Default: [int, unit, bool, int set, int list]

Configuring the test data generation: Further, an attribute *test* : *attribute* is provided, i. e.:

lemma max_abscase [test "maxtest"]:"max 4 7 = 7"

or

declare max_abscase [test "maxtest"]

that can be used for hierarchical test case generation:

→→ test – ⟨*name*⟩ →◂

```
structure myMax = struct
  fun max x y = if (x < y) then y else x
end
```

Table 4.3.: Implementation in SML of max

4.3. Test Execution and Result Verification

In principle, any SML-system, e. g. [5, 4, 6, 2, 3], should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test

- implementations using the .Net platform (more specific: CLR IL), e. g. written in C# using sml.net [6],
- implementations written in C using, e. g. the foreign language interface of sml/NJ [5] or MLton [3],
- implementations written in Java using mlj [2].

Also, depending on the SML-system, the test execution can be done within an interpreter (it is even possible to execute the test script within HOL-TestGen) or using a compiled test executable. In this section, we will demonstrate the test of SML programs (using SML/NJ or MLton) and ANSI C programs.

4.3.1. Testing an SML-Implementation

Assume we have written a max-function in SML (see Table 4.3) stored in the file `max.sml` and we want to test it using the test script generated by HOL-TestGen. Following Figure 4.1 we have to build a test executable based on our implementation, the generic test harness (`harness.sml`) provided by HOL-TestGen, and the generated test script (`test_max.sml`), shown in Table 4.2.

If we want to run our test interactively in the shell provided by sml/NJ, we just have to issue the following commands:

```
use "harness.sml";
use "max.sml";
use "test_max.sml";
```

After the last command, sml/NJ will automatically execute our test and you will see a output similar to the one shown in Table 4.4.

If we prefer to use the compilation manager of sml/NJ, or compile our test to a single test executable using MLton, we just write a (simple) file for the compilation manager of sml/NJ (which is understood both, by MLton and sml/NJ) with the following content:

```
Test Results:
=============
Test 0 -     SUCCESS, result:  69
Test 1 -     SUCCESS, result: ~11


Summary:
--------
Number successful tests cases:  2 of 2 (ca. 100%)
Number of warnings:             0 of 2 (ca. 0%)
Number of errors:               0 of 2 (ca. 0%)
Number of failures:             0 of 2 (ca. 0%)
Number of fatal errors:         0 of 2 (ca. 0%)

Overall result: success
===============
```

Table 4.4.: Test Trace

```
Group  is
harness.sml
max.sml
test_max.sml

#if(defined(SMLNJ_VERSION))
  $/basis.cm
  $smlnj/compiler/compiler.cm
#else
#endif
```

and store it as `test.cm`. We have two options, we can

- use sml/NJ: we can start the sml/NJ interpreter and just enter

  ```
  CM.make("test.cm")
  ```

  which will build a test setup and run our test.

- use MLton to compile a single test executable by executing

  ```
  mlton test.cm
  ```

  on the system shell. This will result in a test executable called `test` which can be directly executed.

In both cases, we will get a test output (test trace) similar to the one presented in Table 4.4.

```
int max (int x, int y) {
        if (x < y) {
                return y;
        }else{
                return x;
        }
}
```

Table 4.5.: Implementation in ANSI C of max

4.3.2. Testing Non-SML Implementations

Suppose we have an ANSI C implementation of max (see Table 4.5) that we want to test using the foreign language interface provided by MLton. First we have to import the max method written in C using the `_import` keyword of MLton. Further, we provide a "wrapper" function doing the pairing of the curried arguments:

```
structure myMax = struct
  val cmax     = _import "max": int * int -> int ;
  fun max a b = cmax(a,b);
end
```

We store this file as `max.sml` and write a small configuration file for the compilation manager:

```
Group  is
harness.sml
max.sml
test_max.sml
```

We can compile a test executable by the command

```
mlton -default-ann 'allowFFI true' test.cm max.c
```

on the system shell. Again, we end up with an test executable `test` which can be called directly. Running our test executable will result in trace similar to the one presented in Table 4.4.

4.4. Profiling Test Generation

HOL-TestGen includes support for profiling the test procedure. By default, profiling is turned off. Profiling can be turned on by issuing the command

`profiling_on`

Profiling can be turned off again with the command

`profiling_off`

When profiling is turned on, the time consumed by gen_test_cases and **gen_test_data** is recorded and associated with the test theorem. The profiling results can be printed by

`print_clocks`

A LaTeX version of the profiling results can be written to a file with the command

`write_clocks` – ⟨*filename*⟩

Users can also record the runtime of their own code. A time measurement can be started by issuing

`start_clock` – ⟨*name*⟩

where ⟨*name*⟩ is a name for identifying the time measured. The time measurement is completed by

`stop_clock` – ⟨*name*⟩

where ⟨*name*⟩ has to be the name used for the preceding start_clock. If the names do not match, the profiling results are marked as erroneous. If several measurements are performed using the same name, the times measured are added. The command

`next_clock`

proceeds to a new time measurement using a variant of the last name used.

These profiling instructions can be nested, which causes the names used to be combined to a path. The `Clocks` structure provides the tactic analogues `start_clock_tac`, `stop_clock_tac` and `next_clock_tac` to these commands. The profiling features available to the user are independent of HOL-TestGen's profiling flag controlled by profiling_on and profiling_off.

A. Glossary

Abstract test data : In contrast to pure ground terms over constants (like integers $1, 2, 3$, or lists over them, or strings ...) abstract test data contain arbitrary predicate symbols (like *triangle 3 4 5*).

Regression testing: Repeating of tests after addition/bug fixes have been introduced into the code and checking that behavior of unchanged portions has not changed.

Stub: Stubs are "simulated" implementations of functions, they are used to simulate functionality that does not yet exist ore cannot be run in the test environment.

Test case: An abstract test stimuli that tests some aspects of the implementation and validates the result.

Test case generation: For each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test data: One or more representative for a given test case.

Test data generation (Test data selection): For each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test execution: The implementation is run with the selected test input data in order to determine the test output data.

Test executable: An executable program that consists of a test harness, the test script and the program under test. The Test executable executes the test and writes a test trace documenting the events and the outcome of the test.

Test harness: When doing unit testing the program under test is not a runnable program in itself. The *test harness* or *test driver* is a main program that initiates test calls (controlled by the test script), i. e. drives the method under test and constitutes a test executable together with the test script and the program under test.

Test hypothesis : The hypothesis underlying a test that makes a successful test equivalent to the validity of the tested property, the test specification. The current implementation of HOL-TestGen only supports uniformity and regularity hypotheses, which are generated "on-the-fly" according to certain parameters given by the user like *depth* and *breadth*.

Test specification : The property the program under test is required to have.

Test result verification: The pair of input/output data is checked against the specification of the test case.

Test script: The test program containing the control logic that drives the test using the test harness. HOL-TestGen can automatically generate the test script for you based on the generated test data.

Test theorem: The test data together with the test hypothesis will imply the test specification. HOL-TestGen conservatively computes a theorem of this form that relates testing explicitly with verification.

Test trace: Output made by a test executable.

Bibliography

[1] Isabelle. URL http://isabelle.in.tum.de.

[2] MLj. URL http://www.dcs.ed.ac.uk/home/mlj/index.html.

[3] MLton. URL http://www.mlton.org/.

[4] Poly/ML. URL http://www.polyml.org/.

[5] SML of New Jersey. URL http://www.smlnj.org/.

[6] sml.net. URL http://www.cl.cam.ac.uk/Research/TSG/SMLNET/.

[7] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, May 1986. ISBN 0120585367.

[8] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM)*, pages 230–239. IEEE Computer Society, Los Alamitos, CA, USA, 2004. ISBN 0-7695-2222-X.

[9] A. Biere, A. Cimatti, Edmund Clarke, Ofer Strichman, and Y. Zhu. *Bounded Model Checking*. Number 58 in Advances In Computers. 2003.

[10] Sascha Böhme and Tjark Weber. Fast lcf-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010. ISBN 978-3-642-14051-8. URL http://dx.doi.org/10.1007/978-3-642-14052-5_14.

[11] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, Heidelberg, 2004. ISBN 3-540-25109-X. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-symbolic-2005.

[12] Achim D. Brucker and Burkhart Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology (STTT)*, 7(3):233–247, 2005. ISSN 1433-2779. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-verification-2005.

[13] Achim D. Brucker and Burkhart Wolff. HOL-TestGen: An interactive test-case generation framework. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, Heidelberg, 2009. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-2009`.

[14] Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012`.

[15] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[16] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, New York, NY USA, 2000. ISBN 1-58113-202-6.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, New York, NY USA, 1977.

[18] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 3rd edition, 1972. ISBN 0-12-200550-3.

[19] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, Heidelberg, April 1993.

[20] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying haskell programs by combining testing and proving. In *Proceedings of the Third International Conference on Quality Software*, page 272. IEEE Computer Society, 2003. ISBN 0-7695-2015-4. URL `http://csdl.computer.org/comp/proceedings/qsic/2003/2015/00/20150272abs.htm`.

[21] Marie Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, TAPSOFT *'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 82–96. Springer-Verlag, Heidelberg, 1995. ISBN 3-540-59293-8.

[22] Susumu Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.

[23] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/*HOL*—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[24] Markus Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 2004. URL `http://isabelle.in.tum.de/dist/Isabelle2004/doc/isar-ref.pdf`.

[25] Hong Zhu, Patrick A.V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997. ISSN 0360-0300. URL `http://www.cs.bris.ac.uk/Tools/Reports/Abstracts/1997-zhu.html`.

Index

theory *TypeSchemes*
imports *Main*
begin

4.7 HOL representation of PikeOS Datatypes

4.7.1 kernel state

record ($'resource$, $'thread\text{-}id$, $'thread$, $'sp\text{-}th\text{-}th$, $'sp\text{-}th\text{-}res$, $'errors$) *kstate* =
resource :: *'resource* — system ressources: memory, files..
current-thread :: *'thread-id* — a thread in the execution context..
thread-list :: *'thread* — list of threads in the system.
communication-rights :: *'sp-th-th* — security policy between threads..
access-rights :: *'sp-th-res* — security policy between threads and ressources..
error-codes :: *'errors* — error returned if a system call is aborted..

4.7.2 atomic actions

Atomic actions can be seen as instructions which can not be interrupted by the system scheduler during there execution. Each API has its own set of atomic actions.

datatype (*'ipc-stage*, *'ipc-direction*) $action_{ipc}$ =
IPC 'ipc-stage 'ipc-direction

datatype (*'mem-param1*, *'mem-param2*) $action_{mem}$ =
MEM 'mem-param1 'mem-param2

datatype (*'evn-param1*, *'evn-param2*) $action_{evn}$ =
EVN 'evn-param1 'evn-param2

datatype (*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *action* =
$atom_{ipc}$ (*'ipc-stage*, *'ipc-direction*) $action_{ipc}$
| $atom_{mem}$ (*'mem-param1*, *'mem-param2*) $action_{mem}$
| $atom_{evn}$ (*'evn-param1*, *'evn-param2*) $action_{evn}$

4.7.3 traces

A trace is sequence of atomic actions..

— An IPC actions trace

type-synonym (*'ipc-stage*, *'ipc-direction*) $trace_{ipc}$ =
(*'ipc-stage*, *'ipc-direction*) $action_{ipc}$ *list*

— A memory actions IPC trace

type-synonym (*'mem-param1*, *'mem-param2*) $trace_{mem}$ =
(*'mem-param1*, *'mem-param2*) $action_{mem}$ *list*

— An event actions trace

type-synonym (*'evn-param1*, *'evn-param2*) $trace_{evn}$ =
(*'evn-param1*, *'evn-param2*) $action_{evn}$ *list*

— A trace that contain all atomic actions

type-synonym (*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *trace* =
(*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *action list*

4.7.4 Threads

A thread is the smallest entity in the operating system.

record $('th\text{-}id, 'thstate, 'stipc, 'vadress, 'cpartner)$ *thread* =
thread-id :: *'th-id*
th-state :: *'thstate*
th-ipc-st :: *'stipc*
own-vmem-adr :: *'vadress*
cpartner :: *'cpartner*
end

4.8 A Shared-Memory-Model

theory *SharedMemory*
imports *Main*
begin

4.9 Shared Memory Model

4.9.1 Prerequisites

Prerequisite: a generalization of *fun-upd-def*: $?f(?a := ?b) \equiv \lambda x.$ *if* $x = ?a$ *then* $?b$ *else* $?f\, x$. It represents updating modulo a sharing equivalence, i.e. an equivalence relation on parts of the domain of a memory.

definition *fun-upd-equivp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$ **where**
fun-upd-equivp eq f a b = $(\lambda x.$ *if eq x a then b else f x*$)$

— This lemma is the same as *Fun.fun-upd-same*: $(?f(?x := ?y))\ ?x = ?y$; applied on our genralization *fun-upd-equivp ?eq ?f ?a ?b* = $(\lambda x.$ *if ?eq x ?a then ?b else ?f x*$)$ of $?f(?a := ?b) \equiv \lambda x.$ *if* $x = ?a$ *then ?b else ?f x*. This proof tell if our function *fun-upd-equivp op* $= f\, x\, y$ is equal to f this is equivalent to the fact that $f\, x = y$

lemma *fun-upd-equivp-iff*: $((\textit{fun-upd-equivp}\ (op =)\ f\ x\ y) = f) = (f\ \ x = \ y)$
by (*simp add :fun-upd-equivp-def, safe, erule subst, auto*)

— Now we try to proof the same lemma applied on any equivalent relation *equivp eqv* instead of the equivalent relation *op* =. For this case, we had split the lemma to 2 parts. the lemma *fun-upd-equivp-iff-part1* to proof the case when *eq* $(f\, a)\ b \longrightarrow$ *eq* (*fun-upd-equivp eqv f a b z*) $(f\, z)$, and the second part is the lemma *fun-upd-equivp-iff-part2* to proof the case *equivp eqv* $\Longrightarrow$ *fun-upd-equivp eqv f a b* $= f \longrightarrow f\, a = b$.

lemma *fun-upd-equivp-iff-part1*:
equivp R $\Longrightarrow (\bigwedge z.\ R\ x\ z \Longrightarrow R\ (f\ z)\ y) \Longrightarrow R$ (*fun-upd-equivp R f x y z*) $(f\ z)$
by (*auto simp*: *fun-upd-equivp-def Equiv-Relations.equivp-reflp Equiv-Relations.equivp-symp*)

lemma *fun-upd-equivp-iff-part2*: *equivp R* $\Longrightarrow$ *fun-upd-equivp R f x y* $= f \longrightarrow f\, x = y$
apply (*simp add :fun-upd-equivp-def, safe*)
apply (*erule subst, auto simp*: *Equiv-Relations.equivp-reflp*)
done

— Just anotther way to formalise *equivp ?R* $\Longrightarrow$ *fun-upd-equivp ?R ?f ?x ?y* $=$ *?f* $\longrightarrow$ *?f ?x* $=$ *?y* without using the strong equality

lemma *equivp R* $\Longrightarrow (\bigwedge z.\ R\ x\ z \Longrightarrow R$ (*fun-upd-equivp R f x y z*) $(f\ z)) \Longrightarrow R\ \ y\ (f\ x)$
by (*simp add*: *fun-upd-equivp-def Equiv-Relations.equivp-symp equivp-reflp*)

— this lemma is the same in $[\![equivp\ ?R;\ \bigwedge z.\ ?R\ ?x\ z \Longrightarrow ?R\ (?f\ z)\ ?y]\!] \Longrightarrow ?R\ (fun\text{-}upd\text{-}equivp\ ?R\ ?f\ ?x\ ?y\ ?z)\ (?f\ ?z)$ where $op =$ is generalized by another equivalence relation

lemma $fun\text{-}upd\text{-}equivp\text{-}idem$: $f\ x = y \Longrightarrow (fun\text{-}upd\text{-}equivp\ (op =)\ f\ x\ y) = f$
  by $(simp\ only: fun\text{-}upd\text{-}equivp\text{-}iff)$

lemma $fun\text{-}upd\text{-}equivp\text{-}triv$: $fun\text{-}upd\text{-}equivp\ (op =)\ f\ x\ (f\ x) = f$
  by $(simp\ only: fun\text{-}upd\text{-}equivp\text{-}iff)$

— This is the generalization of $fun\text{-}upd\text{-}equivp\ op =\ ?f\ ?x\ (?f\ ?x) = ?f$ on a given equivalence relation

lemma $fun\text{-}upd\text{-}equivp\text{-}triv\text{-}part1$:
  $equivp\ R \Longrightarrow (\bigwedge z.\ R\ x\ z \Longrightarrow fun\text{-}upd\text{-}equivp\ (R')\ f\ x\ (f\ x)\ z) \Longrightarrow f\ x$
  apply $(auto\ simp: fun\text{-}upd\text{-}equivp\text{-}def)$
  apply $(metis\ equivp\text{-}reflp)$
done

lemma $fun\text{-}upd\text{-}equivp\text{-}triv\text{-}part2$:
  $equivp\ R \Longrightarrow (\bigwedge z.\ R\ x\ z \Longrightarrow f\ z\ ) \Longrightarrow fun\text{-}upd\text{-}equivp\ (R')\ f\ x\ (f\ x)\ x$
  by $(simp\ add: fun\text{-}upd\text{-}equivp\text{-}def\ equivp\text{-}reflp\ split: split\text{-}if)$

lemma $fun\text{-}upd\text{-}equivp\text{-}apply$ $[simp]$:
  $(fun\text{-}upd\text{-}equivp\ (op =)\ f\ x\ y)\ z = (if\ z = x\ then\ y\ else\ f\ z)$
  by $(simp\ only: fun\text{-}upd\text{-}equivp\text{-}def)$

— This is the generalization of $fun\text{-}upd\text{-}equivp\ op =\ ?f\ ?x\ ?y\ ?z = (if\ ?z = ?x\ then\ ?y\ else\ ?f\ ?z)$ with e given equivalence relation and not only with $op =$

lemma $fun\text{-}upd\text{-}equivp\text{-}apply1$ $[simp]$:
  $equivp\ R \Longrightarrow (fun\text{-}upd\text{-}equivp\ R\ f\ x\ y)\ z = (if\ R\ z\ x\ then\ y\ else\ f\ z)$
  by $(simp\ add: fun\text{-}upd\text{-}equivp\text{-}def)$

lemma $fun\text{-}upd\text{-}equivp\text{-}same$: $(fun\text{-}upd\text{-}equivp\ (op =)\ f\ x\ y)\ x = y$
  by $(simp\ only: fun\text{-}upd\text{-}equivp\text{-}def)simp$

— This is the generalization of $fun\text{-}upd\text{-}equivp\ op =\ ?f\ ?x\ ?y\ ?x = ?y$ with a given equivalence relation

lemma $fun\text{-}upd\text{-}equivp\text{-}same1$: $equivp\ R \Longrightarrow (fun\text{-}upd\text{-}equivp\ R\ f\ x\ y)\ x = y$
  by $(simp\ add: fun\text{-}upd\text{-}equivp\text{-}def\ equivp\text{-}reflp)$

For the special case that @term eq is just the equality @term "op =", sharing update and classical update are identical.

lemma $fun\text{-}upd\text{-}equivp\text{-}vs\text{-}fun\text{-}upd$: $(fun\text{-}upd\text{-}equivp\ (op =)) = fun\text{-}upd$
  by$(rule\ ext,\ rule\ ext,\ rule\ ext, simp\ add: fun\text{-}upd\text{-}def\ fun\text{-}upd\text{-}equivp\text{-}def)$

4.9.2 Definition of the shared-memory type

typedef $('\alpha,\ '\beta)\ memory = \{(\sigma::'\alpha \rightharpoonup '\beta,\ R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
proof
  show $(Map.empty,\ (op =)) \in ?memory$
    by $(auto\ simp: identity\text{-}equivp)$
qed

fun $memory\text{-}inv :: ('a \Rightarrow 'b\ option) \times ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $memory\text{-}inv\ (Pair\ f\ R) = (equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow f\ x = f\ y))$

lemma $Abs\text{-}Rep\text{-}memory$ $[simp]$: $Abs\text{-}memory\ (Rep\text{-}memory\ \sigma) = \sigma$
  by $(simp\ add: Rep\text{-}memory\text{-}inverse)$

lemma *memory-invariant* [*simp*]:
  memory-inv σ-rep = (*Rep-memory* (*Abs-memory σ-rep*) = *σ-rep*)
 using *Rep-memory* [*of Abs-memory σ-rep*] *Abs-memory-inverse mem-Collect-eq*
   prod-caseE prod-caseI2 memory-inv.simps
 by *smt*

lemma *Pair-code-eq* :
 Rep-memory σ = *Pair* (*fst* (*Rep-memory σ*)) (*snd* (*Rep-memory σ*))
 by (*simp add*: *Product-Type.surjective-pairing*)

lemma *snd-memory-equivp* [*simp*]: *equivp*(*snd*(*Rep-memory σ*))
 by(*insert Rep-memory*[*of σ*], *auto*)

4.9.3 Operations on Shared-Memory

definition *init* :: ($'\alpha$, $'\beta$) *memory*
where *init* = *Abs-memory* (*Map.empty*, *op* =)

definition *init-mem-list* :: $'\alpha$ *list* ⇒ (*nat*, $'\alpha$) *memory*
where *init-mem-list s* = *Abs-memory* (*let h* = *zip* (*map nat* [*0* .. *int*(*length s*)]) *s*
    in foldl (λ*x* (*y*,*z*). *fun-upd x y* (*Some z*))
     Map.empty h,
    op =)

— Some execution examples for memory construction
value *init*::(*nat*,*int*)*memory*
value *init-mem-list* [−*22*,*2*,−*3*]
value *map* (λ*x*. *the* (*fst* (*Rep-memory init*)*x*)) [*1* .. *10*]
value *take* (*10*) (*map* (*Pair Map.empty*) [(*op* =)])
value *replicate 10 init*
term *Rep-memory σ*
term [(σ::*nat* ⇀ *int*, *R*)<−*xs* . *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)]

Memory Read Operation

definition *lookup* :: ($'\alpha$, $'\beta$) *memory* ⇒ $'\alpha$ ⇒ $'\beta$ (**infixl** $ *100*)
where σ $ *x* = *the* (*fst* (*Rep-memory σ*) *x*)

setup-lifting *type-definition-memory*

Memory Update Operation

fun *Pair-upd-lifter*:: ($'a$ ⇒ $'b$ *option*) × ($'a$ ⇒ $'a$ ⇒ *bool*) ⇒ $'a$ ⇒ $'b$ ⇒
   ($'a$ ⇒ $'b$ *option*) × ($'a$ ⇒ $'a$ ⇒ *bool*)
 where *Pair-upd-lifter* (*Pair f R*) *x y* = (*fun-upd-equivp R f x* (*Some y*), *R*)

lemma *update_sound*$'$:
 assumes σ ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
 shows *Pair-upd-lifter σ x y* ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
proof –
 obtain *mem* **and** *R*
  where *Pair*: (*mem*, *R*) = σ **and** *Eq*: *equivp R* **and** *Mem*: ∀ *x y* . *R x y* ⟶ *mem x* = *mem y*
   using *assms equivpE* **by** *auto*
 obtain *mem*$'$ **and** *R*$'$
  where *Pair*$'$: (*mem*$'$, *R*$'$) = *Pair-upd-lifter σ x y*
   using *surjective-pairing* **by** *metis*

have *Def1*: $mem' = fun\text{-}upd\text{-}equivp\ R\ mem\ x\ (Some\ y)$
and *Def2*: $R' = R$
  using *Pair Pair'* **by** *auto*
have *Eq'*: *equivp R'*
  using *Def2 Eq* **by** *auto*
moreover have $\forall\ y\ z\ .\ R'\ y\ z \longrightarrow mem'\ y = mem'\ z$
  using *Mem equivp-symp equivp-transp*
    unfolding *Def1 Def2* **by** (*metis Eq fun-upd-equivp-def*)
ultimately show *?thesis*
  using *Pair'* **by** *auto*
qed

lift-definition *update* :: $('a,\ 'b)\ memory \Rightarrow 'a \Rightarrow\ 'b \Rightarrow\ ('a,\ 'b)\ memory$ (- '(- $:=_{\$}$ -') *100*)
  is *Pair-upd-lifter*
  using *update_sound'*
  by *simp*

lemma *update'*: $\sigma\ (x :=_{\$}\ y) = Abs\text{-}memory\ (fun\text{-}upd\text{-}equivp\ (snd\ (Rep\text{-}memory\ \sigma))$
$(fst\ (Rep\text{-}memory\ \sigma))\ x\ (Some\ y),\ (snd\ (Rep\text{-}memory\ \sigma)))$
  using *Rep-memory-inverse surjective-pairing Pair-upd-lifter.simps update.rep-eq*
  by *metis*

fun *update-list-rep* :: $(nat \rightharpoonup 'b) \times (nat \Rightarrow nat \Rightarrow bool) \Rightarrow (nat \times 'b\ )list \Rightarrow$
$(nat \rightharpoonup 'b) \times (nat \Rightarrow nat \Rightarrow bool)$
where
  update-list-rep $(f,\ R)$ *nlist* =
  (*foldl* $(\lambda(f,\ R)\ (addr, val).\ Pair\text{-}upd\text{-}lifter\ (f,\ R)\ addr\ val)\ (f,\ R)\ nlist)$

lemma *update-list-rep-p*:
  assumes *1*: $P\ \sigma$
  and 2: $\bigwedge src\ dst\ \sigma.\ P\ \sigma \Longrightarrow P\ (Pair\text{-}upd\text{-}lifter\ \sigma\ src\ dst)$
  shows $P\ (update\text{-}list\text{-}rep\ \ \sigma\ list)$
  using *1 2*
  apply (*induct list arbitrary*: σ)
  apply *force*
  apply *safe*
  apply (*simp del*: *Pair-upd-lifter.simps*)
  using *surjective-pairing*
  apply *metis*
done

lemma *update-list-rep-sound*:
  assumes *1*: $\sigma \in \{(\sigma,\ R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
  shows $update\text{-}list\text{-}rep\ \ \sigma\ (nlist) \in \{(\sigma,\ R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
  using *1*
  apply (*elim update-list-rep-p*)
  apply (*erule update_sound'*)
done

lift-definition *update-list* :: $(nat,\ '\alpha)\ memory \Rightarrow (nat \times '\alpha\ )list \Rightarrow (nat,\ '\alpha)\ memory$
  is *update-list-rep*
  using *update-list-rep-sound* **by** *simp*

Type-invariant:

lemma *update_sound*:
  assumes $Rep\text{-}memory\ \sigma = (\sigma',\ eq)$

shows $(fun\text{-}upd\text{-}equivp\ eq\ \sigma'\ x\ (Some\ y), eq) \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
using *assms insert Rep-memory[of σ]*
apply(*auto simp: fun-upd-equivp-def*)
apply(*rename-tac xa xb, erule contrapos-np*)
apply(*rule-tac R=eq* **and** *y=xa* **in** *equivp-transp,simp*)
apply(*erule equivp-symp, simp-all*)
apply(*rename-tac xa xb, erule contrapos-np*)
apply(*rule-tac R=eq* **and** *y=xb* **in** *equivp-transp,simp-all*)
done

Memory Transfer Based on Sharing Transformation

fun *transfer-rep* ::
$('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool)$
where *transfer-rep* $(m, r)\ src\ dst =$
$(m\ o\ (id\ (dst := src)), (\lambda\ x\ y\ .\ r\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y)))$

lemma *transfer-rep-sound*:
assumes $\sigma \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
shows $transfer\text{-}rep\ \ \sigma\ src\ dst \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
proof –
obtain *mem* **and** *R*
where P: $(mem, R) = \sigma$ **and** E: *equivp R* **and** M: $\forall\ x\ y\ .\ R\ x\ y \longrightarrow mem\ x = mem\ y$
using *assms equivpE* **by** *auto*
obtain *mem′* **and** *R′*
where P': $(mem', R') = transfer\text{-}rep\ \sigma\ src\ \ dst$
by (*metis surj-pair*)
have $D1$: $mem' = (mem\ o\ (id\ (dst := src)))$
and $D2$: $R' = (\lambda\ x\ y\ .\ R\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y))$
using $P\ P'$ **by** *auto*
have *equivp R′*
using *E* **unfolding** *D2 equivp-def* **by** *metis*
moreover have $\forall\ y\ z\ .\ R'\ y\ z \longrightarrow mem'\ y = mem'\ z$
using *M* **unfolding** *D1 D2* **by** *auto*
ultimately show *?thesis*
using P' **by** *auto*
qed

lift-definition
add_e :: $('a,'b)memory \Rightarrow 'a \Rightarrow 'a\ \Rightarrow ('a, 'b)memory$ (- ′(- ⋈ -′) [0,111,111]110)
is *transfer-rep*
using *transfer-rep-sound*
by *simp*

fun *share-list-rep* :: $('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \times 'a\ )list \Rightarrow$
$('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool)$
where
share-list-rep (f, R) *nlist* =
$(foldl\ (\lambda(f, R)\ (src,dst).\ transfer\text{-}rep\ (f, R)\ src\ dst)\ (f, R)\ nlist)$

fun *share-list-rep′* :: $('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \times 'a\ )list \Rightarrow$
$('a \rightharpoonup 'b) \times ('a \Rightarrow 'a \Rightarrow bool)$
where
$share\text{-}list\text{-}rep'\ (f, R)\ [] = (f, R)$

| *share-list-rep′* (f, R) $(n\#nlist) =$
  share-list-rep′ (*transfer-rep* (f, R) (*fst* n) (*snd* n)) *nlist*

lemma *share-list-rep′-p*:
  assumes *1*: $P\ \sigma$
  and 2: $\bigwedge src\ dst\ \sigma.\ P\ \sigma \Longrightarrow P\ (transfer\text{-}rep\ \sigma\ src\ dst)$
  shows $P\ (share\text{-}list\text{-}rep'\ \sigma\ list)$
  using *1 2*
  apply (*induct list arbitrary*: σ P)
  apply *force*
  apply *safe*
  apply (*simp del*: *transfer-rep.simps*)
  using *surjective-pairing*
  apply *metis*
done

lemma *foldl-preserve-p*:
  assumes *1*: *P mem*
  and 2: $\bigwedge y\ z\ mem\ .\ P\ mem \Longrightarrow P\ (f\ mem\ y\ z)$
  shows $P\ (foldl\ (\lambda a\ (y, z).\ f\ mem\ y\ z)\ mem\ list)$
  using *1 2*
  apply (*induct list arbitrary*: *f mem* , *auto*)
  apply *metis*
done

lemma *share-list-rep-p*:
  assumes *1*: $P\ \sigma$
  and 2: $\bigwedge src\ dst\ \sigma.\ P\ \sigma \Longrightarrow P\ (transfer\text{-}rep\ \sigma\ src\ dst)$
  shows $P\ (share\text{-}list\text{-}rep\ \ \sigma\ list)$
  using *1 2*
  apply (*induct list arbitrary*: σ)
  apply *force*
  apply *safe*
  apply (*simp del*: *transfer-rep.simps*)
  using *surjective-pairing*
  apply *metis*
done

lemma *share-list-rep-sound*:
  assumes *1*: $\sigma \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
  shows $share\text{-}list\text{-}rep\ \ \sigma\ (nlist) \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
  using *1*
  apply (*elim share-list-rep-p*)
  apply (*erule transfer-rep-sound*)
done

lift-definition *init-share-list* :: $(nat, '\alpha)\ memory \Rightarrow (nat \times nat\ )list \Rightarrow (nat, '\alpha)\ memory$
  is *share-list-rep*
  using *share-list-rep-sound* **by** *simp*

definition *update-buff* :: $('\alpha, '\beta)\ memory \Rightarrow '\alpha\ set \Rightarrow '\beta\ \ set {\Rightarrow} ('\alpha, '\beta) memory$
 where $update\text{-}buff\ \sigma\ X\ Y =$
  $(let\ (mem, eq) = Rep\text{-}memory\ \sigma$
  $in\ \ Abs\text{-}memory\ (fun\text{-}upd\text{-}equivp\ eq\ mem\ (SOME\ x.\ x{\in}X)\ (Some(SOME\ y.\ y\ {\in}Y)), eq))$

definition *reset* :: $('\alpha, '\beta)$ *memory* $\Rightarrow$ $'\alpha$ *set*$\Rightarrow$ $('\alpha, '\beta)$*memory* (- $'$(*reset* -$'$) *100*)
where σ (*reset X*) = (*let* (σ',*eq*) = *Rep-memory* σ;
eq$'$ = λ *a b*. *eq a b* $\vee$ ($\exists x{\in}X$. *eq a x* $\vee$ *eq b x*)
in if X={} *then* σ
else Abs-memory (*fun-upd-equivp eq*$'$ σ' (*SOME x*. $x{\in}X$) *None*, *eq*))

The modification of the underlying equivalence relation on adresses is only defined on very strong conditions — which are fulfilled for the empty memory, but difficult to establish on a non-empty-one. And of course, the given relation must be proven to be an equivalence relation. So, the case is geared towards shared-memory scenarios where the sharing is defined initially once and for all.

definition *update*$_R$ :: $('\alpha, '\beta)$*memory* $\Rightarrow$ $('\alpha \Rightarrow '\alpha \Rightarrow$ *bool*$)$ $\Rightarrow$ $('\alpha, '\beta)$*memory* (- :=$_R$ - *100*)
where σ :=$_R$ $R \equiv$ *Abs-memory* (*fst*(*Rep-memory* σ), *R*)

definition *lookup*$_R$ :: $('\alpha, '\beta)$*memory* $\Rightarrow$ $('\alpha \Rightarrow '\alpha \Rightarrow$ *bool*$)$ ($\$_R$ - *100*)
where $\$_R$ $\sigma \equiv$ (*snd*(*Rep-memory* σ))

lemma *update*$_R$-*comp-lookup*$_R$:
assumes *equiv* : *equivp R*
and *sharing-conform* : $\forall$ *x y*. *R x y* $\longrightarrow$ *fst*(*Rep-memory* σ) *x* = *fst*(*Rep-memory* σ) *y*
shows ($\$_R$ (σ :=$_R$ *R*)) = *R*
unfolding *lookup*$_R$-*def update*$_R$-*def*
by(*subst Abs-memory-inverse*, *simp-all add*: *equiv sharing-conform*)

4.9.4 Sharing Relation Definition

definition *sharing* :: $'a \Rightarrow ('a, 'b)$*memory* $\Rightarrow$ $'a \Rightarrow$ *bool*
((- *shares*()-/ -) [*201*, *0*, *201*] *200*)
where (*x shares*$_\sigma$ *y*) $\equiv$ (*snd*(*Rep-memory* σ) *x y*)

definition *Sharing* :: $'a$ *set* $\Rightarrow$ $('a, 'b)$*memory* $\Rightarrow$ $'a$ *set* $\Rightarrow$ *bool*
((- *Shares*()-/ -) [*201*, *0*, *201*] *200*)
where (*X Shares*$_\sigma$ *Y*) $\equiv$ ($\exists$ $x{\in}X$. $\exists$ $y{\in}Y$. *x shares*$_\sigma$ *y*)

4.9.5 Properties on Sharing Relation

lemma *sharing-charn*[*code-unfold*]:
(*x shares*$_\sigma$ *y*) $\Longrightarrow$ *equivp* (*snd* (*Rep-memory* σ))
using *Rep-memory*[*of* σ]
unfolding *sharing-def*
by *auto*

lemma *sharing-charn1*[*code-unfold*]:
equivp (*snd* (*Rep-memory* σ))
using *Rep-memory*[*of* σ]
unfolding *sharing-def*
by *auto*

lemma *sharing-charn*$'$[*simp*, *code-unfold*]:
assumes *1*: (*x shares*$_\sigma$ *y*)
shows ($\exists$*R*. *equivp R* $\wedge$ *R x y*)
by (*auto simp add*: *sharing-def snd-def equivp-def*)

lemma *sharing-charn2* [*simp*, *code-unfold*]:
shows$\exists$*x y*. (*equivp* (*snd* (*Rep-memory* σ)) $\wedge$ (*snd* (*Rep-memory* σ)) *x y*)
using *sharing-charn1* [*THEN equivp-reflp*]
by (*simp*)*fast*

lemma *sharing-charn5* [*simp*, *code-unfold*]:
assumes *1*: $i \neq k$
shows $\neg(i\ shares_{init}\ k)$
unfolding *sharing-def init-def*
using *1*
by (*auto simp*: *Abs-memory-inverse identity-equivp*)

lemma *sharing-charn6* [*simp*, *code-unfold*]:
assumes *1*: $i \neq k$
shows $\neg(i\ shares_{init\text{-}mem\text{-}list\ S}\ k)$
unfolding *sharing-def init-mem-list-def*
using *1*
by (*auto simp*: *Abs-memory-inverse identity-equivp*)

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is reflexive
lemma *sharing-refl* [*simp*]: $(x\ shares_{\sigma}\ x)$
using *insert Rep-memory*[*of* σ]
by (*auto simp*: *sharing-def elim*: *equivp-reflp*)

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is symetric
lemma *sharing-sym* [*sym*]:
assumes $x\ shares_{\sigma}\ y$
shows $y\ shares_{\sigma}\ x$
using *assms Rep-memory*[*of* σ]
by (*auto simp*: *sharing-def elim*: *equivp-symp*)

lemma *sharing-commute* : $x\ shares_{\sigma}\ y = (y\ shares_{\sigma}\ x)$
by(*auto intro*: *sharing-sym*)

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is transitive

lemma *sharing-trans* [*trans*]:
assumes $x\ shares_{\sigma}\ y$
and $y\ shares_{\sigma}\ z$
shows $x\ shares_{\sigma}\ z$
using *assms insert Rep-memory*[*of* σ]
by(*auto simp*: *sharing-def elim*: *equivp-transp*)

lemma *shares-result*:
assumes $x\ shares_{\sigma}\ y$
shows $fst\ (Rep\text{-}memory\ \sigma)\ x = fst\ (Rep\text{-}memory\ \sigma)\ y$
using *assms*
unfolding *sharing-def*
using *Rep-memory*[*of* σ]
by *auto*

4.9.6 Memory Domain Definition

definition *Domain* :: $('\alpha, '\beta)memory \Rightarrow '\alpha\ set$
where $Domain\ \sigma = dom\ (fst\ (Rep\text{-}memory\ \sigma))$

4.9.7 Properties on Memory Domain

lemma *Domain-charn*:
assumes *1*:$x \in Domain\ \sigma$

shows $\exists\ y.\ Some\ y = fst\ (Rep\text{-}memory\ \sigma)\ x$
using *1*
by(*auto simp*: *Domain-def*)

— This lemma says that if *x* and *y* are quivalent this means that they are in the same set of equivalent classes

lemma *shares-dom* [*code-unfold*, *intro*]:
assumes $x\ shares_\sigma\ y$
shows $(x \in Domain\ \sigma) = (y \in Domain\ \sigma)$
using *insert Rep-memory*[*of* σ] *assms*
by (*auto simp*: *sharing-def Domain-def*)

lemma *Domain-mono*:
assumes *1*: $x \in\ Domain\ \sigma$
and 2: $(x\ shares_\sigma\ y)$
shows $y \in Domain\ \sigma$
using *1 2 Rep-memory*[*of* σ]
by (*auto simp add*: *sharing-def Domain-def*)

4.9.8 Sharing Relation and Memory Update

lemma *sharing-upd*: $x\ shares_{(\sigma(a :=_\$ b))}\ y = x\ shares_\sigma\ y$
using *insert Rep-memory*[*of* σ]
by(*auto simp*: *sharing-def update-def Abs-memory-inverse*[*OF update_sound*])

— this lemma says that if we do an update on an adress *x* all the elements that are equivalent of *x* are updated

lemma *update''*:
$\sigma\ (x :=_\$ y) = Abs\text{-}memory(fun\text{-}upd\text{-}equivp\ (\lambda x\ y.\ x\ shares_\sigma\ y)\ (fst\ (Rep\text{-}memory\ \sigma))\ x\ (Some\ y),$
$snd\ (Rep\text{-}memory\ \sigma))$
unfolding *update-def sharing-def*
by (*metis update' update-def*)

theorem *update-cancel*:
assumes $x\ shares_\sigma\ x'$
shows $\sigma(x :=_\$ y)(x' :=_\$ z) = (\sigma(x' :=_\$ z))$
proof –
have $*$: $(fun\text{-}upd\text{-}equivp(snd(Rep\text{-}memory\ \sigma))(fst(Rep\text{-}memory\ \sigma))\ x\ (Some\ y), snd\ (Rep\text{-}memory\ \sigma))$
$\in \{(\sigma, R).\ equivp\ R \wedge (\forall\, x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
unfolding *fun-upd-equivp-def*
by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
have $**$: $\bigwedge R\ \sigma.\ equivp\ R \Longrightarrow R\ x\ x' \Longrightarrow$
$fun\text{-}upd\text{-}equivp\ R\ (fun\text{-}upd\text{-}equivp\ R\ \sigma\ x\ (Some\ y))\ x'\ (Some\ z)$
$= fun\text{-}upd\text{-}equivp\ R\ \sigma\ x'\ (Some\ z)$
unfolding *fun-upd-equivp-def*
apply(*rule ext*)
apply(*case-tac R xa x'*, *auto*)
apply(*erule contrapos-np*, *erule equivp-transp*, *simp-all*)
done
show *?thesis*
apply(*simp add*: *update'*)
apply(*insert sharing-charn*[*OF assms*] *assms*[*simplified sharing-def*])
apply(*simp add*: *Abs-memory-inverse* [*OF* $*$] $**$)
done
qed

theorem *update-commute*:
  assumes *1*:$\neg$ (x *shares*$_\sigma$ x')
  shows $(\sigma(x :=_\$ y))(x' :=_\$ z) = (\sigma(x' :=_\$ z)(x :=_\$ y))$
    proof –
    have $*$: $\bigwedge x\, y$.(*fun-upd-equivp*(*snd*(*Rep-memory* σ))(*fst*(*Rep-memory* σ)) x (*Some* y),*snd* (*Rep-memory* σ))
        $\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\, y.\ R\, x\, y \longrightarrow \sigma\, x = \sigma\, y)\}$
        unfolding *fun-upd-equivp-def*
        by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
    have $**$: $\bigwedge R\, \sigma.\ equivp\ R \Longrightarrow \neg R\, x\, x' \Longrightarrow$
        fun-upd-equivp R (*fun-upd-equivp* $R\ \sigma\ x$ (*Some* y)) x' (*Some* z) =
        fun-upd-equivp R (*fun-upd-equivp* $R\ \sigma\ x'$ (*Some* z)) x (*Some* y)
        unfolding *fun-upd-equivp-def*
        apply(*rule ext*)
        apply(*case-tac R xa* x', *auto*)
        apply(*erule contrapos-np*)
        apply(*frule equivp-transp*, *simp-all*)
        apply(*erule equivp-symp*, *simp-all*)
        done
    show *?thesis*
    apply(*simp add*: *update*$'$)
    apply(*insert assms*[*simplified sharing-def*])
    apply(*simp add*: *Abs-memory-inverse* [*OF* $*$] $**$)
  done
qed

4.9.9 Properties on lookup and update wrt the Sharing Relation

lemma *update-triv*:
  assumes *1*: x *shares*$_\sigma$ y
    and *2*: $y \in$ *Domain* σ
  shows $\sigma\ (x :=_\$ (\sigma\ \$\ y)) = \sigma$
proof –
  {
    fix z
    assume *zx*: z *shares*$_\sigma$ x
    then have *zy*: z *shares*$_\sigma$ y
      using *1* **by** (*rule sharing-trans*)
    have *F*: $y \in$ *Domain* $\sigma \Longrightarrow x$ *shares*$_\sigma$ y
        $\Longrightarrow$ *Some* (*the* (*fst* (*Rep-memory* σ) x)) = *fst* (*Rep-memory* σ) y
      by(*auto simp*: *Domain-def dest*: *shares-result*)
    have *Some* (*the* (*fst* (*Rep-memory* σ) y)) = *fst* (*Rep-memory* σ) z
      using *zx* **and** *shares-result* [*OF zy*] *shares-result* [*OF zx*]
      using *F* [*OF 2 1*]
      by *simp*
  } **note** *3* = *this*
  show *?thesis*
    unfolding *update*$''$ *lookup-def fun-upd-equivp-def*
    by (*simp add*: *3 Rep-memory-inverse if-cong*)
qed

lemma *update-idem* :
  assumes *1*: x *shares*$_\sigma$ y
  and *2*: $x \in$ *Domain* σ
  and *3*: $\sigma\ \$\ x = z$
  shows $\sigma(x :=_\$ z) = \sigma$
proof –
  have $*$: $y \in$ *Domain* σ **by**(*simp add*: *shares-dom*[*OF 1*, *symmetric*] *2*)

have σ $(x :=_\$ (\sigma \$ y)) = \sigma$
using *1 2* $*$ **by** (*simp add*: *update-triv*)
also have $(\sigma \$ y) = \sigma \$ x$
by (*simp only*: *lookup-def shares-result* [*OF 1*])
also note *3*
finally show *?thesis* **.**
qed

lemma *update-apply*: $(\sigma(x :=_\$ y)) \$ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma \$ z)$
proof –
have $*$: $(\lambda z.\ if\ z\ shares_\sigma\ x\ then\ Some\ y\ else\ fst\ (Rep\text{-}memory\ \sigma)\ z,\ snd\ (Rep\text{-}memory\ \sigma))$
$\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
unfolding *sharing-def*
by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
show *?thesis*
proof (*cases z shares*$_\sigma$ *x*)
case *True*
assume *A*: *z shares*$_\sigma$ *x*
show σ $(x :=_\$ y) \$ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma \$ z)$
unfolding *update″ lookup-def fun-upd-equivp-def*
by(*simp add*: *Abs-memory-inverse* [*OF* $*$])
next
case *False*
assume *A*: $\neg$ *z shares*$_\sigma$ *x*
show σ $(x :=_\$ y) \$ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma \$ z)$
unfolding *update″ lookup-def fun-upd-equivp-def*
by(*simp add*: *Abs-memory-inverse* [*OF* $*$])
qed
qed

lemma *update-share*:
assumes *z shares*$_\sigma$ *x*
shows $\sigma(x :=_\$ a) \$ z = a$
using *assms*
by (*simp only*: *update-apply if-True*)

lemma *update-other*:
assumes $\neg$(*z shares*$_\sigma$ *x*)
shows $\sigma(x :=_\$ a) \$ z = \sigma \$ z$
using *assms*
by (*simp only*: *update-apply if-False*)

lemma *lookup-update-rep*:
assumes *1*: (*snd* (*Rep-memory* σ')) *x y*
shows (*fst* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)) *x* =
(*fst* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)) *y*
using *1 shares-result sharing-def sharing-upd update.rep-eq*
by (*metis* (*hide-lams*, *no-types*))

lemma *lookup-update-rep″*:
assumes *1*: *x shares*$_\sigma$ *y*
shows $(\sigma\ (src :=_\$ dst)) \$ x = (\sigma\ (src :=_\$ dst)) \$ y$
using *1 lookup-def lookup-update-rep sharing-def update.rep-eq*
by *metis*

4.9.10 Symbolic Execution rules on Memory Update

lemma *mem-update-E*:

assumes *1*: σ = *Rep-memory*(*update* σ' *x y*)
and *2*: σ = *Pair-upd-lifter* (*Rep-memory* σ') *x y* $\Longrightarrow$
equivp (*snd* σ) $\Longrightarrow$ *snd* σ = (*snd*(*Rep-memory* σ')) $\Longrightarrow$ *Q*
shows *Q*
using *1*
unfolding *update.rep-eq*
using *Rep-memory* [*of* (*update* σ' *x y*)] *sharing-charn2 1 2 Pair-upd-lifter.elims snd-conv*
by (*metis* (*hide-lams*, *no-types*))

lemma *Pair-upd-lifter-E*:
assumes *1*: σ = *Pair-upd-lifter* (*Rep-memory* σ') *x y*
and *2*: σ = ((λ*z*. *if* (*snd*(*Rep-memory* σ')) *z x then Some y else* ((*fst*(*Rep-memory* σ'))) *z*),
(*snd*(*Rep-memory* σ'))) $\Longrightarrow$ *Q*
shows *Q*
proof –
obtain *f* **and** *R*
where *sig*: (*f*, *R*) = σ **and**
f : *fst* σ = *f* **and**
R : *snd* σ = *R*
using *surjective-pairing*[*of* σ] **by** *force*
have *obvf1*: *fst* σ = *fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*)
using *1 surjective-pairing*[*of* (*Rep-memory* σ')] *Pair-upd-lifter.simps fst-conv*
by *metis*
have *obvf2*: *f* = *fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*)
using *f obvf1* **by** *simp*
have *obvR1*: *snd* σ = (*snd*(*Rep-memory* σ'))
using *1 surjective-pairing*[*of* (*Rep-memory* σ')] *Pair-upd-lifter.simps snd-conv*
by *metis*
have *obvR2*: *R* = (*snd*(*Rep-memory* σ'))
using *R obvR1* **by** *simp*
have *obvfR*: (*f*, *R*) = (*fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*),
(*snd*(*Rep-memory* σ')))
using *obvf2 obvR2* **by** *simp*
have *obvsig*: σ = (*fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*),
(*snd*(*Rep-memory* σ')))
using *sig obvfR* **by** *simp*
show *?thesis*
using *obvsig*
unfolding *fun-upd-equivp-def*
by (*elim 2*)
qed

lemma *Pair-upd-lifter-rep*:
Pair-upd-lifter (*Rep-memory* σ') *x y* =
(*fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*), (*snd*(*Rep-memory* σ')))
using *surjective-pairing*[*of* (*Rep-memory* σ')] *Pair-upd-lifter.simps*
by *metis*

lemma *Pair-upd-lifter-fst*:
assumes *1*: σ = *Pair-upd-lifter* (*Rep-memory* σ') *x y*
shows *fst* σ = (λ*z*. *if* (*snd*(*Rep-memory* σ')) *z x then Some y else* ((*fst*(*Rep-memory* σ'))) *z*)
proof –
have *obv1*: *fst* σ =
fun-upd-equivp (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*)
using *1* **unfolding** *Pair-upd-lifter-rep* **by** *simp*
also have *obv2*: *fun-upd-equivp* (*snd*(*Rep-memory* σ')) (*fst*(*Rep-memory* σ')) *x* (*Some y*) =
(λ*z*. *if* (*snd*(*Rep-memory* σ')) *z x then Some y else* ((*fst*(*Rep-memory* σ'))) *z*)
unfolding *fun-upd-equivp-def* **by** *simp*

ultimately show *?thesis*
by *simp*
qed

lemma *Pair-upd-lifter-fst1*:
assumes *1*: $\sigma = \textit{Pair-upd-lifter}\ (\textit{Rep-memory}\ \ \sigma')\ x\ y$
shows $(\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow \textit{fst}\ \sigma\ z = \ \textit{Some}\ y$
using *1* **unfolding** *Pair-upd-lifter-rep*
by *simp*

lemma *Pair-upd-lifter-fst2*:
assumes *1*: $\sigma = \textit{Pair-upd-lifter}\ (\textit{Rep-memory}\ \ \sigma')\ x\ y$
shows $\neg(\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow \textit{fst}\ \sigma\ z = (\textit{fst}(\textit{Rep-memory}\ \sigma'))\ z$
using *1* **unfolding** *Pair-upd-lifter-rep*
by *simp*

lemma *Pair-upd-lifter-snd*:
assumes *1*: $\sigma = \textit{Pair-upd-lifter}\ (\textit{Rep-memory}\ \ \sigma')\ x\ y$
shows $\textit{snd}\ \sigma = (\textit{snd}(\textit{Rep-memory}\ \sigma'))$
using *1* **unfolding** *Pair-upd-lifter-rep*
by *simp*

lemma *Pair-upd-lifter-E'*:
assumes *1*: $\sigma = \textit{Pair-upd-lifter}\ \ (\textit{Rep-memory}\ \ \sigma')\ x\ y$
and *2*: $\bigwedge z.\ (\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow$
$\textit{fst}\ (\textit{Pair-upd-lifter}\ \ (\textit{Rep-memory}\ \ \sigma')\ x\ y)\ z = \textit{Some}\ y \Longrightarrow Q$
and *3*: $\bigwedge z.\ \neg(\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow \textit{fst}\ (\ \textit{Pair-upd-lifter}\ \ (\textit{Rep-memory}\ \ \sigma')\ x\ y)\ z =$
$(\textit{fst}(\textit{Rep-memory}\ \sigma'))\ z \Longrightarrow Q$
shows Q
using *assms Pair-upd-lifter-fst1 Pair-upd-lifter-fst2*
unfolding *Pair-upd-lifter-rep*
by *force*

lemma *mem-update-E'*:
assumes *1*: $\sigma = \textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)$
and *2*: $\bigwedge z.\ (\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow \textit{fst}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))\ z = \ \textit{Some}\ y \Longrightarrow$
$\textit{equivp}\ (\textit{snd}\ \sigma) \Longrightarrow \textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)) = (\textit{snd}(\textit{Rep-memory}\ \sigma')) \Longrightarrow Q$
and *3*: $\bigwedge z.\ \neg(\textit{snd}(\textit{Rep-memory}\ \sigma'))\ z\ x \Longrightarrow$
$\textit{fst}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))\ z = (\textit{fst}(\textit{Rep-memory}\ \sigma'))\ z \Longrightarrow$
$\textit{equivp}\ (\textit{snd}\ \sigma) \Longrightarrow \textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)) = (\textit{snd}(\textit{Rep-memory}\ \sigma')) \Longrightarrow Q$
shows Q
using *assms mem-update-E Pair-upd-lifter-fst1 Pair-upd-lifter-fst2 update.rep-eq*
by *metis*

lemma *mem-update-E''*:
assumes *1*: $\sigma = \textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)$
and *2*: $\bigwedge z.\ z\ \textit{shares}_{\sigma'}\ x \Longrightarrow \textit{fst}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))\ z = \ \textit{Some}\ y \Longrightarrow$
$\textit{equivp}\ (\textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))) \Longrightarrow$
$\textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)) = (\textit{snd}(\textit{Rep-memory}\ \sigma')) \Longrightarrow Q$
and *3*: $\bigwedge z.\ \neg(z\ \textit{shares}_{\sigma'}\ x) \Longrightarrow$
$\textit{fst}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))\ z = (\textit{fst}(\textit{Rep-memory}\ \sigma'))\ z \Longrightarrow$
$\textit{equivp}\ (\textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y))) \Longrightarrow$
$\textit{snd}\ (\textit{Rep-memory}(\textit{update}\ \sigma'\ x\ y)) = (\textit{snd}(\textit{Rep-memory}\ \sigma')) \Longrightarrow Q$
shows Q
using *assms*
unfolding *sharing-def*

by (*elim mem-update-E'*, *simp-all*)

lemma *mem-update-lookup-E*:
  assumes *1*: $\sigma = Rep\text{-}memory(update\ \sigma'\ x\ y)$
  and 2: $\bigwedge z.\ z\ shares_{\sigma'}\ x \Longrightarrow (\sigma'\ (x :=_{\$}\ y))\ \$\ z = y \Longrightarrow$
  $equivp\ (snd\ (Rep\text{-}memory(update\ \sigma'\ x\ y))) \Longrightarrow$
  $(x\ shares_{(update\ \sigma'\ x\ y)}\ z) = (x\ shares_{\sigma'}\ z) \Longrightarrow Q$
  and *3*: $\bigwedge z.\ \neg(z\ shares_{\sigma'}\ x) \Longrightarrow$
  $(\sigma'\ (x :=_{\$}\ y))\ \$\ z = \ \sigma'\ \$\ z \Longrightarrow$
  $equivp\ (snd\ (Rep\text{-}memory(update\ \sigma'\ x\ y))) \Longrightarrow$
  $(x\ shares_{(update\ \sigma'\ x\ y)}\ z) = (x\ shares_{\sigma'}\ z) \Longrightarrow Q$
  shows Q
  using *assms*
  by (*metis mem-update-E sharing-refl update-share*)

lemma *Pair-update-rep-inv-E*:
  assumes *1* : $\sigma \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
  and 2 : $memory\text{-}inv\ (Pair\text{-}upd\text{-}lifter\ \sigma\ src\ dst) \Longrightarrow Q$
  shows Q
  using *assms update_sound'*[*of* σ]
  by (*auto simp*: *Abs-memory-inverse*)

lemma *equivp-update-rep*: $equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))$
  using *Rep-memory* [*of* σ'] *transfer-rep-sound* [*of* (*Rep-memory* σ')]
  apply (*erule-tac src= src* **and** *dst = dst* **in** *Pair-update-rep-inv-E*)
  apply (*rotate-tac* 2)
  apply (*subst* (*asm*) *surjective-pairing*[*of* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)])
  unfolding *memory-inv.simps*
  apply (*erule conjE*)
  apply *assumption*
done

lemma *foldl-update-rep-exI*:
  assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n\ \#\ nlist)$
  shows $\exists \sigma''.\ \ \sigma'' = Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n) \wedge$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist)$
  using *1* **unfolding** *foldl.simps Product-Type.split-beta*
  by (*fold surjective-pairing*[*of* (*Rep-memory* σ')], *blast*)

lemma *foldl-update-rep-E*:
  assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n\ \#\ nlist)$
  and 2: $\bigwedge\sigma''.\ \sigma'' = Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n) \Longrightarrow$
  $equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow Q$
  shows Q
proof –
  have *foldl-exec*: $\sigma\ \ =$
  $foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist)$
  using *1* **unfolding** *foldl.simps Product-Type.split-beta*
  by (*fold surjective-pairing*[*of* (*Rep-memory* σ')])

also have *equivp-upd-lifter'*: *equivp* (*snd* (*Pair-upd-lifter* (*Rep-memory* σ') (*fst n*) (*snd n*)))
  using *equivp-update-rep* **.**
ultimately show *?thesis*
using *2 foldl-exec foldl-update-rep-exI* **by** *blast*
qed

lemma *foldl-update-rep-E'*:
  assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n \# nlist)$
  and 2: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
  $(snd(Rep\text{-}memory\ \sigma'))\ z\ (fst\ n) \Longrightarrow$
  $(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ z = \ Some\ (snd\ n) \Longrightarrow Q$
  and 3: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
  $\neg(snd(Rep\text{-}memory\ \sigma'))\ z\ (fst\ n) \Longrightarrow$
  $(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ z =$
  $(fst(Rep\text{-}memory\ \sigma'))\ z \Longrightarrow Q$
  shows *Q*
  using *1*
  apply (*elim foldl-update-rep-E*)
  apply (*erule Pair-upd-lifter-E'*)
  apply (*rule 2*)
  apply *assumption+*
  apply (*erule 3*)
  apply *assumption+*
done

lemma *lookup-update-rep'*:
  assumes *1*: $x\ shares_{\sigma'}\ y$
  shows $(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x =$
  $(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ y$
  using *1 Rep-memory* [*of* σ'] *transfer-rep-sound* [*of* (*Rep-memory* σ')]
  unfolding *sharing-def*
  apply (*erule-tac src= src* **and** *dst = dst* **in** *Pair-update-rep-inv-E*)
  apply (*rotate-tac* 2)
  apply (*subst* (*asm*) *surjective-pairing*[*of* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)])
  unfolding *memory-inv.simps*
  apply (*erule conjE*)
  apply (*erule allE*)+
  apply (*erule impE*)
  unfolding *Pair-upd-lifter-rep*
  apply *simp*
  apply *assumption*
done

lemma *mem-update-list-E*:
  assumes *1*: $\sigma = update\text{-}list\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (n\#nlist)$
  and 2: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n \# nlist) \Longrightarrow Q$
  shows *Q*

using *1*
apply $(subst\ (asm)\ surjective\text{-}pairing[of\ (Rep\text{-}memory\ \ \sigma')])$
unfolding *update-list-rep.simps*
apply $(fold\ surjective\text{-}pairing[of\ (Rep\text{-}memory\ \ \sigma')])$
apply $(elim\ 2)$
done

lemma *mem-update-list-E′*:
assumes 1: $\sigma = Rep\text{-}memory\ (update\text{-}list\ \sigma'\ (n\#nlist))$
and 2: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
$(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$(snd(Rep\text{-}memory\ \sigma'))\ z\ (fst\ n) \Longrightarrow$
$(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ z = \ Some\ (snd\ n) \Longrightarrow Q$
and 3: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
$(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$\neg(snd(Rep\text{-}memory\ \sigma'))\ z\ (fst\ n) \Longrightarrow$
$(fst\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ z =$
$(fst(Rep\text{-}memory\ \sigma'))\ z \Longrightarrow Q$
shows Q
using *1*
unfolding *update-list.rep-eq*
apply $(elim\ mem\text{-}update\text{-}list\text{-}E)$
apply $(erule\ foldl\text{-}update\text{-}rep\text{-}E')$
apply $(erule\ 2)$
apply *assumption+*
apply $(erule\ 3)$
apply *assumption+*
done

lemma *mem-update-list-E′′*:
assumes 1: $\sigma = Rep\text{-}memory\ (update\text{-}list\ \sigma'\ (n\#nlist))$
and 2: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
$(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$z\ shares_{\sigma'}\ (fst\ n) \Longrightarrow$
$(\sigma'(fst\ n :=_{\$}\ snd\ n))\ \$\ z = snd\ n \Longrightarrow Q$
and 3: $\bigwedge z.\ equivp\ (snd\ (Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ Pair\text{-}upd\text{-}lifter\ (f, R)\ x\ y)$
$(Pair\text{-}upd\text{-}lifter\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$\neg(z\ shares_{\sigma'}\ (fst\ n)) \Longrightarrow$
$(\sigma'(fst\ n :=_{\$}\ snd\ n))\ \$\ z = \sigma'\ \$\ z \Longrightarrow Q$
shows Q
using *1*
unfolding *update-list.rep-eq*
apply $(elim\ mem\text{-}update\text{-}list\text{-}E)$
apply $(erule\ foldl\text{-}update\text{-}rep\text{-}E')$
apply $(erule\ 2)$
unfolding *sharing-def update.rep-eq lookup-def*
apply *assumption+*
apply *simp*
apply $(erule\ 3)$
unfolding *sharing-def update.rep-eq lookup-def*
apply *assumption+*
apply *simp*
done

4.9.11 Symbolic Execution Rules On Memory Transfer

lemma *transfer-rep-inv-E*:
assumes *1* : $\sigma \in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
and 2 : *memory-inv* $(transfer\text{-}rep\ \sigma\ src\ dst) \Longrightarrow Q$
shows Q
using *assms transfer-rep-sound*[*of* σ]
by (*auto simp*: *Abs-memory-inverse*)

lemma *transfer-rep-simp*:
$transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst =$
$((fst\ (Rep\text{-}memory\ \ \sigma'))o(id\ (dst := src)),$
$(\lambda\ x\ y\ .\ (snd\ (Rep\text{-}memory\ \ \sigma'))\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y)))$
using *surjective-pairing*[*of* (*Rep-memory* σ')] *transfer-rep.simps*
by *metis*

lemma *transfer-rep-E*:
assumes *1*: $\sigma = transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst$
and 2: $\sigma = ((fst\ (Rep\text{-}memory\ \ \sigma'))o(id\ (dst := src)),$
$(\lambda\ x\ y\ .\ (snd\ (Rep\text{-}memory\ \ \sigma'))\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y))) \Longrightarrow Q$
shows Q
using *1* **unfolding** *transfer-rep-simp*
by (*elim* 2)

lemma *transfer-rep-fst-E*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and 2: $\sigma = (fst\ (Rep\text{-}memory\ \ \sigma'))o(id\ (dst := src)) \Longrightarrow Q$
shows Q
using *1* **unfolding** *transfer-rep-simp fst-conv*
by (*elim* 2)

lemma *transfer-rep-fst1-E*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and 2: $\bigwedge x.\ \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (if\ x{=}dst\ then\ src\ else\ id\ x) \Longrightarrow Q$
shows Q
using *1 2* **unfolding** *transfer-rep-simp fst-conv*
by *simp*

lemma *transfer-rep-fst1*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
shows $\bigwedge x.\ x = dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ src$
using *1* **unfolding** *transfer-rep-simp*
by *simp*

lemma *transfer-rep-fst2*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
shows $\bigwedge x.\ x \neq\ dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)$
using *1* **unfolding** *transfer-rep-simp*
by *simp*

lemma *transfer-rep-fst-E'*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and 2: $\bigwedge x.\ x = dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ src \Longrightarrow Q$
and 3: $\bigwedge x.\ x \neq\ dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow Q$
shows Q
using *assms* **unfolding** *transfer-rep-simp*
by *force*

lemma *transfer-rep-snd-E*:

assumes *1*: $\sigma = snd\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and *2*: $\sigma = (\lambda\, x\, y\, .\ (snd\ (Rep\text{-}memory\ \ \sigma'))\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y)) \Longrightarrow Q$
shows Q
using *1* **unfolding** *transfer-rep-simp snd-conv*
by (*elim 2*)

lemma *transfer-rep-snd1-E*:
assumes *1*: $\sigma = snd(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and *2*: $\bigwedge x\ y.\ \sigma\ x\ y = (snd\ (Rep\text{-}memory\ \ \sigma'))\ (if\ x{=}dst\ then\ src\ else\ id\ x)$
$(if\ y{=}dst\ then\ src\ else\ id\ y) \Longrightarrow Q$
shows Q
using *1 2* **unfolding** *transfer-rep-simp fst-conv*
by *simp*

lemma *transfer-rep-snd-E'*:
assumes *1*: $\sigma = snd(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and *2*: $\bigwedge x\ y.\ x = dst \Longrightarrow y = dst \Longrightarrow \sigma\ x\ y = (snd\ (Rep\text{-}memory\ \ \sigma'))\ src\ src \Longrightarrow Q$
and *3*: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y \neq dst \Longrightarrow \sigma\ x\ y = (snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (id\ y) \Longrightarrow Q$
and *4*: $\bigwedge x\ y.\ x = dst \Longrightarrow y \neq dst \Longrightarrow \sigma\ x\ y = (snd\ (Rep\text{-}memory\ \ \sigma'))\ (src)\ (id\ y) \Longrightarrow Q$
and *5*: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y = dst \Longrightarrow \sigma\ x\ y = (snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (src) \Longrightarrow Q$
shows Q
using *assms* **unfolding** *transfer-rep-simp*
by *force*

lemma *transfer-rep-E'*:
assumes *1*: $\sigma = (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
and *2*: $\bigwedge x\ y.\ x = dst \Longrightarrow y = dst \Longrightarrow$
$(fst\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ src \Longrightarrow$
$(snd\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ src\ src \Longrightarrow Q$
and *3*: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y \neq dst \Longrightarrow$
$(snd\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (id\ y) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow Q$
and *4*: $\bigwedge x\ y.\ x = dst \Longrightarrow y \neq dst \Longrightarrow$
$(fst\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ src \Longrightarrow$
$(snd\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (src)\ (id\ y) \Longrightarrow Q$
and *5*: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y = dst \Longrightarrow$
$(snd\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (src) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow Q$
shows Q
using *assms* **unfolding** *transfer-rep-simp*
by *force*

lemma *lookup-transfer-rep*:
assumes *1*: $(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x\ y$
shows $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ x =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ y$
using *1 Rep-memory* [*of* σ']
apply (*erule-tac src= src* **and** *dst = dst* **in** *transfer-rep-inv-E*)
apply (*rotate-tac 1*)
apply (*subst* (*asm*) *surjective-pairing*[*of* (*transfer-rep* (*Rep-memory* σ') *src dst*)])
unfolding *memory-inv.simps*

apply (*erule conjE*)
apply (*erule allE*)+
apply (*erule impE*)
unfolding *transfer-rep-simp*
apply *simp-all*
done

lemma *lookup-transfer-rep′*:
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ dst$
using *Rep-memory* [*of* σ']
apply (*erule-tac src= src* **and** *dst = dst* **in** *transfer-rep-inv-E*)
apply (*rotate-tac 1*)
apply (*subst* (*asm*) *surjective-pairing*[*of* (*transfer-rep* (*Rep-memory* σ') *src dst*)])
unfolding *memory-inv.simps*
apply (*erule conjE*)
apply (*erule allE*)+
apply (*erule impE*)
unfolding *transfer-rep-simp*
apply *auto*
using *equivp-reflp snd-memory-equivp*
apply *metis*
done

lemma *mem-share-list-E*:
assumes *1*: $\sigma = share\text{-}list\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (n\#nlist)$
and 2: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
$(Rep\text{-}memory\ \ \sigma')\ (n\ \#\ nlist) \Longrightarrow Q$
shows *Q*
using *1*
apply (*subst* (*asm*) *surjective-pairing*[*of* (*Rep-memory* σ')])
unfolding *share-list-rep.simps*
apply (*fold surjective-pairing*[*of* (*Rep-memory* σ')])
apply (*elim* 2)
done

lemma *foldl-transfer-E*:
assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
$(Rep\text{-}memory\ \ \sigma')\ (n\ \#\ nlist)$
and 2: $equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
$(transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
shows *Q*
using *1*
unfolding *foldl.simps Product-Type.split-beta*
apply (*fold surjective-pairing*[*of* (*Rep-memory* σ')])
using *Rep-memory* [*of* σ'] *transfer-rep-sound* [*of* (*Rep-memory* σ')]
apply (*erule-tac src= fst n* **and** *dst = snd n* **in** *transfer-rep-inv-E*)
apply (*rotate-tac* 2)
apply (*subst* (*asm*) *surjective-pairing*[*of* (*transfer-rep* (*Rep-memory* σ') (*fst n*) (*snd n*))])
unfolding *memory-inv.simps*
apply (*erule conjE*)
apply (*erule* 2)
apply *assumption*
apply (*rule lookup-transfer-rep′*)
done

lemma *foldl-transfer-rep-exI*:
 assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n \# nlist)$
 shows $\exists \sigma''.\ \sigma'' = transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n) \wedge$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
   $(transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist)$
 using *1* **unfolding** *foldl.simps Product-Type.split-beta*
 by (*fold surjective-pairing*[*of* (*Rep-memory* σ')], *blast*)

lemma *equivp-transfer-rep*: *equivp* (*snd* (*transfer-rep* (*Rep-memory* σ') *src dst*))
 using *Rep-memory* [*of* σ']
 apply (*erule-tac src= src* **and** *dst = dst* **in** *transfer-rep-inv-E*)
 apply (*subst* (*asm*) *surjective-pairing*[*of* (*transfer-rep* (*Rep-memory* σ') *src dst*)])
 unfolding *memory-inv.simps*
 apply (*erule conjE*)
 apply *assumption*
done

lemma *foldl-transfer-rep-E*:
 assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n \# nlist)$
 and 2: $\bigwedge\sigma''.\ \sigma'' = transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n) \Longrightarrow$
  $equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
   $(transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow Q$
 shows Q
proof –
 have *foldl-exec*: $\sigma\ =$
   $foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
    $(transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist)$
  using *1* **unfolding** *foldl.simps Product-Type.split-beta*
  by (*fold surjective-pairing*[*of* (*Rep-memory* σ')])
 also have *equivp-upd-lifter'*: *equivp* (*snd* (*transfer-rep* (*Rep-memory* σ') (*fst n*) (*snd n*)))
  using *equivp-transfer-rep* **.**
 ultimately show *?thesis*
 using *2 foldl-exec* **by** *fast*
qed

lemma *foldl-transfer-E'*:
 assumes *1*: $\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
  $(Rep\text{-}memory\ \ \sigma')\ (n \# nlist)$
 and 2: $\bigwedge x\ y.\ x = (snd\ n) \Longrightarrow y = (snd\ n) \Longrightarrow$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
  $(fst\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n) \Longrightarrow$
  $(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
  $(snd\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n)\ (fst\ n) \Longrightarrow$
  $equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
 and 3: $\bigwedge x\ y.\ x \neq (snd\ n) \Longrightarrow y \neq (snd\ n) \Longrightarrow$
  $(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
  $(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (id\ y) \Longrightarrow$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
  $(fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow$
  $equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
  $(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$

and 4: $\bigwedge x\, y.\ x = (snd\ n) \Longrightarrow y \neq (snd\ n) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n)\ (id\ y) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
and 5: $\bigwedge x\, y.\ x \neq (snd\ n) \Longrightarrow y = (snd\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (fst\ n) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
shows Q
using *1*
apply (*elim foldl-transfer-rep-E*)
apply (*erule transfer-rep-E′*)
apply (*erule 2*)
apply *assumption+*
apply (*rule lookup-transfer-rep′*)
apply (*erule 3*)
apply *assumption+*
apply (*rule lookup-transfer-rep′*)
apply (*erule 4*)
apply *assumption+*
apply (*rule lookup-transfer-rep′*)
apply (*erule 5*)
apply *assumption+*
apply (*rule lookup-transfer-rep′*)
done

lemma *mem-init-share-list-E′*:
assumes 1: $\sigma = Rep\text{-}memory\ (init\text{-}share\text{-}list\ \sigma'\ (n\#nlist))$
and 2: $\bigwedge x\, y.\ x = (snd\ n) \Longrightarrow y = (snd\ n) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n)\ (fst\ n) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
and 3: $\bigwedge x\, y.\ x \neq (snd\ n) \Longrightarrow y \neq (snd\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (id\ y) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
and 4: $\bigwedge x\, y.\ x = (snd\ n) \Longrightarrow y \neq (snd\ n) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (fst\ n)\ (id\ y) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$

$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
and $5: \bigwedge x\ y.\ x \neq (snd\ n) \Longrightarrow y = (snd\ n) \Longrightarrow$
$(snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x\ y =$
$(snd\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)\ (fst\ n) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ x =$
$(fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x) \Longrightarrow$
$equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
shows Q
using *1*
unfolding *init-share-list.rep-eq*
apply (*elim mem-share-list-E*)
apply (*erule foldl-transfer-E'*)
apply (*erule 2*)
apply *assumption+*
apply (*erule 3*)
apply *assumption+*
apply (*erule 4*)
apply *assumption+*
apply (*erule 5*)
apply *assumption+*
done

lemma *mem-init-share-list-E''*:
assumes $1: \sigma = Rep\text{-}memory\ (init\text{-}share\text{-}list\ \sigma'\ (n\#nlist))$
and $2: equivp\ (snd\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))) \Longrightarrow$
$\sigma = foldl\ (\lambda(f, R)\ (x, y).\ transfer\text{-}rep\ (f, R)\ x\ y)$
$(transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n))\ (nlist) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (fst\ n) =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ (fst\ n)\ (snd\ n)))\ (snd\ n) \Longrightarrow Q$
shows Q
using *1*
unfolding *init-share-list.rep-eq*
apply (*elim mem-share-list-E*)
apply (*erule foldl-transfer-E*)
apply (*erule 2*)
apply *assumption*
apply (*rule lookup-transfer-rep'*)
done

lemma *Rep-memory-E*:
assumes $1: (\sigma = Rep\text{-}memory\ (\sigma'))$
and $2: memory\text{-}inv\ \sigma \Longrightarrow Q$
shows Q
apply (*insert 1*)
apply (*insert Rep-memory*[*of* σ'])
apply *hypsubst*
apply (*insert 1*)
apply (*rotate-tac*)
apply (*subst* (*asm*) *HOL.eq-commute*)
apply (*simp only*:)
apply (*metis 2 Rep-memory-inverse memory-invariant*)
done

lemma *mem-add$_e$-E*:
assumes *1*: $\sigma = Rep\text{-}memory(add_e\ \sigma'\ src\ dst)$
and 2: $\sigma = transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst \Longrightarrow$
$equivp\ (snd\ \sigma) \Longrightarrow$
$snd\ \sigma = (\lambda\ x\ y\ .\ (snd(Rep\text{-}memory\ \sigma'))\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y)) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ dst \Longrightarrow Q$
shows Q
using *1*
unfolding *add$_e$.rep-eq*
apply (*elim* 2)
using *1 2*
apply *simp*
using *1*
unfolding *add$_e$.rep-eq snd-def*
apply (*simp add*: *Product-Type.split-beta del:fun-upd-apply*)
using *snd-conv transfer-rep.elims*
apply *metis*
apply (*rule lookup-transfer-rep$'$*)
done

lemma *mem-add$_e$-E$'$*:
assumes *1*: $\sigma = Rep\text{-}memory(add_e\ \sigma'\ src\ dst)$
and 2: $\bigwedge x\ y.\ x = dst \Longrightarrow y = dst \Longrightarrow (fst\ \sigma)\ x = (fst\ (Rep\text{-}memory\ \sigma'))\ src \Longrightarrow$
$(snd\ \sigma)\ x\ y = (snd\ (Rep\text{-}memory\ \sigma'))\ src\ src \Longrightarrow$
$equivp\ (snd\ \sigma) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ dst \Longrightarrow Q$
and 3: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y \neq dst \Longrightarrow (snd\ \sigma)\ x\ y = (snd\ (Rep\text{-}memory\ \sigma'))\ (id\ x)\ (id\ y) \Longrightarrow$
$(fst\ \sigma)\ x = (fst\ (Rep\text{-}memory\ \sigma'))\ (id\ x) \Longrightarrow$
$equivp\ (snd\ \sigma) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ dst \Longrightarrow Q$
and 4: $\bigwedge x\ y.\ x = dst \Longrightarrow y \neq dst \Longrightarrow (fst\ \sigma)\ x = (fst\ (Rep\text{-}memory\ \sigma'))\ src \Longrightarrow$
$(snd\ \sigma)\ x\ y = (snd\ (Rep\text{-}memory\ \sigma'))\ (src)\ (id\ y) \Longrightarrow$
$equivp\ (snd\ \sigma) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ dst \Longrightarrow Q$
and 5: $\bigwedge x\ y.\ x \neq dst \Longrightarrow y = dst \Longrightarrow (snd\ \sigma)\ x\ y = (snd\ (Rep\text{-}memory\ \sigma'))\ (id\ x)\ (src) \Longrightarrow$
$(fst\ \sigma)\ x = (fst\ (Rep\text{-}memory\ \sigma'))\ (id\ x) \Longrightarrow$
$equivp\ (snd\ \sigma) \Longrightarrow$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \sigma')\ src\ dst))\ dst \Longrightarrow Q$
shows Q
using *1*
apply (*elim mem-add$_e$-E*)
apply (*elim transfer-rep-E$'$*)
apply (*erule* 2)
apply *assumption+*
apply (*metis 1 add$_e$.rep-eq*)
apply (*metis 1 add$_e$.rep-eq*)
apply *metis*
apply *assumption+*
apply (*erule* 3)
apply *assumption+*
apply (*metis 1 add$_e$.rep-eq*)
apply (*metis 1 add$_e$.rep-eq*)
apply *metis*

apply *assumption+*
apply (*erule 4*)
apply *assumption+*
apply (*metis 1* add_e*.rep-eq*)
apply (*metis 1* add_e*.rep-eq*)
apply *metis*
apply *assumption+*
apply (*erule 5*)
apply *assumption+*
apply (*metis 1* add_e*.rep-eq*)
apply (*metis 1* add_e*.rep-eq*)
apply *metis*
apply *assumption+*
done

lemma *mem-init-share-list-E*:
assumes *1*: $\sigma = \textit{Rep-memory}\ (\textit{init-share-list}\ \sigma'\ (n\#\textit{nlist}))$
and 2: $\sigma = \textit{share-list-rep}\ (\textit{Rep-memory}\ \ \sigma')\ (n\#\textit{nlist}) \Longrightarrow Q$
shows Q
using *1*
unfolding *init-share-list.rep-eq*
by (*elim 2*)

lemma *Rep-memory-E''*:
assumes *1* : $\sigma \in \{(\sigma, R).\ \textit{equivp}\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
and 2 : $\textit{memory-inv}\ \sigma \Longrightarrow Q$
shows Q
using *assms*
by (*auto simp*: *Abs-memory-inverse*)

4.9.12 Properties on Memory Transfer

lemma add_e*-share*:$x\ \textit{shares}_{add_e\ \sigma\ x\ y}\ y$
using *fun-upd-apply id-apply mem-*add_e*-E sharing-refl*
unfolding *sharing-def*
by *metis*

lemma add_e*-share-lookup1*:
$(\sigma(x \bowtie y))\ \$\ x = \sigma\ \$\ x$
using *lookup-transfer-rep' transfer-rep-fst1*
unfolding *lookup-def* add_e*.rep-eq*
by *metis*

lemma add_e*-share-lookup2*:
$(\sigma(x \bowtie y))\ \$\ y = \sigma\ \$\ x$
using *transfer-rep-fst1*
unfolding add_e*.rep-eq lookup-def*
by *metis*

lemma add_e*-share-mono*:
assumes *1*: $(x\ \textit{shares}_\sigma\ y)$
and 2: $\neg(x\ \textit{shares}_\sigma\ y')$
shows $(x\ \textit{shares}_{\sigma\ (x' \bowtie y')}\ y)$
using *assms*
unfolding *sharing-def*
using *2 fun-upd-apply id-apply mem-*add_e*-E sharing-refl*
by *metis*

lemma add_e*-share-charn* [*simp*, *code-unfold*]:
assumes *1*: $\neg(i\ \ shares_\sigma\ k')$
and 2: $\neg(k\ shares_\sigma\ k')$
shows $i\ shares_{\sigma(i' \bowtie k')}\ k = i\ shares_\sigma\ k$
using *assms fun-upd-apply id-def mem-add*$_e$*-E sharing-def sharing-refl*
by *metis*

lemma add_e*-share-trans*:
assumes *1*: $(x\ shares_\sigma\ z)$
shows $(z\ shares_{\sigma(x \bowtie y)}\ y)$
using *1*
unfolding *sharing-def*
using add_e*-share* add_e*-share-mono* *mem-add*$_e$*-E*
equivp-reflp equivp-symp fun-upd-def id-def snd-memory-equivp
by *smt*

lemma add_e*-share-rec*:
$x'\ shares_{\sigma(x \bowtie y)}\ y' =$
$((x'\ shares_{\sigma(x \bowtie y)}\ x) \wedge (y'\ shares_{\sigma(x \bowtie y)}\ y)) \vee$
$((x'\ shares_{\sigma(x' \bowtie y)}\ y) \wedge (y'\ shares_{\sigma(y' \bowtie x)}\ x)) \vee$
$(x'\ shares_\sigma\ y')$
by (*metis* add_e*-share*)

lemma add_e*-share-trans'*:
assumes *1*: $(x\ shares_{(\sigma(x \bowtie y))}\ z)$
shows $(y\ shares_{(\sigma(x \bowtie y))}\ z)$
using *1* add_e*-share sharing-sym sharing-trans*
by *fast*

lemma add_e*-share-old-new-trans*:
assumes *1*: $(x\ shares_\sigma\ z)$
shows $(y\ shares_{(\sigma(x \bowtie y))}\ z)$
using *1* add_e*-share-trans sharing-sym*
by *fast*

lemma add_e*-not-share-lookup*:
assumes *1*: $\neg(x\ shares_\sigma\ z)$
and 2: $\neg(y\ shares_\sigma\ z)$
shows $\sigma\ (x \bowtie y)\ \$\ z = \sigma\ \$\ z$
using *assms*
unfolding *sharing-def lookup-def* add_e*.rep-eq*
using *id-def sharing-def sharing-refl transfer-rep-fst2*
by *metis*

lemma add_e*-share-dom*:
assumes *1*: $z \in Domain\ \sigma$
and 2: $\neg(y\ shares_\sigma\ z)$
shows $(\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ z$
using *assms*
unfolding *Domain-def sharing-def lookup-def*
using *2* add_e*.rep-eq id-apply sharing-refl transfer-rep-fst2*
by *metis*

lemma *shares-result'*:
assumes *1*: $(x\ shares_\sigma\ y)$
shows $\sigma\ \$\ x = \sigma\ \$\ y$
using *assms lookup-def shares-result*

by *metis*

lemma add_e*-share-cancel1*:
assumes *1*: $(x\ shares_\sigma\ z)$
shows $(\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ x$
using *1* add_e*.rep-eq* add_e*-share-trans lookup-def lookup-transfer-rep sharing-def transfer-rep-fst1*
by *metis*

4.9.13 Test on Sharing and Transfer via smt ...

lemma $\forall\, x\ y.\ x \neq y \longrightarrow \neg(x\ shares_\sigma\ y) \Longrightarrow$
$\sigma\ \$\ x > \sigma\ \$\ y \Longrightarrow\ \sigma(3 \bowtie (4{::}nat)) = \sigma' \Longrightarrow$
$\sigma'' = (\sigma'(3 :=_\$ ((\sigma'\ \$\ 4) + 2))) \Longrightarrow$
$x \neq 3 \Longrightarrow x \neq 4 \Longrightarrow y \neq 3 \Longrightarrow y \neq 4 \Longrightarrow\ \sigma''\ \$\ x > \sigma''\ \$\ y$
by (*smt* add_e*-not-share-lookup* add_e*-share-charn update-apply*)

4.9.14 Adaptation For the smt Solver

lemma add_e*-share-charn-smt* :
$\neg(i\ \ shares_\sigma\ k') \wedge$
$\neg(k\ shares_\sigma\ k') \longrightarrow$
$i\ shares_{\sigma(i' \bowtie k')}\ k = i\ shares_\sigma\ k$
using add_e*-share-charn*
by *simp*

lemma add_e*-not-share-lookup-smt*:
$\neg(x\ shares_\sigma\ z) \wedge \neg(y\ shares_\sigma\ z) \longrightarrow (\sigma\ (x \bowtie y)\ \$\ z) = (\sigma\ \$\ z)$
using add_e*-not-share-lookup*
by *auto*

lemma add_e*-share-dom-smt*:
$z \in Domain\ \sigma \wedge \neg(y\ shares_\sigma\ z) \longrightarrow (\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ z$
using add_e*-share-dom*
by *auto*

lemma add_e*-share-cancel1-smt*:
$(x\ shares_\sigma\ z) \longrightarrow (\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ x$
using add_e*-share-cancel1*
by *auto*

lemma *lookup-update-rep''-smt*:
$x\ shares_\sigma\ y \longrightarrow (\sigma\ (src :=_\$ dst))\ \$\ x = (\sigma\ (src :=_\$ dst))\ \$\ y$
using *lookup-update-rep''*
by *auto*

theorem *update-commute-smt*:
$\neg\ (x\ shares_\sigma\ x') \longrightarrow ((\sigma(x :=_\$ y))(x' :=_\$ z)) = (\sigma(x' :=_\$ z)(x :=_\$ y))$
using *update-commute*
by *auto*

theorem *update-cancel-smt*:
$(x\ shares_\sigma\ x') \longrightarrow (\sigma(x :=_\$ y)(x' :=_\$ z)) = (\sigma(x' :=_\$ z))$
using *update-cancel*
by *auto*

lemma *update-other-smt*:

$\neg(z\ shares_\sigma\ x) \longrightarrow (\sigma(x :=_\$ a)\ \$\ z) = \sigma\ \$\ z$
using *update-other*
by *auto*

lemma *update-share-smt*:
$(z\ shares_\sigma\ x) \longrightarrow (\sigma(x :=_\$ a)\ \$\ z) = a$
using *update-share*
by *auto*

lemma *update-idem-smt* :
$(x\ shares_\sigma\ y) \wedge x \in Domain\ \sigma \wedge (\sigma\ \$\ x = z) \longrightarrow (\sigma(x{:=_\$}\ z)) = \sigma$
using *update-idem*
by *fast*

lemma *update-triv-smt*:
$(x\ shares_\sigma\ y) \wedge y \in Domain\ \sigma \longrightarrow (\sigma\ (x :=_\$ (\sigma\ \$\ y))) = \sigma$
using *update-triv*
by *auto*

lemma *shares-result-smt*:
$x\ shares_\sigma\ y \longrightarrow \sigma\ \$\ x = \sigma\ \$\ y$
using *shares-result'*
by *fast*

lemma *shares-dom-smt* :
$x\ shares_\sigma\ y \longrightarrow (x \in Domain\ \sigma) = (y \in Domain\ \sigma)$
using *shares-dom*
by *fast*
lemma *sharing-refl-smt* : $(x\ shares_\sigma\ x)$
using *sharing-refl*
by *simp*

lemma *sharing-sym-smt* :
$x\ shares_\sigma\ y \longrightarrow y\ shares_\sigma\ x$
using *sharing-sym*
by (*auto*)

lemma *sharing-commute-smt* : $x\ shares_\sigma\ y = (y\ shares_\sigma\ x)$
by(*auto intro*: *sharing-sym*)

lemma *sharing-trans-smt*:
$x\ shares_\sigma\ y \longrightarrow y\ shares_\sigma\ z \longrightarrow x\ shares_\sigma\ z$
using *sharing-trans*
by(*auto*)

lemma *nat-0-le-smt*: $0 \leq z \longrightarrow int\ (nat\ z) = z$
by *transfer clarsimp*

lemma *nat-le-0-smt*: $0 > z \longrightarrow int\ (nat\ z) = 0$
by *transfer clarsimp*

lemma *update-apply-smt*: $(\sigma(x :=_\$ y))\ \$\ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma\ \$\ z)$
using *update-apply*
by *fast*

lemma *add_e-share-lookup2-smt*:
$(\sigma(x \bowtie y))\ \$\ y = \sigma\ \$\ x$

using add_e-*share-lookup2*
by *fast*

lemma add_e-*share-trans-smt*:
$(x\ shares_{\sigma}\ z) \longrightarrow (z\ shares_{\sigma(x \bowtie y)}\ y)$
using add_e-*share-trans*
by *fast*

lemma add_e-*share-mono-smt*:
$(x\ shares_{\sigma}\ y) \longrightarrow \neg(x\ shares_{\sigma}\ y') \longrightarrow (x\ shares_{\sigma\ (x' \bowtie y')}\ y)$
using add_e-*share-mono*
by *fast*

lemma add_e-*share-lookup1-smt*:
$(\sigma(x \bowtie y))\ \$\ x = \sigma\ \$\ x$
using add_e-*share-lookup1*
by *fast*

lemma add_e-*share-smt*:$x\ shares_{add_e\ \sigma\ x\ y}\ y$
using add_e-*share*
by *fast*

lemma add_e-*share-trans′-smt*:
$(x\ shares_{(\sigma(x \bowtie y))}\ z) \longrightarrow (y\ shares_{(\sigma(x \bowtie y))}\ z)$
using add_e-*share-trans′*
by *fast*

lemma add_e-*share-old-new-trans-smt*:
$(x\ shares_{\sigma}\ z) \longrightarrow (y\ shares_{(\sigma(x \bowtie y))}\ z)$
using add_e-*share-old-new-trans*
by *fast*

lemma *Domain-mono-smt*:
$x \in\ Domain\ \sigma \longrightarrow (x\ shares_{\sigma}\ y) \longrightarrow y \in Domain\ \sigma$
using *Domain-mono*
by *fast*

lemma *sharing-upd-smt*: $x\ shares_{(\sigma(a :=_{\$} b))}\ y = x\ shares_{\sigma}\ y$
using *sharing-upd*
by *fast*

lemma *sharing-charn6-smt* :
$i \neq k \longrightarrow \neg(i\ shares_{init\text{-}mem\text{-}list\ S}\ k)$
using *sharing-charn6*
by *fast*

lemma *mem1-smt*:$(\sigma(a \bowtie b\ )\$\ a) = (\sigma\ (a \bowtie b)\ \$b)$
by (*metis* add_e-*share-lookup1-smt* add_e-*share-lookup2-smt*)

lemma *transfer-rep-fst2-smt*:
$\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst) \longrightarrow$
$x \neq\ dst\ \longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)$
using *transfer-rep-fst2*
by *metis*

lemma *mem-add*$_e$-*E-smt*:

$\sigma = \textit{Rep-memory}(add_e\ \sigma'\ src\ dst) \wedge$
$(\sigma = \textit{transfer-rep}\ (\textit{Rep-memory}\ \sigma')\ src\ dst \wedge$
$\quad \textit{equivp}\ (snd\ \sigma) \wedge$
$\quad snd\ \sigma = (\lambda\, x\, y\, .\, (snd(\textit{Rep-memory}\ \sigma'))\ ((id\ (dst := src))\ x)\ ((id\ (dst := src))\ y)) \wedge$
$\quad (fst\ (\textit{transfer-rep}\ (\textit{Rep-memory}\ \sigma')\ src\ dst))\ src =$
$\quad (fst\ (\textit{transfer-rep}\ (\textit{Rep-memory}\ \sigma')\ src\ dst))\ dst \longrightarrow Q) \longrightarrow Q$
using *mem-add*$_e$*-E*
by *metis*

end

theory *IPC-errors-type*
imports *../TypeSchemes*
../Memory/SharedMemory

begin

4.9.15 Error codes datatype

4.10 HOL representation of PikeOS IPC error codes

— error codes are returned if an IPC action is aborted, the error codes has the following specificities:

- Must indicates which stage the error was occured.
- Each IPC stage has its own set of error codes
- Errors in the receiving stages does not affect sending stages
- Errors in sending stages affect receiving stages

We have another type of errors which is related to the different memory functionality.

— IPC errors
datatype *error-IPC* =

no-IPC-error
| *error-IPC-4* — if an action is used in stepping function with the wrong stage
— errors of the SEND part of IPC

| *error-IPC-21-in-PREP-SEND* — IF the receiver is an OR
| *error-IPC-22-in-PREP-SEND* — IF the receiver is an CR and the sender is not the one who can send msg to this receiver
| *error-IPC-23-in-PREP-SEND* — IF the receiver is an NR
| *error-IPC-4-in-PREP-SEND*— if an action is used in the wrong stage

| *error-IPC-21-in-PREP-RECV* — IF the receiver is an OR
| *error-IPC-22-in-PREP-RECV* — IF the receiver is an CR and the sender is not the one who can send msg to this receiver
| *error-IPC-23-in-PREP-RECV* — IF the receiver is an NR
| *error-IPC-4-in-PREP-RECV*— if an action is used in the wrong stage

| *error-IPC-1-in-WAIT-SEND* — if the thread has no rights to communicate with his partner
| *error-IPC-2-in-WAIT-SEND* — if the thread has no rights to access to this list of virtual adresses
| *error-IPC-3-in-WAIT-SEND* — if the thread try to send an IPC msg to him self

| *error-IPC-4-in-WAIT-SEND*— if an action is used in the wrong stage
| *error-IPC-5-in-WAIT-SEND* — if the receiver dont exist in the list of threads in the systeme
| *error-IPC-6-in-WAIT-SEND* — if the list of threads in the systeme is Nil
| *error-IPC-7-in-WAIT-SEND* — if the caller can not communicate with the receiver

|*error-IPC-1-in-BUF-SEND* — if the thread has no rights to access to this list of virtual adresses

|*error-IPC-1-in-BUF-RECV* — if the thread has no rights to access to this list of virtual adresses

| *error-IPC-1-in-WAIT-RECV* — if the thread has no rights to communicate with his partner
| *error-IPC-2-in-WAIT-RECV* — if the thread has no rights to access to this list of virtual adresses
| *error-IPC-3-in-WAIT-RECV* — if the thread try to send an IPC msg to him self
| *error-IPC-4-in-WAIT-RECV*— if an action is used in the wrong stage
| *error-IPC-5-in-WAIT-RECV* — if the receiver dont exist in the list of threads in the systeme Go to Done stage
| *error-IPC-6-in-WAIT-RECV* — if the list of threads in the systeme is Nil
| *error-IPC-7-in-WAIT-RECV* — if the caller can not communicate with the receiver

— memory errors
datatype *error-memory* =
no-mem-error — no errors related to memory adresses
| *not-valid-sender-addr-in-PREP-SEND* — error related to the adresses of the sender
| *not-valid-receiver-addr-in-PREP-SEND* — error related to the adresses of the receiver
| *not-valid-receiver-addr-in-PREP-RECV*
| *not-valid-sender-addr-in-PREP-RECV*

— datatype that contain memory and IPC errors

datatype *errors* =
NO-ERRORS
| *ERROR-MEM error-memory*
| *ERROR-IPC error-IPC*

type-synonym $error_{ipc}$ = *errors*
end

theory *IPC-thread-type*
imports ../*Memory*/*SharedMemory*
../*TypeSchemes*

begin

4.11 HOL representation of PikeOS threads type

datatype *thread-state* = *CURRENT* | *WAITING* | *READY* | *STOPPED* | *INACTIVE*

In addition to the communication rights, the scope of IPC communication can further constrained by the receiving thread.

- If thread initiates an OR operation, any threads having rights can send msg to this thread.
- If thread initiates CR operation, it limits the IPC sending partner to one specific thread.
- If thread initiates NR operation, no thread can send a message to this thread.

datatype *th-ipc-st* =
OR — Open Receive
| *CR* — Close Receive

| *NR* — Nil Receive

type-synonym $thread_{id}$ = (*nat* ∗ *nat* ∗ *nat*)

type-synonym $thread_{ipc}$ = ($thread_{id}$, *thread-state*, *th-ipc-st*, (*nat*, *int*) *memory* ,$thread_{id}$) *thread*

4.11.1 interface between thread and memory

definition *update-th-smm-equiv*
where *update-th-smm-equiv th addr val* = *update* (*own-vmem-adr th*) *addr val*

4.11.2 Relation between threads adresses and memory adresses

This section contains some predicate that defines relations between own thread addresses and memory addresses those predicate will be used to define some error codes related to own thread addresses.

— predicate that specify if this list of addresses are part of the addresses of the memory

definition *is-part-mem* ::
('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-mem mem addr* = (*addr* ∈ (*dom o fst o Rep-memory*) *mem*)

definition *is-part-mem-th* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-mem-th th mem addr* = (*is-part-mem* (*own-vmem-adr th*) *addr* ⟶ *is-part-mem mem addr*)

— predicate that specify if this list of addresses are part of the an other list of addresses

definition *is-part-addr-addr* ::
('*a*, '*b*) *memory* ⇒ ('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-addr-addr mem mem*' *addr*= (*is-part-mem mem*' *addr* ⟶ *is-part-mem mem addr*)

— This definition assures that a given list of addresses is part of list of addresses of thread
definition *is-part-addr-th* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒ '*a* ⇒ *bool*
where *is-part-addr-th th addr* = (*is-part-mem* (*own-vmem-adr th*) *addr*)

— This predicate assures that a given list of addresses is a part of memory addresses and part of thread addresses and the thread addresses are part of the memory

definition *is-part-addr-th-mem* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒ ('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-addr-th-mem th mem ns* = (*is-part-addr-addr mem* (*own-vmem-adr th*) *ns*)

lemma [*simp*]:*is-part-addr-th-mem th mem ns* = *is-part-mem-th th mem ns*
unfolding *is-part-addr-th-mem-def is-part-mem-th-def is-part-addr-addr-def*
by *simp*

4.11.3 Updating thread list in the state

— We will specify thread list inside our system by a partial function that takes a thread id and returns thread informations

type-synonym ('*th-id*, '*th-info*) *thread-tab* = '*th-id* ⇀ '*th-info*

fun *thread-tab-update* ::
('*th-id* ⇀ '*th-info*) ⇒ '*th-id* ⇒ '*th-info* ⇒ ('*th-id* ⇀ '*th-info*)

where *thread-tab-update th-tab th-id th-info* = *th-tab*(*th-id* ↦ *th-info*)

— Invariant on updating thread table

fun *update-th-waiting-true*::
  ('*th-id* ⇀ ('*a*, *thread-state*, '*b*, '*c*, '*d*, '*e*) *thread-scheme*) ⇒ '*th-id* ⇒ *bool*
where *update-th-waiting-true th-tab th-id* =
  (*th-id* ∈ *dom th-tab* ∧ ((*th-state o the o th-tab*) *th-id*) = *WAITING*)

fun *update-th-ready-true*::
  ('*th-id* ⇀ ('*a*, *thread-state*, '*b*, '*c*, '*d*, '*e*) *thread-scheme*) ⇒ '*th-id* ⇒ *bool*
where *update-th-ready-true th-tab th-id* =
  (*th-id* ∈ *dom th-tab* ∧ ((*th-state o the o th-tab*) *th-id*) = *READY*)

fun *update-th-current-true*::
  ('*th-id* ⇀ ('*a*, *thread-state*, '*b*, '*c*, '*d*, '*e*) *thread-scheme*) ⇒ '*th-id* ⇒ *bool*
where *update-th-current-true th-tab th-id* =
  (*th-id* ∈ *dom th-tab* ∧ ((*th-state o the o th-tab*) *th-id*) = *CURRENT*)

fun *update-th-stopped-true*::
  ('*th-id* ⇀ ('*a*, *thread-state*, '*b*, '*c*, '*d*, '*e*) *thread-scheme*) ⇒ '*th-id* ⇒ *bool*
where *update-th-stopped-true th-tab th-id* =
  (*th-id* ∈ *dom th-tab* ∧ ((*th-state o the o th-tab*) *th-id*) = *STOPPED*)

— update functions for thread state

fun *update-th-waiting*
where *update-th-waiting th-id th-tab* = (*if th-id* ∈ *dom th-tab*
  then th-tab(*th-id* ↦ ((*the o th-tab*) *th-id*)
    ⦇*th-state* := *WAITING*⦈)
  else th-tab)

fun *update-th-ready*
where *update-th-ready th-id th-tab* = (*if th-id* ∈ *dom th-tab*
  then th-tab(*th-id* ↦ ((*the o th-tab*) *th-id*)
    ⦇*th-state* := *READY*⦈)
  else th-tab)

fun *update-th-current*
where *update-th-current th-id th-tab* = (*if th-id* ∈ *dom th-tab*
  then th-tab(*th-id* ↦ ((*the o th-tab*) *th-id*)
    ⦇*th-state* := *CURRENT*⦈)
  else th-tab)

fun *update-th-stopped*
where *update-th-stopped th-id th-tab* = (*if th-id* ∈ *dom th-tab*
  then th-tab(*th-id* ↦ ((*the o th-tab*) *th-id*)
    ⦇*th-state* := *STOPPED*⦈)
  else th-tab)

4.11.4 Get thread by thread ID

— Function that find an element in the list under a given condition

primrec *find* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a option* **where**
find - [] = *None* |
find P (*x*#*xs*) = (*if P x then Some x else find P xs*)

— A thread equality procedure ... 2 threads are equal if they have the same ID

definition *thread-eq*
where $\textit{thread-eq th-id thread} = (\textit{th-id} = \textit{thread-id thread})$

— An interface that let us to get a thread structure using the thread ID

definition *get-thread-by-id*
where $\textit{get-thread-by-id th-id thl} = \textit{find}\ (\textit{thread-eq th-id})\ \textit{thl}$

end

theory *IPC-state-model*

imports *IPC-errors-type IPC-thread-type*

begin

4.12 HOL representation of state type model for IPC

4.12.1 informations on threads

record $('\textit{thread-id}, '\textit{error})\ \textit{th-info} =$
$\quad \textit{act-info} :: '\textit{thread-id} \rightharpoonup '\textit{error}$

record $\textit{state}_{id} = ((\textit{nat}, \textit{int})\textit{memory}, \textit{thread}_{id}, (\textit{thread}_{id}, \textit{thread}_{ipc})\ \textit{thread-tab},$
$(\textit{thread}_{id} \Rightarrow \textit{thread}_{id} \Rightarrow \textit{bool}),$
$(\textit{thread}_{id} \Rightarrow (\textit{nat}, \textit{int})\textit{memory} \Rightarrow \textit{bool}), \textit{errors})\ \textit{kstate} +$
$\textit{th-flag} :: (\textit{thread}_{id}, \ \textit{errors})\ \textit{th-info}$

4.12.2 Interface between IPC state and threads

— An interface that let us to get a thread structure using the thread ID inside a state

definition *get-thread-by-id'*
where $\textit{get-thread-by-id}'\ \textit{th-id}\ \sigma = (\textit{thread-list}\ (\sigma :: 'a\ \textit{state}_{id}\textit{-scheme}))\ \textit{th-id}$

4.12.3 Interface between IPC state and memory model

definition *upd-st-res-equiv*
where $\textit{upd-st-res-equiv}\ \sigma\ \textit{msg} = (\ \textit{update-th-smm-equiv}\ (\textit{current-thread}\ \sigma)\ (\textit{resource}\ \sigma)\ \textit{msg})$

definition $\textit{upd-st-res-equiv}_{id}$
where $\textit{upd-st-res-equiv}_{id}\ (\sigma :: \textit{state}_{id})\ \textit{msg} =$
$\quad \textit{update-th-smm-equiv}\ ((\textit{the}\ o\ (\textit{get-thread-by-id}'\ o\ \textit{current-thread})\ \sigma)\ \sigma)\ \textit{msg}$
$\qquad ((\textit{the}\ o\ (\textit{fst}\ o\ \textit{Rep-memory}\ o\ \textit{resource})\ \sigma)\ \textit{msg}\ )$

term $(\textit{thread-list}\ (\sigma :: 'a\ \textit{state}_{id}\textit{-scheme}))(\textit{partner}\ (\sigma :: 'a\ \textit{state}_{id}\textit{-scheme}))$

term $\textit{fold}\ (\lambda\ \textit{addr}\ y.\ \textit{update}\ y\ \textit{addr}\ ((\textit{the}\ o\ ((\textit{fst}\ o\ \textit{Rep-memory})\ (\textit{resource}\ (\sigma :: 'a\ \textit{state}_{id}\textit{-scheme}))))$
$\quad \textit{addr}))\ \textit{addr}\ (\textit{resource}\ (\sigma :: 'a\ \textit{state}_{id}\textit{-scheme}))$

term *fold* $(\lambda$ *addr y*. *update y addr val*$)$ *addr mem*

abbreviation
update-state caller σ *f error* $\equiv \sigma(|$*current-thread* := *caller*,
thread-list := *f caller* (*thread-list* σ),
error-codes := *error*$|)$

abbreviation
init-act-info caller partner $\sigma \equiv \sigma(|$*th-flag* := (*th-flag* σ)
$(|$*act-info* := (((*act-info o th-flag*)σ)
(*caller* := *None*,
partner:= *None*))$|)$
$|)$

lemma *fun-upd* (*fun-upd f x z*) *y* $z' = f(x{:=}z, y{:=}z')$
by *auto*

lemma
assumes *1*:$x \neq y$
and 2: *fun-upd f x z* = *g*
shows *g y* = *f y*
using *assms*
by *auto*

lemma
assumes *1*:$z \neq$ *None*
and 2:*fun-upd f x z* = *g*
shows *the z* $\in$ (*ran g*)
using *assms*
unfolding *ran-def*
by *auto*

end

theory *IPC-actions-preconditions*
imports *IPC-state-model*
begin

4.13 HOL representation of IPC preconditions

4.13.1 IPC conditions on threads parameters

This definition assures thats the partener thread is an Open Receive thread. If this condition is not satisfied when it is checked in a given IPC stage the corresponding error code *error-IPC-21-in-PREP-SEND* is returned

definition *IPC-params-c1* ::
('*a*, '*b*, *th-ipc-st*, '*c*, '*d*, '*e*) *thread-scheme* $\Rightarrow$ *bool*
where *IPC-params-c1 th* = (*th-ipc-st th* = *OR*)

lemma *IPC-params-c1-direct1*[*simp*] :
IPC-params-c1 $(|$*thread-id* = a_1, *th-state* = a_2,*th-ipc-st* = *OR*, *own-vmem-adr* = a_3,*cpartner* = $a_4|)$
by(*simp add*:*IPC-params-c1-def*)

lemma *IPC-params-c1-direct2*[*simp*] :

$\neg$*IPC-params-c1* $(\!|$*thread-id* $= a_1$, *th-state* $= a_2$,*th-ipc-st* $=$ *CR*, *own-vmem-adr* $= a_3$,*cpartner* $= a_4|\!)$
by(*simp add:IPC-params-c1-def*)

lemma *IPC-params-c1-direct3*[*simp*] :
$\neg$*IPC-params-c1* $(\!|$*thread-id* $= a_1$, *th-state* $= a_2$,*th-ipc-st* $=$ *NR*, *own-vmem-adr* $= a_3$,*cpartner* $= a_4|\!)$
by(*simp add:IPC-params-c1-def*)

the corresponding error code *error-IPC-22-in-PREP-SEND* is returned

definition *IPC-params-c2* ::
('*a*, '*b*, *th-ipc-st*, '*c*, '*d*, '*e*) *thread-scheme* $\Rightarrow$ *bool*
where *IPC-params-c2 th* = (*th-ipc-st th* = *CR*)

the corresponding error code *error-IPC-23-in-PREP-SEND* is returned

definition *IPC-params-c3* ::
('*a*, '*b*, *th-ipc-st*, '*c*, '*d*, '*e*) *thread-scheme* $\Rightarrow$ *bool*
where *IPC-params-c3 th* = (*th-ipc-st th* = *NR*)

definition *IPC-params-c4*
::$thread_{id} \Rightarrow thread_{id} \Rightarrow bool$
where *IPC-params-c4 caller partner* = (*caller* $\neq$ *partner*)

definition *IPC-params-c6*
::$thread_{id} \Rightarrow (thread_{id}$, '*b*, *th-ipc-st*, '*c*, $thread_{id}$, '*e*) *thread-scheme* $\Rightarrow$ *bool*
where *IPC-params-c6 caller partner* = (*caller* = *cpartner partner*)

definition *IPC-params-c5*
::$thread_{id} \Rightarrow$ '*a* $state_{id}$-*scheme* $\Rightarrow$ *bool*
where *IPC-params-c5 caller* σ = (*caller* $\in$ (*dom* (*thread-list* σ)) $\wedge$
(*th-state o the*)((*thread-list* σ) *caller*) $\neq$ *STOPPED*)

4.13.2 IPC conditions on threads communication rights

definition *IPC-sub-sub-sp*
::$thread_{id} \Rightarrow thread_{id} \Rightarrow (thread_{id} \Rightarrow thread_{id} \Rightarrow bool) \Rightarrow (thread_{id} \rightharpoonup thread_{ipc}) \Rightarrow bool$
where *IPC-sub-sub-sp caller partner rel thl* = (*reflp rel* $\wedge$ *rel caller partner* $\wedge$
caller $\in$ *dom thl* $\wedge$ *partner* $\in$ *dom thl*)

definition *IPC-send-comm-check*
::$thread_{id} \Rightarrow thread_{id} \Rightarrow (thread_{id} \Rightarrow thread_{id} \Rightarrow bool) \Rightarrow (thread_{id} \rightharpoonup thread_{ipc}) \Rightarrow bool$
where *IPC-send-comm-check caller partner rel thl* =
(*IPC-sub-sub-sp caller partner rel thl* $\wedge$ *IPC-params-c4 caller partner*)

definition *IPC-recv-comm-check*
::$thread_{id} \Rightarrow thread_{id} \Rightarrow (thread_{id} \Rightarrow thread_{id} \Rightarrow bool) \Rightarrow (thread_{id} \rightharpoonup thread_{ipc}) \Rightarrow bool$
where *IPC-recv-comm-check caller partner rel thl* = *IPC-sub-sub-sp caller partner rel thl*

4.13.3 IPC conditions on threads access rights

definition *IPC-sub-obj-sp*
where *IPC-sub-obj-sp* = *undefined*

definition *IPC-buf-check*
:: $thread_{id} \Rightarrow thread_{id} \Rightarrow$ (*nat*, *int*) *memory* $\Rightarrow$ ($thread_{id} \Rightarrow$ (*nat*, *int*) *memory* $\Rightarrow$*bool*)$\Rightarrow$
$(thread_{id} \rightharpoonup thread_{ipc}) \Rightarrow$*bool*
where *IPC-buf-check caller partner mem rel thl* =
(*caller* $\in$ *dom thl* $\wedge$ *partner* $\in$ *dom thl* $\wedge$
(*dom o fst o Rep-memory*)((*own-vmem-adr o the o thl*) *caller*) $\subseteq$
((*dom o fst o Rep-memory*) *mem*) $\wedge$ *rel partner mem*)

definition *IPC-map-check*
where $IPC\text{-}map\text{-}check = undefined$

4.13.4 interface between IPC Preconditions and IPC $'a\ state_{id}\text{-}scheme$

definition *IPC-send-comm-check-st*$_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow 'a\ state_{id}\text{-}scheme \Rightarrow bool$
where $IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma =$
$(IPC\text{-}sub\text{-}sub\text{-}sp\ caller\ partner\ (communication\text{-}rights\ \sigma)\ (thread\text{-}list\ \sigma) \wedge$
$IPC\text{-}params\text{-}c4\ caller\ partner)$

definition *IPC-recv-comm-check-st*$_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow 'a\ state_{id}\text{-}scheme \Rightarrow bool$
where $IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma =$
$IPC\text{-}sub\text{-}sub\text{-}sp\ caller\ partner\ (communication\text{-}rights\ \sigma)\ (thread\text{-}list\ \sigma)$

definition *IPC-buf-check-st*$_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow 'a\ state_{id}\text{-}scheme \Rightarrow bool$
where $IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma =$
$IPC\text{-}buf\text{-}check\ caller\ partner\ (resource\ \sigma)\ (access\text{-}rights\ \sigma)\ (thread\text{-}list\ \sigma)$

definition *IPC-map-check-st*$_{id}$
where $IPC\text{-}map\text{-}check\text{-}st_{id} = undefined$

end

theory *IPC-atomic-actions*
imports *IPC-actions-preconditions ../../../../src/TestLib*

begin

4.14 HOL representation of PikeOS IPC atomic actions

4.14.1 Types instantiation

In order to model PikeOS IPC API atomic actions, we will instantiate types of the parameters of *a* by other Isabelle datatypes as following:

datatype *p4-stage*$_{ipc}$ =
PREP — checking file descriptor informations
| *WAIT* — synchronising
| *BUF* — MEM COPY
| *MAP* — MEM MAP
| *DONE* — IPC end

datatype (*'thread-id* , *'adresses*)
p4-direct$_{ipc}$ =
SEND 'thread-id 'thread-id 'adresses
| *RECV 'thread-id 'thread-id 'adresses*

datatype (*'thread-id* , *'adresses*) *action*$_{ipc}$*-simplified* =
IPC-SEND 'thread-id 'thread-id 'adresses
| *IPC-RECV 'thread-id 'thread-id 'adresses*

To avoid the complexe representation of memory, we represent the memory content as a list of integers and the adresses are natural numbers. An id of the thread is represented by a tuple of natural numbers that specify, the task and the partition that the thread belongs to. To use this abstraction on PikeOS IPC API in our nvironment, we will just define a new type and instantiate our free variables a and b by Isabelle natural numbers type as follwing:

type-synonym $\mathit{p4\text{-}action}_{ipc}\mathit{\text{-}simplified} = (\mathit{nat} \times \mathit{nat} \times \mathit{nat}\ , \mathit{nat\ list})\ \mathit{action}_{ipc}\mathit{\text{-}simplified}$

type-synonym $\mathit{ACTION}_{ipc} = (\mathit{p4\text{-}stage}_{ipc}, (\mathit{nat} \times \mathit{nat} \times \mathit{nat}, \mathit{nat\ list})\ \mathit{p4\text{-}direct}_{ipc})\ \mathit{action}_{ipc}$

type-synonym $('o,'\sigma)Mon_{SE} = '\sigma \rightharpoonup ('o * '\sigma)$

4.14.2 Atomic actions semantics

Actually, PikeOS IPC API provides 7 system calls. An execution of each system call will split it to atomic actions. Those atomic actions are called *stages*. In order to execute The $p4\_ipc\_send$ call, the kernel will split it into 4 stages:

1. *PREP* stage
2. *WAIT* stage
3. *BUF* stage
4. *DONE* stage

In addition of providing interruption points, the execution of those stages is used to provide a security model to the IPC mechanism. In each stage and during the execution a set of conditions will be checked by the kernel. If one of the conditions is not satisfied, for example the communication security policy is not respected, the kernel abort the call and return an error code.

4.14.3 Semantics of atomic actions with thread IDs as arguments

lemma *is-part-addr-th-mem a b c= is-part-mem-th a b c*
 unfolding *is-part-addr-th-mem-def is-part-mem-th-def is-part-addr-addr-def*
 by (*simp*)

definition $\mathit{PREP\text{-}SEND}_{id}$
  $:: 'a\ \mathit{state}_{id}\mathit{\text{-}scheme} \Rightarrow \mathit{ACTION}_{ipc} \Rightarrow 'a\ \mathit{state}_{id}\mathit{\text{-}scheme}$
where $\mathit{PREP\text{-}SEND}_{id}\ \sigma\ \mathit{act} =$
  (*case act of* (*IPC PREP* (*SEND caller partner msg*)) ⇒
   if list-all ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*
   then
    if IPC-params-c1 ((*the o thread-list* σ) *partner*)
    then σ(|*current-thread* := *caller*,
       thread-list := *update-th-ready caller* (*thread-list* σ),
       error-codes := *NO-ERRORS*|)
    else
     (*if IPC-params-c2* ((*the o thread-list* σ) *partner*)
     then
      if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
      then σ(|*current-thread* := *caller*,
        thread-list := *update-th-ready caller* (*thread-list* σ),
        error-codes := *NO-ERRORS*|)
      else
       σ(|*current-thread* := *caller*,
        thread-list := *update-th-current caller* (*thread-list* σ),

error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*⦈
else σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*⦈)
else σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*⦈)

definition $PREP\text{-}RECV_{id}$
:: $'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow\ 'a\ state_{id}\text{-}scheme$
where $PREP\text{-}RECV_{id}$ σ *act* = (
case act of (*IPC PREP* (*RECV caller partner msg*)) ⇒
if list-all ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*
then
if IPC-params-c1 ((*the o thread-list* σ) *partner*)
then σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈
else
(*if IPC-params-c2* ((*the o thread-list* σ) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
then σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈
else
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-RECV*⦈
else σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-RECV*⦈)
else σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*⦈)

definition $WAIT\text{-}SEND_{id}$
:: $'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow\ 'a\ state_{id}\text{-}scheme$
where $WAIT\text{-}SEND_{id}$ σ *act* =
(*case act of* (*IPC WAIT* (*SEND caller partner msg*)) ⇒
if ¬ $IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ
then σ ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*⦈
else
if ¬ *IPC-params-c4 caller partner*
then σ ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*⦈
else
if ¬*IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
σ ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*⦈

$| \; Some \; th \Rightarrow \sigma(\!|current\text{-}thread := caller \,,$
$\quad thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$\quad error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND|\!))$
$else$
$\sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}waiting \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := NO\text{-}ERRORS|\!))$

definition *WAIT-RECV*$_{id}$
$::'a \; state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow {}'a \; state_{id}\text{-}scheme$
where $WAIT\text{-}RECV_{id} \; \sigma \; act =$
$(case \; act \; of \;\; (IPC \; WAIT \; (RECV \; caller \; partner \; msg) \;) \Rightarrow$
$if \; \neg \; IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id} \; caller \; partner \; \sigma$
$then \; \sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV|\!)$
$else$
$if \; \neg \; IPC\text{-}params\text{-}c4 \;\; caller \; partner$
$then \; \sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV|\!)$
$else$
$if \; \neg \; IPC\text{-}params\text{-}c5 \; partner \; \sigma$
$then$
$(case \; (thread\text{-}list \; \sigma) \; caller \; of \; None \Rightarrow$
$\sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV|\!)$
$| \; Some \; th \Rightarrow \sigma(\!|current\text{-}thread := caller \,,$
$\quad thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$\quad error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV|\!))$
$else$
$\sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}waiting \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := NO\text{-}ERRORS|\!))$

definition *BUF-SEND*$_{id}$
$::'a \; state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow {}'a \; state_{id}\text{-}scheme$
where $BUF\text{-}SEND_{id} \; \sigma \; act =$
$(case \; act \; of \; (IPC \; BUF \; (SEND \; caller \; partner \; msg) \;) \Rightarrow$
$if \;\; \neg \; IPC\text{-}buf\text{-}check\text{-}st_{id} \; caller \; partner \; \sigma$
$then \;\; \sigma(\!|current\text{-}thread := caller \,,$
$thread\text{-}list \;\; := update\text{-}th\text{-}current \; caller \; (thread\text{-}list \; \sigma),$
$error\text{-}codes \;\; := ERROR\text{-}IPC \; error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND|\!)$
$else$
$\sigma(\!|current\text{-}thread := caller,$
$resource \;\;\; := update\text{-}list \; (resource \; \sigma)$
$\quad (zip \; ((sorted\text{-}list\text{-}of\text{-}set.F \; o \; dom \;\; o \; fst \; o \; Rep\text{-}memory)$
$\quad\quad ((own\text{-}vmem\text{-}adr \; o \; the \; o \; thread\text{-}list \; \sigma) \; partner))$
$\quad (map \; ((the \; o \; (fst \; o \; Rep\text{-}memory) \; (resource \; \sigma))) \; msg)),$
$thread\text{-}list \;\; := update\text{-}th\text{-}ready \; caller$
$\quad (update\text{-}th\text{-}ready \; partner$
$\quad (thread\text{-}list \; \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS|\!)$
$(*if \; a \; BUF \; op \; is \; execute \; this \; means \; that \; there \; are \; no \; errors$

```
            in check stages∗))

definition BUF-RECV_id
     ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where   BUF-RECV_id σ act =
       (case act of (IPC BUF (RECV caller partner msg) ) ⇒
        if  ¬ IPC-buf-check-st_id caller partner σ
        then  σ(|current-thread := caller ,
             thread-list   := update-th-current caller (thread-list σ),
             error-codes   := ERROR-IPC error-IPC-1-in-BUF-RECV|)
        else
         σ(|current-thread := caller,
           resource     := update-list (resource σ)
                             (zip  ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                             ((own-vmem-adr o the o thread-list σ) caller))
                             (map ((the o (fst o Rep-memory) (resource σ))) msg)),
           thread-list  := update-th-ready caller
                       (update-th-ready partner
                       (thread-list σ)),
           error-codes  := NO-ERRORS|))



definition MAP-SEND_id
     ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where   MAP-SEND_id σ act =
       (case act of (IPC MAP (SEND caller partner msg) ) ⇒
         σ(|current-thread := caller,
           resource    := init-share-list (resource σ)
                      (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                           ((own-vmem-adr o the o thread-list σ) partner))),
           thread-list  := update-th-ready caller
                      (update-th-ready partner
                      (thread-list σ)),
            error-codes := NO-ERRORS|)
            (∗if a MAP op is execute this means that BUF was executed without errors∗))

definition MAP-RECV_id
     ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where   MAP-RECV_id σ act =
       (case act of (IPC MAP (RECV caller partner msg) ) ⇒
         σ(|current-thread := caller,
           resource    := init-share-list (resource σ)
                      (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                           ((own-vmem-adr o the o thread-list σ) caller))),
           thread-list  := update-th-ready caller
                      (update-th-ready partner
                      (thread-list σ)),
            error-codes := NO-ERRORS|)
            (∗if a MAP op is execute this means that BUF was executed without errors∗))



definition DONE-SEND_id
     ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where  DONE-SEND_id σ act = σ

definition DONE-RECV_id
```

```
  ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where  DONE-RECV_id σ act =  σ
```

4.14.4 Semantics of atomic actions based on monads

```
fun PREP-SEND_MON :: ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where
  PREP-SEND_MON  (IPC PREP (SEND caller partner msg)) σ =
   ( if list-all ((is-part-addr-th-mem o the) ((thread-list σ) caller) (resource σ))msg
    then
     if list-all ((is-part-mem-th o the) ((thread-list σ) partner) (resource σ))msg
     then
      if IPC-params-c1 ((the o thread-list σ)  partner)
      then unit_SE (NO-ERRORS)
            (σ(|current-thread := caller,
              thread-list    := update-th-ready caller (thread-list σ),
              error-codes    := NO-ERRORS|))
      else
       (if IPC-params-c2 ((the o thread-list σ)  partner)
        then
         if IPC-params-c6 caller ((the o thread-list σ) partner)
         then unit_SE (NO-ERRORS)
              (σ(|current-thread := caller,
                thread-list    := update-th-ready caller (thread-list σ),
                error-codes    := NO-ERRORS|))
         else
           unit_SE (ERROR-IPC error-IPC-22-in-PREP-SEND)
              (σ(|current-thread := caller,
                thread-list    := update-th-current caller (thread-list σ),
                error-codes    := ERROR-IPC error-IPC-22-in-PREP-SEND|))
        else unit_SE (ERROR-IPC error-IPC-23-in-PREP-SEND)
              (σ(|current-thread := caller,
                thread-list    := update-th-current caller (thread-list σ),
                error-codes    := ERROR-IPC error-IPC-23-in-PREP-SEND|)))
     else unit_SE (ERROR-MEM not-valid-receiver-addr-in-PREP-SEND)
          (σ(|current-thread := caller,
            thread-list    := update-th-current caller (thread-list σ),
            error-codes    := ERROR-MEM not-valid-receiver-addr-in-PREP-SEND|))
    else unit_SE (ERROR-MEM not-valid-sender-addr-in-PREP-SEND)
         (σ(|current-thread := caller,
           thread-list    := update-th-current caller (thread-list σ),
           error-codes    := ERROR-MEM not-valid-sender-addr-in-PREP-SEND|)))

  (∗hyÃĺpothese: all other atomic actions have no purge∗)
| PREP-SEND_MON   a σ =  unit_SE (error-codes σ ) σ


fun PREP-RECV_MON :: ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where
 PREP-RECV_MON   (IPC PREP (RECV caller partner msg)) σ =
  (if list-all ((is-part-addr-th-mem o the) ((thread-list σ) caller) (resource σ))msg
   then
    if list-all ((is-part-mem-th o the) ((thread-list σ) partner) (resource σ))msg
    then
     if IPC-params-c1 ((the o thread-list σ)  partner)
     then unit_SE (NO-ERRORS)
           (σ(|current-thread := caller,
             thread-list    := update-th-ready caller (thread-list σ),
```

```
              error-codes   := NO-ERRORS|))
    else
     (if IPC-params-c2 ((the o thread-list σ)  partner)
          then
           if IPC-params-c6 caller ((the o thread-list σ) partner)
           then unit_SE (NO-ERRORS)
                (σ(|current-thread := caller,
                  thread-list   := update-th-ready caller (thread-list σ),
                  error-codes   := NO-ERRORS|))
           else
            unit_SE (ERROR-IPC error-IPC-22-in-PREP-RECV)
               (σ(|current-thread := caller ,
                 thread-list   := update-th-current caller (thread-list σ),
                 error-codes   := ERROR-IPC error-IPC-22-in-PREP-RECV|))
          else
           unit_SE (ERROR-IPC error-IPC-23-in-PREP-RECV)
            (σ(|current-thread := caller ,
              thread-list   := update-th-current caller (thread-list σ),
              error-codes   := ERROR-IPC error-IPC-23-in-PREP-RECV|)))
        else
         unit_SE (ERROR-MEM not-valid-receiver-addr-in-PREP-RECV)
            (σ(|current-thread := caller ,
              thread-list   := update-th-current caller (thread-list σ),
              error-codes   := ERROR-MEM not-valid-receiver-addr-in-PREP-RECV|))
       else
        unit_SE (ERROR-MEM not-valid-sender-addr-in-PREP-RECV)
           (σ(|current-thread := caller ,
             thread-list   := update-th-current caller (thread-list σ),
             error-codes   := ERROR-MEM not-valid-sender-addr-in-PREP-RECV|)))

  (*hyÃĺpothese: all other atomic actions have no purge*)

| PREP-RECV_MON  a σ = unit_SE (error-codes σ ) σ

fun WAIT-SEND_MON :: ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme) option
where
  WAIT-SEND_MON  (IPC WAIT (SEND caller partner msg)) σ =
   (if ¬ IPC-send-comm-check-st_id caller partner σ
    then unit_SE (ERROR-IPC error-IPC-1-in-WAIT-SEND)
          (σ(|current-thread := caller,
            thread-list   := update-th-current caller (thread-list σ),
            error-codes   := ERROR-IPC error-IPC-1-in-WAIT-SEND|))
    else
     if ¬ IPC-params-c4  caller  partner
     then  unit_SE (ERROR-IPC error-IPC-3-in-WAIT-SEND)
            (σ (|current-thread := caller,
              thread-list   := update-th-current caller (thread-list σ),
              error-codes   := ERROR-IPC error-IPC-3-in-WAIT-SEND|))
     else
      if ¬IPC-params-c5 partner σ
      then
       (case (thread-list σ) caller of None ⇒
        unit_SE (ERROR-IPC error-IPC-6-in-WAIT-SEND)
         (σ (|current-thread := caller,
           thread-list   := update-th-waiting caller (thread-list σ),
           error-codes   := ERROR-IPC error-IPC-6-in-WAIT-SEND|))
       | Some th ⇒  unit_SE (ERROR-IPC error-IPC-5-in-WAIT-SEND)
                (σ(|current-thread := caller ,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\qquad\qquad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-5-in-WAIT-SEND}))) \\
\qquad \textit{else} \\
\qquad\quad \textit{unit}_{SE}\ (\textit{NO-ERRORS})\ (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\qquad\qquad \textit{thread-list} \quad := \textit{update-th-waiting caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\qquad\qquad \textit{error-codes} \quad := \textit{NO-ERRORS}))) \\
\mid \textit{WAIT-SEND}_{MON}\ \ a\ \sigma = \ \textit{unit}_{SE}\ (\textit{error-codes } \sigma\ )\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \textit{WAIT-RECV}_{MON} :: \textit{ACTION}_{ipc} \Rightarrow {}'a\ \textit{state}_{id}\textit{-scheme} \Rightarrow (\textit{errors} * {}'a\ \textit{state}_{id}\textit{-scheme})\ \textit{option} \\
\textbf{where } \textit{WAIT-RECV}_{MON}\ \ (\textit{IPC WAIT } (\textit{RECV caller partner msg}))\ \sigma = \\
\quad (\ \textit{if } \neg\ \textit{IPC-recv-comm-check-st}_{id}\ \textit{caller partner } \sigma \\
\quad\ \textit{then unit}_{SE}\ (\textit{ERROR-IPC error-IPC-1-in-WAIT-RECV}) \\
\qquad\qquad (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\ \ \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\ \ \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-1-in-WAIT-RECV})) \\
\quad\ \textit{else} \\
\quad\ \ \textit{if } \neg\ \textit{IPC-params-c4}\ \ \textit{caller partner} \\
\quad\ \ \textit{then}\ \ \textit{unit}_{SE}\ (\textit{ERROR-IPC error-IPC-3-in-WAIT-RECV}) \\
\qquad\qquad\ (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\quad \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\quad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-3-in-WAIT-RECV})) \\
\quad\ \ \textit{else} \\
\quad\ \ \ \textit{if } \neg\ \textit{IPC-params-c5 partner } \sigma \\
\quad\ \ \ \textit{then} \\
\quad\ \ \ \ (\textit{case } (\textit{thread-list } \sigma)\ \textit{caller of None} \Rightarrow \\
\quad\ \ \ \ \ \ \textit{unit}_{SE}\ (\textit{ERROR-IPC error-IPC-6-in-WAIT-RECV}) \\
\qquad\qquad\ (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\quad \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\quad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-6-in-WAIT-RECV})) \\
\quad\ \ \ \ \mid \textit{Some th} \Rightarrow \textit{unit}_{SE}\ (\textit{ERROR-IPC error-IPC-5-in-WAIT-RECV}) \\
\qquad\qquad\qquad\quad (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\qquad\qquad \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\qquad\qquad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-5-in-WAIT-RECV}))) \\
\quad\ \ \ \textit{else} \\
\quad\ \ \ \ \textit{unit}_{SE}\ (\textit{NO-ERRORS}) \\
\qquad\qquad (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\ \ \textit{thread-list} \quad := \textit{update-th-waiting caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\ \ \textit{error-codes} \quad := \textit{NO-ERRORS}))) \\
\\
\mid \textit{WAIT-RECV}_{MON}\ \ a\ \sigma = \textit{unit}_{SE}\ (\textit{error-codes } \sigma\ )\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \textit{BUF-SEND}_{MON} :: \textit{ACTION}_{ipc} \Rightarrow {}'a\ \textit{state}_{id}\textit{-scheme} \Rightarrow (\textit{errors} * {}'a\ \textit{state}_{id}\textit{-scheme})\ \textit{option} \\
\textbf{where} \\
\ \textit{BUF-SEND}_{MON}\ \ (\textit{IPC BUF } (\textit{SEND caller partner msg}))\ \sigma = \\
\quad (\textit{if}\ \ \neg\ \textit{IPC-buf-check-st}_{id}\ \textit{caller partner } \sigma \\
\quad\ \textit{then}\ \ \textit{unit}_{SE}\ (\textit{ERROR-IPC error-IPC-1-in-BUF-RECV}) \\
\qquad\qquad (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\qquad\ \ \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\qquad\qquad\ \ \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-1-in-BUF-RECV})) \\
\quad\ \textit{else} \\
\quad\ \ \textit{unit}_{SE}\ (\textit{NO-ERRORS}) \\
\qquad\ (\sigma(\textit{current-thread} := \textit{caller}, \\
\qquad\ \ \textit{resource} \qquad := \textit{update-list } (\textit{resource } \sigma) \\
\qquad\qquad\qquad\qquad\qquad (\textit{zip}\ \ ((\textit{sorted-list-of-set.F o dom}\ \ \textit{o fst o Rep-memory}) \\
\qquad\qquad\qquad\qquad\qquad ((\textit{own-vmem-adr o the o thread-list } \sigma)\ \textit{partner})) \\
\qquad\qquad\qquad\qquad\qquad (\textit{map } ((\textit{the o } (\textit{fst o Rep-memory})\ (\textit{resource } \sigma)))\ \textit{msg})), \\
\qquad\ \ \textit{thread-list} \quad := \textit{update-th-ready caller} \\
\qquad\qquad\qquad\qquad (\textit{update-th-ready partner}
\end{array}
$$

```
                    (thread-list σ)),
            error-codes   := NO-ERRORS⦈))
| BUF-SEND_MON  a σ =  unit_SE (error-codes σ ) σ

fun BUF-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where
 BUF-RECV_MON  (IPC BUF (RECV caller partner msg)) σ =
   ( if  ¬ IPC-buf-check-st_id caller partner σ
    then
     unit_SE (ERROR-IPC error-IPC-1-in-BUF-RECV)
     (σ⦇current-thread := caller ,
       thread-list   := update-th-current caller (thread-list σ),
       error-codes    := ERROR-IPC error-IPC-1-in-BUF-RECV⦈)
    else
     unit_SE (NO-ERRORS)
     (σ⦇current-thread := caller,
       resource      := update-list (resource σ)
                        (zip  ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                        ((own-vmem-adr o the o thread-list σ) caller))
                        (map ((the o (fst o Rep-memory) (resource σ))) msg)),
       thread-list   := update-th-ready caller
                    (update-th-ready partner
                    (thread-list σ)),
       error-codes   := NO-ERRORS⦈))
| BUF-RECV_MON  a σ =  unit_SE (error-codes σ ) σ


fun MAP-SEND_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where MAP-SEND_MON    (IPC MAP (SEND caller partner msg) ) σ =
     unit_SE (NO-ERRORS ) (σ⦇current-thread := caller,
                   resource     := init-share-list (resource σ)
                                (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                                  ((own-vmem-adr o the o thread-list σ) partner))),
                   thread-list  := update-th-ready caller
                               (update-th-ready partner
                               (thread-list σ)),
                   error-codes := NO-ERRORS⦈)
  | MAP-SEND_MON  a σ =  unit_SE (error-codes σ ) σ


fun MAP-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where MAP-RECV_MON    (IPC MAP (SEND caller partner msg) ) σ =
     unit_SE (NO-ERRORS)
         (σ⦇current-thread := caller,
          resource       := init-share-list (resource σ)
                         (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                            ((own-vmem-adr o the o thread-list σ) caller))),
          thread-list    := update-th-ready caller
                      (update-th-ready partner
                      (thread-list σ)),
           error-codes   := NO-ERRORS⦈)
 | MAP-RECV_MON  a σ =  unit_SE (error-codes σ ) σ

fun DONE-SEND_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where DONE-SEND_MON    a σ = unit_SE (error-codes σ ) σ

fun DONE-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors ∗ 'a state_id-scheme) option
where DONE-RECV_MON    a σ = unit_SE (error-codes σ ) σ
```

definition *IPC-protocol a* =
$(out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ (out2 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ (out3 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;$
$(out4 \leftarrow WAIT\text{-}RECV_{MON}\ a;\ (out5 \leftarrow BUF\text{-}SEND_{MON}\ a;\ (out6 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;$
$(out7 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a)))))))$

4.14.5 Execution function for PikeOS IPC atomic actions with thread IDs as arguments

fun $exec\text{-}action_{id}$
$::'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow 'a\ state_{id}\text{-}scheme$
where
$PREP\text{-}SEND\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg)) =$
$PREP\text{-}SEND_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))|$

$PREP\text{-}RECV\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))|$

$WAIT\text{-}SEND\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ WAIT(SEND\ caller\ partner\ msg)) =$
$WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))|$

$WAIT\text{-}RECV\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ WAIT(RECV\ caller\ partner\ msg)) =$
$WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))|$

$BUF\text{-}SEND\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg)) =$
$BUF\text{-}SEND_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ |$

$BUF\text{-}RECV\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ BUF(RECV\ caller\ partner\ msg)) =$
$BUF\text{-}RECV_{id}\ \sigma\ (IPC\ BUF(RECV\ caller\ partner\ msg))\ |$
$MAP\text{-}SEND\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ MAP\ (SEND\ caller\ partner\ msg)) =$
$MAP\text{-}SEND_{id}\ \sigma\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ |$

$MAP\text{-}RECV\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ MAP\ (RECV\ caller\ partner\ msg)) =$
$MAP\text{-}RECV_{id}\ \sigma\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ |$

$DONE\text{-}SEND\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ DONE(SEND\ caller\ partner\ msg)) = \sigma\ \ |$

$DONE\text{-}RECV\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ DONE(RECV\ caller\ partner\ msg)) = \sigma$

4.14.6 Predicates on atomic actions

Different cases of send action

definition $actions\text{-}send\text{-}cases\ a\ caller\ partner\ msg = (a = IPC\ PREP\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ (SEND\ caller\ partner\ msg))$

Different cases of receive action

definition $actions\text{-}receiv\text{-}cases\ a\ caller\ partner\ msg = (a = IPC\ PREP\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ (RECV\ caller\ partner\ msg)\ )$

A comparison procedure between actions. Used to indentify actions that can reply to an aborted system call.

definition $actioneq\text{-}op\ a\ a' = (case\ a\ of$
$(IPC\ PREP\ (SEND\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}receiv\text{-}cases\ a'\ partner\ caller\ msg)$

| (IPC PREP (RECV caller partner msg)) $\Rightarrow$
  (actions-send-cases a' partner caller msg)
| (IPC WAIT (SEND caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' partner caller msg)
| (IPC WAIT (RECV caller partner msg)) $\Rightarrow$
  (actions-send-cases a' partner caller msg)
| (IPC BUF (SEND caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' partner caller msg)
| (IPC BUF (RECV caller partner msg)) $\Rightarrow$
  (actions-send-cases a' partner caller msg)
| (IPC DONE (SEND caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' partner caller msg)
| (IPC DONE (RECV caller partner msg)) $\Rightarrow$
  (actions-send-cases a' partner caller msg)

)

A comparison procedure between actions. Used to indentify actions that will be aborted.

definition *actioneq a a'* = (*case a of*
  (IPC PREP (SEND caller partner msg)) $\Rightarrow$
  (actions-send-cases a' caller partner msg)
| (IPC PREP (RECV caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' caller partner msg)
| (IPC WAIT (SEND caller partner msg)) $\Rightarrow$
  (actions-send-cases a' caller partner msg)
| (IPC WAIT (RECV caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' caller partner msg)
| (IPC BUF (SEND caller partner msg)) $\Rightarrow$
  (actions-send-cases a' caller partner msg)
| (IPC BUF (RECV caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' caller partner msg)
| (IPC DONE (SEND caller partner msg)) $\Rightarrow$
  (actions-send-cases a' caller partner msg)
| (IPC DONE (RECV caller partner msg)) $\Rightarrow$
  (actions-receiv-cases a' caller partner msg)
)

4.14.7 Lemmas and simplification rules related to atomic actions

lemma *mem-inv1*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) = *resource* σ
  apply (*auto simp* : *WAIT-SEND*$_{id}$*-def*)
  apply (*cases thread-list* σ *caller,auto*)
  done

lemma *mem-inv2*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) = *resource* σ
  apply (*auto simp* : *WAIT-RECV*$_{id}$*-def*)
  apply (*cases thread-list* σ *caller,auto*)
  done

lemma *mem-inv3*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) = *resource* σ
  by (*auto simp* : *PREP-RECV*$_{id}$*-def*)

lemma *mem-inv4*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) = *resource* σ
  by (*auto simp* : *PREP-SEND*$_{id}$*-def*)

lemma *mem-inv5*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC BUF*(*RECV caller partner msg*))) =
   (*if* $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ
    then resource σ
    else update-list (*resource* σ)
              (*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
              ((*own-vmem-adr o the o thread-list* σ) *caller*))
              (*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)))
  by (*auto simp* : *BUF-RECV*$_{id}$*-def*)

lemma *mem-inv5-E*:
  assumes *1*: σ' = *resource* (*exec-action*$_{id}$ σ (*IPC BUF*(*RECV caller partner msg*)))
  and   *2*: $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \Longrightarrow \sigma'$ = *resource* $\sigma \Longrightarrow Q$
  and   *3*: *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \Longrightarrow$
        σ' = *update-list* (*resource* σ)
              (*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
              ((*own-vmem-adr o the o thread-list* σ) *caller*))
              (*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)) $\Longrightarrow Q$
  shows *Q*
proof –
  show *?thesis*
  using *1* **unfolding** *mem-inv5*
   proof (*cases* $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ)
    case *True*
    show *?thesis*
    using *True 1* **unfolding** *mem-inv5*
    by (*simp*, *elim 2*)
   next
    case *False*
    show *?thesis*
    using *False 1* **unfolding** *mem-inv5*
    by (*simp*, *elim 3*, *simp*)
   qed
qed

lemma *mem-inv6*[*simp*]:
  resource (*exec-action*$_{id}$ σ (*IPC BUF*(*SEND caller partner msg*))) =
   (*if* $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ
    then resource σ
    else update-list (*resource* σ)
              (*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
               ((*own-vmem-adr o the o thread-list* σ) *partner*))
               (*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)))
  by (*auto simp* :*BUF-SEND*$_{id}$*-def*)

lemma *mem-inv6-E*:
  assumes *1*: σ' = *resource* (*exec-action*$_{id}$ σ (*IPC BUF*(*SEND caller partner msg*)))
  and   *2*: $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \Longrightarrow \sigma'$ = *resource* $\sigma \Longrightarrow Q$
  and   *3*: *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \Longrightarrow$
        σ' = *update-list* (*resource* σ)
              (*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
              ((*own-vmem-adr o the o thread-list* σ) *partner*))
              (*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)) $\Longrightarrow Q$
  shows *Q*
proof –
  show *?thesis*
  using *1* **unfolding** *mem-inv5*

proof (*cases* $\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ)
case *True*
show *?thesis*
using *True 1* **unfolding** *mem-inv6*
by (*simp*, *elim 2*)
next
case *False*
show *?thesis*
using *False 1* **unfolding** *mem-inv6*
by (*simp*, *elim 3*, *simp*)
qed
qed

lemma *mem-inv7*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*))) = *resource* σ
by *simp*

lemma *mem-inv8*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*))) = *resource* σ
by *simp*

lemma *mem-inv9*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*)))
unfolding *mem-inv3 mem-inv4*
by *simp*

lemma *mem-inv10*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*)))
unfolding *mem-inv4 mem-inv1*
by *simp*

lemma *mem-inv11*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*)))
unfolding *mem-inv2 mem-inv4*
by *simp*

lemma *mem-inv12*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv4*
by *simp*

lemma *mem-inv13*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv4*
by *simp*

lemma *mem-inv14*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*)))
unfolding *mem-inv3 mem-inv1*
by *simp*

lemma *mem-inv15*[*simp*]:

resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*)))
unfolding *mem-inv2 mem-inv3*
by *simp*

lemma *mem-inv16*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv3*
by *simp*

lemma *mem-inv17*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv1*
by *simp*

lemma *mem-inv18*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv1*
by *simp*

lemma *mem-inv19*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv2*
by *simp*

lemma *mem-inv20*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv2*
by *simp*

lemma *mem-inv21*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
by *simp*

4.14.8 Composition equality on same action

For the general case the order of the executions of PikeOS matter iff executed on the same action, because the semantics of the execution related to each action is separated

lemma *sem-comp-prep-send1*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*) = (*out1* ← *PREP-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send2*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*) = (*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send3*:

$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{WAIT-RECV}_{MON}\ a) = (out1 \leftarrow \textit{WAIT-RECV}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send4*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{BUF-SEND}_{MON}\ a) = (out1 \leftarrow \textit{BUF-SEND}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*,
rule p4-direct$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send5*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{BUF-RECV}_{MON}\ a) = (out1 \leftarrow \textit{BUF-RECV}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*,
rule p4-direct$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send6*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{MAP-SEND}_{MON}\ a) = (out1 \leftarrow \textit{MAP-SEND}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*,
rule p4-direct$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send7*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{MAP-RECV}_{MON}\ a) = (out1 \leftarrow \textit{MAP-RECV}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*,
rule p4-direct$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send8*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{DONE-SEND}_{MON}\ a) = (out1 \leftarrow \textit{DONE-SEND}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send9*:
$(out1 \leftarrow \textit{PREP-SEND}_{MON}\ a\ ;\ \textit{DONE-RECV}_{MON}\ a) = (out1 \leftarrow \textit{DONE-RECV}_{MON}\ a\ ;\ \textit{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv2*:
$(out1 \leftarrow \textit{PREP-RECV}_{MON}\ a\ ;\ \textit{WAIT-SEND}_{MON}\ a) = (out1 \leftarrow \textit{WAIT-SEND}_{MON}\ a\ ;\ \textit{PREP-RECV}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv3*:
$(out1 \leftarrow \textit{PREP-RECV}_{MON}\ a\ ;\ \textit{WAIT-RECV}_{MON}\ a) = (out1 \leftarrow \textit{WAIT-RECV}_{MON}\ a\ ;\ \textit{PREP-RECV}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv4*:
$(out1 \leftarrow \textit{PREP-RECV}_{MON}\ a\ ;\ \textit{BUF-SEND}_{MON}\ a) = (out1 \leftarrow \textit{BUF-SEND}_{MON}\ a\ ;\ \textit{PREP-RECV}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv5*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv6*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv7*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv8*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv9*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send4*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send5*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send6*:
(*out1* ←*WAIT-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send7*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ;*MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)

by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send8*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send9*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a*; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv4*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv5*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv6*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv7*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv8*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv9*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send6*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send7*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ a;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send8*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a) = (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send9*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a) = (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv6*:
$(out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv7*:
$(out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv8*:
$(out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a) = (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv9*:
$(out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a) = (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send6*:
$(out1 \leftarrow \text{MAP-SEND}_{MON}\ a\ ;\ \text{DONE-SEND}_{MON}\ a) = (out1 \leftarrow \text{DONE-SEND}_{MON}\ a\ ;\ \text{MAP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send7*:
$(out1 \leftarrow \text{MAP-SEND}_{MON}\ a\ ;\ \text{DONE-RECV}_{MON}\ a) = (out1 \leftarrow \text{DONE-RECV}_{MON}\ a\ ;\ \text{MAP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send8*:
$(out1 \leftarrow \text{MAP-SEND}_{MON}\ a\ ;\ \text{BUF-SEND}_{MON}\ a) = (out1 \leftarrow \text{BUF-SEND}_{MON}\ a\ ;\ \text{MAP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send9*:
$(out1 \leftarrow \text{MAP-SEND}_{MON}\ a\ ;\ \text{BUF-RECV}_{MON}\ a) = (out1 \leftarrow \text{BUF-RECV}_{MON}\ a\ ;\ \text{MAP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv6*:
$(out1 \leftarrow \text{MAP-RECV}_{MON}\ a\ ;\ \text{DONE-SEND}_{MON}\ a) = (out1 \leftarrow \text{DONE-SEND}_{MON}\ a\ ;\ \text{MAP-RECV}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv7*:
$(out1 \leftarrow \text{MAP-RECV}_{MON}\ a\ ;\ \text{DONE-RECV}_{MON}\ a) = (out1 \leftarrow \text{DONE-RECV}_{MON}\ a\ ;\ \text{MAP-RECV}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

4.14.9 Composition equality on different same actions: partial order reduction

For the specific case of IPC protocol the order of the executions of PikeOS does matter iff executed on different actions, because the semantics of the execution related to each action can react in some cases on the same field of the state, eg: the field related to erro codes... So the switch between the execution order related to IPC actions can be done but under some assumptions and only for a subset of actions

lemma *sem-comp-prep-send10*:
$(out1 \leftarrow \text{PREP-SEND}_{MON}\ a\ ;\ \text{DONE-SEND}_{MON}\ b) = (out1 \leftarrow \text{DONE-SEND}_{MON}\ b\ ;\ \text{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send11*:
$(out1 \leftarrow \text{PREP-SEND}_{MON}\ a\ ;\ \text{DONE-RECV}_{MON}\ b) = (out1 \leftarrow \text{DONE-RECV}_{MON}\ b\ ;\ \text{PREP-SEND}_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-prep-recv10*:
$(out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ b) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ b\ ;\ PREP\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-prep-recv11*:
$(out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ b) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ b\ ;\ PREP\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

term (*resource o snd o the*) $((out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ b)\ \sigma)$

lemma *WAIT-SEND*$_{MON}$*-None*: $WAIT\text{-}SEND_{MON}\ (IPC\ WAIT\ a)\ \sigma \neq None$
by (*induct a*, *auto simp add*: *unit-SE-def split*:*option*.*split*)

lemma *WAIT-RECV*$_{MON}$*-None*: $WAIT\text{-}RECV_{MON}\ (IPC\ WAIT\ a)\ \sigma \neq None$
by (*induct a*, *auto simp add*: *unit-SE-def split*:*option*.*split*)

lemma *sem-comp-wait-send10*:
$(out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ b) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ b\ ;\ WAIT\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-wait-send11*:
$(out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ b) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ b;\ WAIT\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-wait-recv10*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ b) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ b\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-wait-recv11*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ b) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ b\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def* , *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option*.*split*)

lemma *sem-comp-buf-send10*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ b) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ b\ ;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,

simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send11*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b*; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv10*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv11*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send10*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send11*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv8*:
(*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *MAP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv9*:
(*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *MAP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

end

theory *IPC-traces*

imports *IPC-atomic-actions*

begin

4.15 HOL representation of PikeOS IPC traces

type-synonym $trace_{ipc} = ACTION_{ipc}\ list$

4.15.1 Execution function for PikeOS IPC traces

definition $exec\text{-}action_{id}\text{-}Mon$
where $exec\text{-}action_{id}\text{-}Mon = (\lambda actl\ st.\ Some\ (error\text{-}codes(exec\text{-}action_{id}\ st\ actl),$
$exec\text{-}action_{id}\ st\ actl))$

4.15.2 Trace refinement

4.15.3 Execution function for actions with thread ID

lemma $(((act\text{-}info\ (th\text{-}flag\ \sigma))\ caller) = None) = (caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
by *auto*

lemma $caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$the((act\text{-}info\ (th\text{-}flag\ \sigma))\ caller) \in ran\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
by $(auto\ simp: ranI)$

abbreviation
$get\text{-}caller\text{-}error\ caller\ \sigma \equiv (the\ o(act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller$

abbreviation
$remove\text{-}caller\text{-}error\ caller\ \sigma \equiv \sigma(|th\text{-}flag := (th\text{-}flag\ \sigma)\ (|act\text{-}info := ((act\text{-}info\ (th\text{-}flag\ \sigma))$
$(caller := None))|)$
$|)$

abbreviation
$set\text{-}caller\text{-}partner\text{-}error\ caller\ partner\ \sigma\ \sigma'\ out' \equiv \sigma'(|th\text{-}flag := (th\text{-}flag\ \sigma)$
$(|act\text{-}info := ((act\text{-}info\ (th\text{-}flag\ \sigma))$
$(caller := Some(out'$
$(*just\ (a,out')?*)$
$),$
$partner := Some\ (out'$
$(*just\ (a,out')?*)$
$)))|)$
$|)$

abbreviation
$error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma' \equiv \sigma'(|th\text{-}flag := (th\text{-}flag\ \sigma)|)$

abbreviation
$set\text{-}no\text{-}error\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ msg \equiv \sigma'\ (|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto NO\text{-}ERRORS)|)|)$

abbreviation
$set\text{-}no\text{-}error\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ msg \equiv \sigma'\ (|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto NO\text{-}ERRORS)|)|)$

abbreviation
$set\text{-}no\text{-}error\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ msg \equiv \sigma'\ (|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto NO\text{-}ERRORS)|)|)$

abbreviation
set-no-error-dones caller partner σ σ' *msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *NO-ERRORS*)|)|)

abbreviation
set-no-error-prepr caller partner σ σ' *msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *NO-ERRORS*)|)|)

abbreviation
set-no-error-waitr caller partner σ σ' *msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *NO-ERRORS*)|)|)

abbreviation
set-no-error-bufr caller partner σ σ' *msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$*NO-ERRORS*)|)|)

abbreviation
set-no-error-doner caller partner σ σ' *msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *NO-ERRORS*)|)|)

abbreviation
set-error-mem-preps caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-waits caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-bufs caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-maps caller partner σ σ' *error-mem msg* $\equiv$
σ' (|*th-flag* :=*th-flag* σ(|*act-info* := *act-info* (*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-dones caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-prepr caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-waitr caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)|)|)

abbreviation
set-error-mem-bufr caller partner σ σ' *error-mem msg* $\equiv$ σ' (|*state*$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ

$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}mem\text{-}doner\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}dones\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

abbreviation
$set\text{-}error\text{-}ipc\text{-}doner\ caller\ partner\ \sigma\ \sigma'\ error\text{-}ipc\ msg \equiv \sigma'\ (\!| state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!| act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc) |\!) |\!)$

```
fun abort_lift :: (ACTION_ipc ⇒ (errors, 'a state_id-scheme)Mon_SE) ⇒
          (ACTION_ipc ⇒ (errors, 'a state_id-scheme)Mon_SE)
where abort_lift ioprog a σ =
   (case a of
     (IPC DONE (SEND caller partner msg)) ⇒
        if caller ∈ dom (act-info (th-flag σ)) (∗should add the condition: in which action ID
                                    the error occurs∗)
        then Some((the((act-info (th-flag σ)) caller))(∗shoud be: my error∗),
              σ(|th-flag := (th-flag σ) (|act-info := ((act-info (th-flag σ))
                                        (caller := None))|)
                      |))
        else (case ioprog a σ of
              None ⇒ None (∗never happens in our exec fun∗)
             | Some(out', σ') ⇒ Some(NO-ERRORS, σ)) (∗execute done∗)
    | (IPC DONE (RECV caller partner msg)) ⇒
        if caller ∈ dom (act-info (th-flag σ))
        then Some((the((act-info (th-flag σ)) caller))(∗shoud be: my error∗),
              σ(|th-flag := (th-flag σ) (|act-info := ((act-info (th-flag σ))
                                        (caller := None))|)
                      |))
        else (case ioprog a σ of
              None ⇒ None (∗never happens in our exec fun∗)
             | Some(out', σ') ⇒ Some(NO-ERRORS, σ)) (∗execute done∗)
    | (IPC - (SEND caller partner msg)) ⇒
         if caller ∈ dom (act-info (th-flag σ))
         then Some(get-caller-error caller σ(∗should be: my error∗), σ) (∗ purge ∗)
         else (case ioprog a σ of
              None ⇒ None (∗never happens in our exec fun∗)
             | Some(NO-ERRORS, σ') ⇒ Some(NO-ERRORS, error-tab-transfer caller σ σ')
             | Some(ERROR-MEM error-memory, σ') ⇒
               Some(ERROR-MEM error-memory,
                    set-caller-partner-error caller partner σ σ' (ERROR-MEM error-memory) )
             | Some(ERROR-IPC error-IPC, σ') ⇒
               Some(ERROR-IPC error-IPC,
                    set-caller-partner-error caller partner σ σ' (ERROR-IPC error-IPC) ))
              (∗both caller and partner were 'informed' to be in error−state.∗)
    | (IPC - (RECV caller partner msg)) ⇒
         if caller ∈ dom (act-info (th-flag σ))
         then Some(get-caller-error caller σ (∗should be: my error∗), σ) (∗purge∗)
         else (case ioprog a σ of
              None ⇒ None (∗never happens in our exec fun∗)
             | Some(NO-ERRORS, σ') ⇒ Some(NO-ERRORS, error-tab-transfer caller σ σ')
             | Some(ERROR-MEM error-memory, σ') ⇒
               Some(ERROR-MEM error-memory,
                    set-caller-partner-error caller partner σ σ' (ERROR-MEM error-memory) )
             | Some(ERROR-IPC error-IPC, σ') ⇒
               Some(ERROR-IPC error-IPC,
                    set-caller-partner-error caller partner σ σ' (ERROR-IPC error-IPC) ))
              (∗both caller and partner were 'informed' to be in error−state.∗)

     (∗hypothese: all other atomic actions have no purge∗)
   )
```

lemma *exec-action*$_{id}$*-Mon-th-flag0*:
  $a = IPC\ ipc\text{-}stage\ (ipc\text{-}direction) \implies ipc\text{-}stage \neq DONE \implies$
  $exec\text{-}action_{id}\text{-}Mon\ a\ \sigma = Some\ (NO\text{-}ERRORS,\sigma') \implies th\text{-}flag\ \sigma = th\text{-}flag\ \sigma'$
 unfolding *exec-action*$_{id}$*-Mon-def*
 apply *auto*
 apply (*cases ipc-stage*)
 apply (*case-tac ipc-direction*)
 apply *simp-all*
 unfolding *PREP-SEND*$_{id}$*-def PREP-RECV*$_{id}$*-def*
 apply *simp-all*
 apply (*case-tac ipc-direction*)
 apply *simp-all*
 unfolding *WAIT-SEND*$_{id}$*-def*
 apply *simp-all*
 apply *safe*
 apply (*case-tac thread-list* σ (*a*, *aa*, *b*))
 apply *simp-all*
 unfolding *WAIT-RECV*$_{id}$*-def*
 apply *simp-all*
 apply *safe*
 apply *simp-all*
 apply (*case-tac thread-list* σ (*a*, *aa*, *b*))
 apply *simp-all*
 apply (*case-tac ipc-direction*)
 apply *simp-all*
 unfolding *BUF-SEND*$_{id}$*-def*
 apply *simp-all*
 unfolding *BUF-RECV*$_{id}$*-def*
 apply *simp-all*
 apply (*cases ipc-direction*)
 apply (*simp-all add*: *MAP-SEND*$_{id}$*-def MAP-RECV*$_{id}$*-def*)
 done

4.15.4 IPC operations with thread ID

We define an *operation* as a trace with a given order on atomic actions. For the IPC API we will define two types of operations, we call the first type *request* and the second type *reply*. Following this terminology a given PikeOS thread can request to communicate with another thread or reply to a communication request. The Isabelle specification of operations is as following:

definition *ipc-send-request*$_{id}$
  $:: thread_{id} \Rightarrow nat\ list \Rightarrow thread_{id} \Rightarrow trace_{ipc}\ ((\text{-} \rhd_{id}\ \text{-} \rhd_{id}/\ \text{-})\ [201, 0, 201]\ 200)$
where
  $caller \rhd_{id} msg \rhd_{id} partner \equiv [IPC\ PREP\ (SEND\ caller\ partner\ msg),$
  $IPC\ WAIT\ (SEND\ caller\ partner\ msg)]$

definition *ipc-recv-request*$_{id}$
  $:: thread_{id} \Rightarrow nat\ list \Rightarrow thread_{id} \Rightarrow trace_{ipc}\ ((\text{-} \lhd_{id}\ \text{-} \lhd_{id}/\ \text{-})\ [201, 0, 201]\ 200)$
where
  $caller \lhd_{id} msg \lhd_{id} partner \equiv [IPC\ PREP\ (RECV\ caller\ partner\ msg),$
  $IPC\ WAIT\ (RECV\ caller\ partner\ msg)]$

— A thread can do response operation to sending or receiving message response

definition *ipc-send-response*$_{id}$
  $:: thread_{id} \Rightarrow nat\ list \Rightarrow thread_{id} \Rightarrow trace_{ipc}\ ((\text{-} \unrhd_{id}\ \text{-} \unrhd_{id}/\ \text{-})\ [201, 0, 201]\ 200)$

where

$caller \trianglerighteq_{id}\ msg \trianglerighteq_{id}\ partner \equiv [IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$IPC\ WAIT\ (SEND\ caller\ partner\ msg),$
$IPC\ BUF\ \ (SEND\ caller\ partner\ msg),$
$IPC\ DONE\ (SEND\ caller\ partner\ msg),$
$IPC\ DONE\ (RECV\ partner\ caller\ msg)]$

definition *ipc-recv-response*$_{id}$
$::thread_{id} \Rightarrow nat\ list \Rightarrow thread_{id} \Rightarrow trace_{ipc}$ ((- $\trianglelefteq_{id}$ - $\trianglelefteq_{id}$/ -) [201, 0, 201] 200)
where

$caller \trianglelefteq_{id}\ msg \trianglelefteq_{id}\ partner \equiv [IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$IPC\ BUF\ \ (RECV\ caller\ partner\ msg),$
$IPC\ DONE\ (SEND\ partner\ caller\ msg),$
$IPC\ DONE\ (RECV\ caller\ partner\ msg)]$

lemmas *request-normalizer* =
ipc-send-response$_{id}$*-def* *ipc-recv-response*$_{id}$*-def* *ipc-send-request*$_{id}$*-def* *ipc-recv-request*$_{id}$*-def*

4.15.5 IPC operations with free variables

abbreviation *ipc-send-request* ((- $\triangleright$ - $\triangleright$/ -) [201, 0, 201] 200)
where $caller \triangleright msg \triangleright partner \equiv [IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$IPC\ WAIT\ (SEND\ caller\ partner\ msg)]$

abbreviation *ipc-recv-request* ((- $\triangleleft$ - $\triangleleft$/ -) [201, 0, 201] 200)
where $caller \triangleleft msg \triangleleft partner \equiv [IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$IPC\ WAIT\ (RECV\ caller\ partner\ msg)]$

abbreviation *ipc-send-response* ((- $\trianglerighteq$ - $\trianglerighteq$/ -) [201, 0, 201] 200)
where $caller \trianglerighteq msg \trianglerighteq partner \equiv [IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$IPC\ WAIT\ (SEND\ caller\ partner\ msg),$
$IPC\ BUF\ \ (SEND\ caller\ partner\ msg),$
$IPC\ MAP\ \ (SEND\ caller\ partner\ msg),$
$IPC\ DONE\ (SEND\ caller\ partner\ msg),$
$IPC\ DONE\ (RECV\ partner\ caller\ msg)]$

abbreviation *ipc-recv-response* ((- $\trianglelefteq$ - $\trianglelefteq$/ -) [201, 0, 201] 200)
where

$caller \trianglelefteq msg \trianglelefteq partner \equiv [IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$IPC\ BUF\ \ (RECV\ caller\ partner\ msg),$
$IPC\ MAP\ \ (RECV\ caller\ partner\ msg),$
$IPC\ DONE\ (SEND\ partner\ caller\ msg),$
$IPC\ DONE\ (RECV\ caller\ partner\ msg)]$

4.15.6 Pridicates on operations

definition *is-ipc-trace*
where $is\text{-}ipc\text{-}trace\ actl = (\forall a \in set(actl::trace_{ipc}). \exists caller\ partner\ msg.$
$a = IPC\ PREP\ \ (RECV\ caller\ partner\ msg) \lor$
$a = IPC\ WAIT\ \ (RECV\ caller\ partner\ msg) \lor$
$a = IPC\ BUF\ \ \ (RECV\ caller\ partner\ msg) \lor$
$a = IPC\ DONE\ \ (RECV\ caller\ partner\ msg) \lor$
$a = IPC\ PREP\ \ (SEND\ caller\ partner\ msg) \lor$
$a = IPC\ WAIT\ \ (SEND\ caller\ partner\ msg) \lor$
$a = IPC\ BUF\ \ \ (SEND\ caller\ partner\ msg) \lor$
$a = IPC\ DONE\ \ (SEND\ caller\ partner\ msg)\ )$

definition *is-ipc-trace$_{id}$*
where *is-ipc-trace$_{id}$ actl* $= (\forall\, a \in set(actl::trace_{ipc}).\exists\, caller\ partner\ msg.$
$a = IPC\ PREP\ \ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ \ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ \ \ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ \ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ PREP\ \ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ \ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ \ \ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ \ (SEND\ caller\ partner\ msg)\ )$

4.15.7 Simplification rules related to traces

lemma *prep-send-comp-mbind-eq2*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send1*)

lemma *prep-send-comp-mbind-eq3*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send2*)

lemma *prep-send-comp-mbind-eq4*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send3*)

lemma *prep-send-comp-mbind-eq5*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send4*)

lemma *prep-send-comp-mbind-eq6*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send5*)

lemma *prep-send-comp-mbind-eq7*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send6*)

lemma *prep-send-comp-mbind-eq8*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send7*)

lemma *prep-send-comp-mbind-eq9*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send8*)

lemma *prep-send-comp-mbind-eq10*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send9*)

lemma *prep-recv-comp-mbind-eq1*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ \ WAIT\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv2*)

lemma *prep-recv-comp-mbind-eq2*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ \ WAIT\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv3*)

lemma *prep-recv-comp-mbind-eq3*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv4*)

lemma *prep-recv-comp-mbind-eq4*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv5*)

lemma *prep-recv-comp-mbind-eq5*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv6*)

lemma *prep-recv-comp-mbind-eq6*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv7*)

lemma *prep-recv-comp-mbind-eq7*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv8*)

lemma *prep-recv-comp-mbind-eq8*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv9*)

lemma *wait-send-comp-mbind-eq1*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-send4*)

lemma *wait-send-comp-mbind-eq2*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-send5*)

lemma *wait-send-comp-mbind-eq3*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$

by (*simp only*: *sem-comp-wait-send6*)

lemma *wait-send-comp-mbind-eq4*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-send7*)

lemma *wait-send-comp-mbind-eq5*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-send8*)

lemma *wait-send-comp-mbind-eq6*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-send9*)

lemma *wait-recv-comp-mbind-eq1*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv4*)

lemma *wait-recv-comp-mbind-eq2*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv5*)

lemma *wait-recv-comp-mbind-eq3*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv6*)

lemma *wait-recv-comp-mbind-eq4*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv7*)

lemma *wait-recv-comp-mbind-eq5*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv8*)

lemma *wait-recv-comp-mbind-eq6*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-wait-recv9*)

lemma *buf-send-comp-mbind-eq1*:
$mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
$mbind\ is\ (\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-send6*)

lemma *buf-send-comp-mbind-eq2*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}SEND}_{MON}\ a\ ;\ \mathit{DONE\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}RECV}_{MON}\ a\ ;\ \mathit{BUF\text{-}SEND}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-send7*)

lemma *buf-send-comp-mbind-eq3*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}SEND}_{MON}\ a\ ;\ \mathit{MAP\text{-}SEND}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}SEND}_{MON}\ a\ ;\ \mathit{BUF\text{-}SEND}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-send8*)

lemma *buf-send-comp-mbind-eq4*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}SEND}_{MON}\ a\ ;\ \mathit{MAP\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}RECV}_{MON}\ a\ ;\ \mathit{BUF\text{-}SEND}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-send9*)

lemma *map-send-comp-mbind-eq1*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}SEND}_{MON}\ a\ ;\ \mathit{DONE\text{-}SEND}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}SEND}_{MON}\ a\ ;\ \mathit{MAP\text{-}SEND}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-map-send6*)

lemma *map-send-comp-mbind-eq2*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}SEND}_{MON}\ a\ ;\ \mathit{DONE\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}RECV}_{MON}\ a\ ;\ \mathit{MAP\text{-}SEND}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-map-send7*)

lemma *buf-recv-comp-mbind-eq1*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}RECV}_{MON}\ a\ ;\ \mathit{DONE\text{-}SEND}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}SEND}_{MON}\ a\ ;\ \mathit{BUF\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-recv6*)

lemma *buf-recv-comp-mbind-eq2*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}RECV}_{MON}\ a\ ;\ \mathit{DONE\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}RECV}_{MON}\ a\ ;\ \mathit{BUF\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-recv7*)

lemma *buf-recv-comp-mbind-eq3*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}RECV}_{MON}\ a\ ;\ \mathit{MAP\text{-}SEND}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}SEND}_{MON}\ a\ ;\ \mathit{BUF\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-recv8*)

lemma *buf-recv-comp-mbind-eq4*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{BUF\text{-}RECV}_{MON}\ a\ ;\ \mathit{MAP\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}RECV}_{MON}\ a\ ;\ \mathit{BUF\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-buf-recv9*)

lemma *map-recv-comp-mbind-eq1*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}RECV}_{MON}\ a\ ;\ \mathit{DONE\text{-}SEND}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}SEND}_{MON}\ a\ ;\ \mathit{MAP\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-map-recv6*)

lemma *map-recv-comp-mbind-eq2*:
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{MAP\text{-}RECV}_{MON}\ a\ ;\ \mathit{DONE\text{-}RECV}_{MON}\ a))\ \sigma =$
$\mathit{mbind\ is}\ (\lambda a.\ (\mathit{out1} \leftarrow \mathit{DONE\text{-}RECV}_{MON}\ a\ ;\ \mathit{MAP\text{-}RECV}_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-map-recv7*)

end

theory *IPC-step-normalizer*

imports *IPC-traces*

begin

4.16 IPC Stepping Function and Traces

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg =$
$(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \longrightarrow$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge$
$IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)))$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact2\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}fact3\ caller\ error\text{-}mem\ \sigma\ msg =$
$(\neg(list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$error\text{-}mem = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}fact4\ caller\ partner\ error\text{-}mem\ \sigma\ msg =$
$((list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ partner)\ (resource\ \sigma))msg) \wedge$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}fact3\ caller\ error\text{-}mem\ \sigma\ msg =$
$(\neg(list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$error\text{-}mem = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}RECV)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}fact4\ caller\ partner\ error\text{-}mem\ \sigma\ msg =$
$((list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ partner)\ (resource\ \sigma))msg) \wedge$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact5\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\vee$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\wedge$
$IPC\text{-}params\text{-}c4\ caller\ partner\ ) \wedge$
$IPC\text{-}params\text{-}c3\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\ )$

definition

$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact6\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\lor$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\land$
$IPC\text{-}params\text{-}c4\ \ caller\ \ partner\ )\ \land$
$\neg IPC\text{-}params\text{-}c3\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact7\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\lor$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\land$
$IPC\text{-}params\text{-}c4\ \ caller\ \ partner)\ )$

4.16.1 Simplification rules related to the stepping function $exec\text{-}action_{id}\text{-}Mon$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}mbind\text{-}obvious$:
$\bigwedge \sigma\ S.\ mbind\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by *simp*

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}mbind\text{-}obvious'$:
$(case\ mbind\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma\ of$
$None \Rightarrow Some\ ([get\text{-}caller\text{-}error\ caller\ \sigma], \sigma)$
$|\ Some\ (outs, \sigma'') \Rightarrow a) = a$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume $hyp0$: $mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
by *simp*
qed

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}all\text{-}obvious1$:
$\forall\ a\ \sigma.\exists\ errors\ \sigma'.\ \ exec\text{-}action_{id}\text{-}Mon\ a\ \sigma = Some\ (errors, \sigma')$
by $(auto, rule\ action_{ipc}.induct, auto\ simp{:}exec\text{-}action_{id}\text{-}Mon\text{-}def)$

Simplification rules on *PREP* action

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}obvious0$:
$\bigwedge\sigma.\ exec\text{-}action_{id}\text{-}Mon\ \ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by *simp*

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}obvious1$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma) =$
$(if\ (list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$then$
$if\ IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)$
$then\ Some\ (NO\text{-}ERRORS,$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS|\!))$

else
if IPC-params-c2 ((*the o thread-list* σ) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
then Some (*NO-ERRORS*,
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
else
Some(*ERROR-IPC error-IPC-22-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|))
else Some (*ERROR-IPC error-IPC-23-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|))
else Some (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|)))
by (*simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious2*:
(*fst o the*)(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)
then
if IPC-params-c1 ((*the o thread-list* σ) *partner*)
then NO-ERRORS
else
(*if IPC-params-c2* ((*the o thread-list* σ) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
then NO-ERRORS
else
ERROR-IPC error-IPC-22-in-PREP-SEND
else ERROR-IPC error-IPC-23-in-PREP-SEND)
else ERROR-MEM not-valid-sender-addr-in-PREP-SEND)
by (*simp add*:*exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious3*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')) =
(σ'= σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) $\wedge$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\wedge$
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ)
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
exec-action$_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious4*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')) =
((σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|)$\wedge$

$\neg$(*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*) $\wedge$
error-mem = *not-valid-sender-addr-in-PREP-SEND*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious5*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
((σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|)$\wedge$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\wedge$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\wedge$
IPC-params-c2 ((*the o thread-list* σ) *partner*) $\wedge$
$\neg$ *IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) $\wedge$
error-IPC = *error-IPC-22-in-PREP-SEND*) $\vee$
(σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|)$\wedge$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\wedge$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\wedge$
$\neg$ *IPC-params-c2* ((*the o thread-list* σ) *partner*) $\wedge$
error-IPC = *error-IPC-23-in-PREP-SEND*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def*
exec-action$_{id}$*-Mon-prep-fact0-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious0*:
$\forall\sigma$. *exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)
then
if IPC-params-c1 ((*the o thread-list* σ) *partner*)
then Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
else
(*if IPC-params-c2* ((*the o thread-list* σ) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
then Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
else
Some(*ERROR-IPC error-IPC-22-in-PREP-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-RECV*|))

else Some(*ERROR-IPC error-IPC-23-in-PREP-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-RECV*|)))
else Some (*ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*|)))
by(*simp add*: *exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-prep-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) *σ*)) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list σ*) *caller*) (*resource σ*))*msg*)
then
if IPC-params-c1 ((*the o thread-list σ*) *partner*)
then NO-ERRORS
else
(*if IPC-params-c2* ((*the o thread-list σ*) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list σ*) *partner*)
then NO-ERRORS
else
ERROR-IPC error-IPC-22-in-PREP-RECV
else ERROR-IPC error-IPC-23-in-PREP-RECV)
else ERROR-MEM not-valid-receiver-addr-in-PREP-RECV)
unfolding *exec-action$_{id}$-Mon-def*
by (*simp add*: *exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-prep-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ′*)) =
(*σ′*= *σ*(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list σ*),
error-codes := *NO-ERRORS*|) ∧
exec-action$_{id}$-Mon-prep-fact0 caller partner σ msg∧
exec-action$_{id}$-Mon-prep-fact1 caller partner σ)
by (*auto simp add*: *exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def exec-action$_{id}$-Mon-prep-fact0-def*
exec-action$_{id}$-Mon-prep-fact1-def
split: *errors.split split-if split-if-asm*)

lemma *exec-action$_{id}$-Mon-prep-recv-obvious4*:
(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ′*)) =
((*σ′*= *σ*(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*|)∧
¬(*list-all* ((*is-part-mem-th o the*) ((*thread-list σ*) *caller*) (*resource σ*))*msg*) ∧
error-mem = *not-valid-receiver-addr-in-PREP-RECV*))
by (*auto simp add*: *exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action$_{id}$-Mon-prep-recv-obvious5*:
(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) *σ* = *Some*(*ERROR-IPC error-IPC*, *σ′*)) =
((*σ′*=*σ*(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-RECV*|)∧
exec-action$_{id}$-Mon-prep-fact0 caller partner σ msg ∧

¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
IPC-params-c2 ((*the o thread-list* σ) *partner*) ∧
¬*IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-22-in-PREP-RECV*) ∨
(σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-RECV*|)∧
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
¬ *IPC-params-c2* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-23-in-PREP-RECV*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-RECV*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def*
exec-action$_{id}$*-Mon-prep-fact0-def*
split: *errors.split split-if split-if-asm*)

Simplification rules on WAIT action

lemma *exec-action*$_{id}$*-Mon-wait-send-obvious0*:
⋀σ. *exec-action*$_{id}$*-Mon* (*IPC WAIT* (*SEND caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

definition
exec-action$_{id}$*-Mon-wait-send-upd caller* σ =
(*case* (*thread-list* σ) *caller of None* ⇒
σ (|*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*|)
| *Some th* ⇒ σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*|))

lemma *exec-action*$_{id}$*-Mon-wait-send-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ) =
(*if* ¬ *IPC-send-comm-check-st*$_{id}$ *caller partner* σ
then Some(*ERROR-IPC error-IPC-1-in-WAIT-SEND* ,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*|))
else
if ¬ *IPC-params-c4 caller partner*
then Some(*ERROR-IPC error-IPC-3-in-WAIT-SEND*,
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*|)
)
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
Some (*ERROR-IPC error-IPC-6-in-WAIT-SEND* ,
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*|))
| *Some th* ⇒ *Some* (*ERROR-IPC error-IPC-5-in-WAIT-SEND* ,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),

error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*⦈))
else
Some(*NO-ERRORS*,
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈))
by (*simp add*: *exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def list.induct split*: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-send-obvious2*:
fst (*the*(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ)) =
(*if* ¬ *IPC-send-comm-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-WAIT-SEND
else
if ¬ *IPC-params-c4 caller partner*
then ERROR-IPC error-IPC-3-in-WAIT-SEND
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
ERROR-IPC error-IPC-6-in-WAIT-SEND
| *Some th* ⇒ *ERROR-IPC error-IPC-5-in-WAIT-SEND*)
else
NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def list.induct*
split: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-send-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')) =
(σ'= σ⦇*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈ ∧
IPC-send-comm-check-st$_{id}$ caller partner σ ∧
IPC-params-c4 caller partner ∧
IPC-params-c5 partner σ)
by (*auto simp add*: *exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def split*: *option.split-asm*)

definition
update-state-wait-send-params5 σ *caller* =
(*case* (*thread-list* σ) *caller of None* ⇒
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*⦈
| *Some th* ⇒ σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*⦈)

definition
update-state-wait-recv-params5 σ *caller* =
(*case* (*thread-list* σ) *caller of None* ⇒
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-RECV*⦈
| *Some th* ⇒ σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*⦈)

lemma *exec-action$_{id}$-Mon-wait-send-obvious4*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
((¬ *IPC-send-comm-check-st$_{id}$ caller partner* σ ⟶
σ'= σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*⦈ ∧
error-IPC = *error-IPC-1-in-WAIT-SEND*) ∧
(*IPC-send-comm-check-st$_{id}$ caller partner* σ ⟶
((¬ *IPC-params-c4 caller partner* ⟶
σ'= σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*⦈ ∧
error-IPC = *error-IPC-3-in-WAIT-SEND*) ∧
(*IPC-params-c4 caller partner* ⟶
((¬ *IPC-params-c5 partner* σ ⟶
σ'= *update-state-wait-send-params5* σ *caller* ∧
error-codes (*update-state-wait-send-params5* σ *caller*) = *ERROR-IPC error-IPC*) ∧
¬ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-send-params5-def exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def*
split: *split-if-asm option.split-asm*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious0*:
⋀σ. *exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-wait-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ) =
(*if* ¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ
then Some(*ERROR-IPC error-IPC-1-in-WAIT-RECV*,
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*⦈)
else
if ¬ *IPC-params-c4 caller partner*
then Some(*ERROR-IPC error-IPC-3-in-WAIT-RECV*,
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*⦈)
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
Some(*ERROR-IPC error-IPC-6-in-WAIT-RECV*,
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-RECV*⦈)
| *Some th* ⇒ *Some*(*ERROR-IPC error-IPC-5-in-WAIT-RECV*,
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*⦈))
else
Some(*NO-ERRORS*,

σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*simp add*: *exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def list.induct split*: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ)) =
(*if* ¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-WAIT-RECV
else
if ¬ *IPC-params-c4 caller partner*
then ERROR-IPC error-IPC-3-in-WAIT-RECV
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
ERROR-IPC error-IPC-6-in-WAIT-RECV
| *Some th* ⇒ *ERROR-IPC error-IPC-5-in-WAIT-RECV*)
else
NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def list.induct split*: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')) =
(σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) ∧
IPC-recv-comm-check-st$_{id}$ caller partner σ ∧
IPC-params-c4 caller partner ∧
IPC-params-c5 partner σ)
by (*auto simp add*: *exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def split*: *list.split-asm*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious4*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
((¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ ⟶
σ'=σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|) ∧
error-IPC = *error-IPC-1-in-WAIT-RECV*) ∧
(*IPC-recv-comm-check-st$_{id}$ caller partner* σ ⟶
((¬ *IPC-params-c4 caller partner* ⟶
σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|) ∧
error-IPC = *error-IPC-3-in-WAIT-RECV*) ∧
(*IPC-params-c4 caller partner* ⟶
((¬ *IPC-params-c5 partner* σ ⟶
σ'= *update-state-wait-recv-params5* σ *caller* ∧
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) ∧
¬ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-recv-params5-def exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def*
split: *split-if-asm list.split-asm*)

Simplification rules on BUF action

lemma *exec-action$_{id}$-Mon-buf-send-obvious0*:

$\bigwedge\sigma$. *exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-buf-send-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) σ) =
(*if* ¬ *IPC-buf-check-st$_{id}$ caller partner* σ
then Some (*ERROR-IPC error-IPC-1-in-BUF-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*|))
else
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|)))
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-send-obvious2*:
fst (*the*(*exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) σ)) =
(*if* ¬ *IPC-buf-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-BUF-SEND
else NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-send-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*error*, σ')) =
((¬ *IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
σ' = σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*|) ∧
error = *ERROR-IPC error-IPC-1-in-BUF-SEND*) ∧
(*IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
(σ' = σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|) ∧
error =*NO-ERRORS*)))
by (*auto simp add*: *exec-action$_{id}$-Mon-def BUF-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-recv-obvious0*:
$\forall\,\sigma.$ *exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-buf-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ) =
(*if* $\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ
then Some (*ERROR-IPC error-IPC-1-in-BUF-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|))
else
Some(*NO-ERRORS*,
(σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))))
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ)) =
(*if* $\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-BUF-RECV
else NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*error*, σ')) =
(($\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
σ' = σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|) $\wedge$
error = *ERROR-IPC error-IPC-1-in-BUF-RECV*) $\wedge$
(*IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
(σ' = σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|) $\wedge$
error =*NO-ERRORS*)))
by (*auto simp add*: *exec-action$_{id}$-Mon-def BUF-RECV$_{id}$-def*)

Simplification rules on MAP action

lemma *exec-action$_{id}$-Mon-map-send-obvious0*:
$\bigwedge\sigma$. *exec-action$_{id}$-Mon* (*IPC MAP* (*SEND caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-map-send-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC MAP* (*SEND caller partner msg*)) σ) =
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*simp add*: *exec-action$_{id}$-Mon-def MAP-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-map-send-obvious2*:
fst (*the*(*exec-action$_{id}$-Mon* (*IPC MAP* (*SEND caller partner msg*)) σ)) = *NO-ERRORS*
by (*simp add*: *exec-action$_{id}$-Mon-def MAP-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-map-send-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*error*, σ')) =
(σ' = σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|) $\wedge$
error =*NO-ERRORS*)
by (*auto simp add*: *exec-action$_{id}$-Mon-def MAP-SEND$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-map-recv-obvious0*:
$\bigwedge\sigma$. *exec-action$_{id}$-Mon* (*IPC MAP* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-map-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ) =
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),

$error\text{-}codes := NO\text{-}ERRORS\rVert)$
by $(simp\ add: exec\text{-}action_{id}\text{-}Mon\text{-}def\ MAP\text{-}RECV_{id}\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}recv\text{-}obvious2$:
$fst\ (the(exec\text{-}action_{id}\text{-}Mon\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma)) = NO\text{-}ERRORS$
by $(simp\ add: exec\text{-}action_{id}\text{-}Mon\text{-}def\ MAP\text{-}RECV_{id}\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}recv\text{-}obvious3$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(error, \sigma')) =$
$(\sigma' = \sigma\lVert current\text{-}thread := caller,$
$resource := init\text{-}share\text{-}list\ (resource\ \sigma)$
$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\rVert \wedge$
$error = NO\text{-}ERRORS)$
by $(auto\ simp\ add: exec\text{-}action_{id}\text{-}Mon\text{-}def\ MAP\text{-}RECV_{id}\text{-}def)$

Simplification rules on DONE action

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}done\text{-}send\text{-}obvious0$:
$\forall\,\sigma.\ exec\text{-}action_{id}\text{-}Mon\ \ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by $simp$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}done\text{-}send\text{-}obvious1$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma) =$
$Some(error\text{-}codes\ \sigma, \sigma)$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by $simp$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}done\text{-}send\text{-}obvious2$:
$fst\ (the(exec\text{-}action_{id}\text{-}Mon\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma)) =$
$error\text{-}codes\ \sigma$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by $simp$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}done\text{-}send\text{-}obvious3$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma = Some(error, \sigma')) =$
$(\sigma' = \sigma \wedge error\text{-}codes\ \sigma = error)$
by $(auto\ simp\ add: exec\text{-}action_{id}\text{-}Mon\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}done\text{-}recv\text{-}obvious0$:
$\bigwedge\sigma.\ exec\text{-}action_{id}\text{-}Mon\ \ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by $simp$

lemma *exec-action$_{id}$-Mon-done-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ) =
Some(*error-codes* σ, σ)
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-done-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ)) =
error-codes σ
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-done-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ = *Some*(*error*, σ')) =
($\sigma' = \sigma \wedge$ *error-codes* σ = *error*)
by (*auto simp add*: *exec-action$_{id}$-Mon-def*)

lemma *exec-action$_{id}$-Mon-act-info-obvious0*:
(*exec-action$_{id}$-Mon a* σ = *Some*(*error*, σ')) $\Longrightarrow$
(*act-info* (*state$_{id}$.th-flag* σ) = *act-info* (*state$_{id}$.th-flag* σ'))
unfolding *exec-action$_{id}$-Mon-def*
by (*auto*, *rule action$_{ipc}$.induct* , *rule p4-stage$_{ipc}$.induct*,*rule p4-direct$_{ipc}$.induct*,
auto, *rule action$_{ipc}$.induct*, *simp-all*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
auto simp: *PREP-SEND$_{id}$-def PREP-RECV$_{id}$-def* ,*rule p4-direct$_{ipc}$.induct*, *auto*,
simp add: *WAIT-SEND$_{id}$-def split*: *option.split*, *simp add*: *WAIT-RECV$_{id}$-def split*: *option.split*,
rule p4-direct$_{ipc}$.induct, *auto simp add*: *BUF-SEND$_{id}$-def BUF-RECV$_{id}$-def*,
rule p4-direct$_{ipc}$.induct,*auto simp add*: *MAP-SEND$_{id}$-def MAP-RECV$_{id}$-def*,
rule p4-direct$_{ipc}$.induct, *auto*)

lemma *exec-action$_{id}$-Mon-act-info-obvious0'*:
(*exec-action$_{id}$-Mon a* σ = *Some*(*error*, σ')) =
(*act-info* (*state$_{id}$.th-flag* σ) = *act-info* (*state$_{id}$.th-flag* σ') $\wedge$
error-codes (*exec-action$_{id}$* σ *a*) = *error* $\wedge$ *exec-action$_{id}$* σ *a* = σ')
unfolding *exec-action$_{id}$-Mon-def*
by (*auto*, *rule action$_{ipc}$.induct* , *rule p4-stage$_{ipc}$.induct*,*rule p4-direct$_{ipc}$.induct*,
auto, *rule action$_{ipc}$.induct*, *simp-all*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
auto simp: *PREP-SEND$_{id}$-def PREP-RECV$_{id}$-def* ,*rule p4-direct$_{ipc}$.induct*, *auto*,
simp add: *WAIT-SEND$_{id}$-def split*: *option.split*, *simp add*: *WAIT-RECV$_{id}$-def split*: *option.split*,
rule p4-direct$_{ipc}$.induct, *auto simp add*: *BUF-SEND$_{id}$-def BUF-RECV$_{id}$-def*,
rule p4-direct$_{ipc}$.induct,*auto simp add*: *MAP-SEND$_{id}$-def MAP-RECV$_{id}$-def*,
rule p4-direct$_{ipc}$.induct, *auto*)

lemma *exec-action$_{id}$-Mon-act-info-obvious1*:
exec-action$_{id}$-Mon (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*error*, σ') $\Longrightarrow$
act-info (*state$_{id}$.th-flag* σ) *caller*= *act-info* (*state$_{id}$.th-flag* σ') *caller*
by (*auto simp*:*exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-act-info-obvious2*:*act-info* (*state$_{id}$.th-flag* σ) *caller* =
act-info(*th-flag*(*snd*(*the*(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ)))) *caller*

unfolding *exec-action$_{id}$-Mon-def*

by (*simp add*: $PREP\text{-}RECV_{id}$*-def*)

lemma *exec-errors-obvious0*: $(exec\text{-}action_{id}\text{-}Mon\ a\ \sigma) = Some\ (NO\text{-}ERRORS,\sigma') \Longrightarrow$
$error\text{-}codes\ \sigma' = NO\text{-}ERRORS$
by (*auto simp only*: $exec\text{-}action_{id}$*-Mon-def prod.inject the.simps*)

lemma *exec-errors-obvious1*: $(exec\text{-}action_{id}\text{-}Mon\ a\ \sigma) = Some\ (NO\text{-}ERRORS,\sigma') \Longrightarrow$
$error\text{-}codes\ \sigma' \neq ERROR\text{-}MEM\ error\text{-}mem$
by (*auto simp only*: $exec\text{-}action_{id}$*-Mon-def prod.inject the.simps*)

lemma *exec-errors-obvious2*: $(exec\text{-}action_{id}\text{-}Mon\ a\ \sigma) = Some\ (NO\text{-}ERRORS,\sigma') \Longrightarrow$
$error\text{-}codes\ \sigma' \neq ERROR\text{-}IPC\ error\text{-}ipc$
by (*auto simp only*: $exec\text{-}action_{id}$*-Mon-def prod.inject the.simps*)

lemmas *step-normalizer-None* =
$exec\text{-}action_{id}$*-Mon-prep-send-obvious0* $exec\text{-}action_{id}$*-Mon-prep-recv-obvious0*
$exec\text{-}action_{id}$*-Mon-wait-send-obvious0* $exec\text{-}action_{id}$*-Mon-wait-recv-obvious0*
$exec\text{-}action_{id}$*-Mon-buf-send-obvious0* $exec\text{-}action_{id}$*-Mon-buf-recv-obvious0*
$exec\text{-}action_{id}$*-Mon-done-send-obvious0* $exec\text{-}action_{id}$*-Mon-done-recv-obvious0*

lemmas *step-normalizer-Some* = $exec\text{-}action_{id}\text{-}Mon\text{-}act\text{-}info\text{-}obvious0'$
end

theory *IPC-atomic-action-normalizer*

imports *IPC-step-normalizer*

begin

4.17 Atomic Actions Reasoning

4.17.1 Symbolic Execution Rules of Atomic Actions

lemma *prep-send-obvious*:
$(PREP\text{-}SEND_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg)) = \sigma') =$
$(((\sigma' = \sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := NO\text{-}ERRORS|\!)\ \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma))\vee$

$(\ (\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)\ \wedge$
$\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND|\!)))\vee$

$((\sigma' = \ \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND|\!)\ \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))\ \ \vee$
$(\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$

error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*⦈ ∧
exec-action$_{id}$*-Mon-prep-fact0 caller partner σ msg* ∧
¬*IPC-params-c1* ((*the o thread-list σ*) *partner*) ∧
¬*IPC-params-c2* ((*the o thread-list σ*) *partner*))))
by (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
exec-action$_{id}$*-Mon-prep-fact1-def*
exec-action$_{id}$*-Mon-prep-fact2-def*)

lemma *wait-send-obvious*:
(*WAIT-SEND*$_{id}$ *σ* (*IPC WAIT* (*SEND caller partner msg*)) = *σ′*) =
(*σ′*= *σ*⦇*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list σ*),
error-codes := *NO-ERRORS*⦈ ∧
IPC-send-comm-check-st$_{id}$ *caller partner σ* ∧
IPC-params-c4 caller partner ∧
IPC-params-c5 partner σ) ∨

((¬ *IPC-send-comm-check-st*$_{id}$ *caller partner σ* ⟶
σ′= *σ* ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*⦈) ∧
(*IPC-send-comm-check-st*$_{id}$ *caller partner σ* ⟶
((¬ *IPC-params-c4 caller partner* ⟶
σ′= *σ* ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*⦈)))))
by (*auto simp add*: *update-state-wait-send-params5-def WAIT-SEND*$_{id}$*-def*
split: *split-if-asm option.split-asm*)

lemma *buf-send-obvious*:
(*BUF-SEND*$_{id}$ *σ* (*IPC BUF* (*SEND caller partner msg*)) = *σ′*) =
((¬ *IPC-buf-check-st*$_{id}$ *caller partner σ* ⟶
σ′ = *σ*⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*⦈) ∧
(*IPC-buf-check-st*$_{id}$ *caller partner σ* ⟶
(*σ′* = *σ*⦇*current-thread* := *caller*,
resource := *update-list* (*resource σ*)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list σ*) *partner*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource σ*))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list σ*)),
error-codes := *NO-ERRORS*⦈)))
by (*auto simp add*:*BUF-SEND*$_{id}$*-def*)

lemma *map-send-obvious*:
(*MAP-SEND*$_{id}$ *σ* (*IPC MAP* (*SEND caller partner msg*)) = *σ′*) =
(*σ′* = *σ*⦇*current-thread* := *caller*,
resource := *init-share-list* (*resource σ*)

$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))),$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\|))$
by $(auto\ simp\ add{:}\ MAP\text{-}SEND_{id}\text{-}def)$

lemma *prep-recv-obvious*:
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) = \sigma') =$
$(((\sigma' = \sigma(|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS|) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma)) \vee$

$(\ (\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$\sigma' = \sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV|))) \vee$

$(\ (\sigma' = \ \sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV|) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))\ \ \vee$
$(\sigma' = \sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV|) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner))\ ))$
by $(auto\ simp\ add{:}\ PREP\text{-}RECV_{id}\text{-}def\ \ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact2\text{-}def$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\text{-}def\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\text{-}def)$

lemma *wait-recv-obvious*:
$(WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \ = \ \sigma') =$
$(\sigma' = \sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS|) \wedge$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \wedge$
$IPC\text{-}params\text{-}c4\ \ caller\ partner\ \ \wedge$
$IPC\text{-}params\text{-}c5\ partner\ \sigma) \vee$

$((\neg\ IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$\sigma' = \sigma\ (|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV|)\ ) \wedge$
$(IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\neg\ IPC\text{-}params\text{-}c4\ caller\ partner \longrightarrow$

$\sigma' = \sigma$ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|)))))
by (*auto simp add*: *update-state-wait-recv-params5-def* $\textit{WAIT-RECV}_{id}$*-def*
split: *split-if-asm*)

lemma *buf-recv-obvious*:
($\textit{BUF-RECV}_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) = σ') =
(($\neg$ $\textit{IPC-buf-check-st}_{id}$ *caller partner* σ $\longrightarrow$
$\sigma' = \sigma$(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|)) $\wedge$
($\textit{IPC-buf-check-st}_{id}$ *caller partner* σ $\longrightarrow$
($\sigma' = \sigma$(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))))
by (*auto simp add*: $\textit{BUF-RECV}_{id}$*-def*)

lemma *map-recv-obvious*:
($\textit{MAP-RECV}_{id}$ σ (*IPC MAP* (*RECV caller partner msg*)) = σ') =
($\sigma' = \sigma$(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: $\textit{MAP-RECV}_{id}$*-def*)

4.17.2 Symbolic Execution Rules for Error Codes Field

lemma $\textit{PREP-SEND}_{id}$*-obvious0*:
(*error-codes* ($\textit{PREP-SEND}_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *NO-ERRORS*) =
($\textit{exec-action}_{id}$*-Mon-prep-fact0 caller partner* σ *msg*$\wedge$
$\textit{exec-action}_{id}$*-Mon-prep-fact1 caller partner* σ $\wedge$
($\textit{PREP-SEND}_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*auto simp add*: $\textit{PREP-SEND}_{id}$*-def* $\textit{exec-action}_{id}$*-Mon-prep-fact0-def*
$\textit{exec-action}_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)

lemma $\textit{PREP-SEND}_{id}$*-obvious1*:
(*error-codes* ($\textit{PREP-SEND}_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *ERROR-MEM error-mem*) =

$(\neg((\textit{list-all}\ ((\textit{is-part-mem-th}\ o\ \textit{the})\ ((\textit{thread-list}\ \sigma)\ \textit{caller})\ (\textit{resource}\ \sigma))\textit{msg}))\ \wedge$
$\textit{error-mem} = \textit{not-valid-sender-addr-in-PREP-SEND}\ \wedge$
$(\textit{PREP-SEND}_{id}\ \sigma\ (\textit{IPC PREP}\ (\textit{SEND caller partner msg})) =$
$\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$
$\textit{thread-list}\ := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$
$\textit{error-codes}\ := \textit{ERROR-MEM not-valid-sender-addr-in-PREP-SEND}|\!)))$
by (*auto simp add*: *PREP-SEND*$_{id}$*-def split*: *errors.split split-if split-if-asm*)

lemma *PREP-SEND*$_{id}$*-obvious2*:
$(\textit{error-codes}\ (\textit{PREP-SEND}_{id}\ \sigma\ (\textit{IPC PREP}\ (\textit{SEND caller partner msg}))) = \textit{ERROR-IPC error-IPC}) =$
$(\neg(\textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner}\ \sigma\ \textit{msg}\ \wedge$
$\neg\textit{IPC-params-c1}\ ((\textit{the}\ o\ \textit{thread-list}\ \sigma)\ \ \textit{partner})\ \wedge$
$\textit{IPC-params-c2}\ ((\textit{the}\ o\ \textit{thread-list}\ \sigma)\ \ \textit{partner})\ \wedge$
$\neg\textit{IPC-params-c6 caller}\ ((\textit{the}\ o\ \textit{thread-list}\ \sigma)\ \textit{partner})\ \wedge$
$\textit{error-IPC} = \textit{error-IPC-22-in-PREP-SEND}\ \wedge$
$(\textit{PREP-SEND}_{id}\ \sigma\ (\textit{IPC PREP}\ (\textit{SEND caller partner msg})) =$
$\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$
$\textit{thread-list}\ := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$
$\textit{error-codes}\ := \textit{ERROR-IPC error-IPC-22-in-PREP-SEND}|\!)))\ \longrightarrow$
$(\textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner}\ \sigma\ \textit{msg}\ \wedge$
$\neg\textit{IPC-params-c1}\ ((\textit{the}\ o\ \textit{thread-list}\ \sigma)\ \ \textit{partner})\ \wedge$
$\neg\textit{IPC-params-c2}\ ((\textit{the}\ o\ \textit{thread-list}\ \sigma)\ \ \textit{partner})\ \wedge$
$\textit{error-IPC} = \textit{error-IPC-23-in-PREP-SEND}\ \wedge$
$(\textit{PREP-SEND}_{id}\ \sigma\ (\textit{IPC PREP}\ (\textit{SEND caller partner msg})) =$
$\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$
$\textit{thread-list}\ := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$
$\textit{error-codes}\ := \textit{ERROR-IPC error-IPC-23-in-PREP-SEND}|\!)))$
$)$
by (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def exec-action*$_{id}$*-Mon-prep-fact1-def*
exec-action$_{id}$*-Mon-prep-fact0-def*
split: *errors.split split-if split-if-asm*)

lemma *WAIT-SEND*$_{id}$*-obvious0*:
$(\textit{error-codes}\ (\textit{WAIT-SEND}_{id}\ \sigma\ (\textit{IPC WAIT}\ (\textit{SEND caller partner msg}))) = \textit{NO-ERRORS}) =$
$(\textit{IPC-send-comm-check-st}_{id}\ \textit{caller partner}\ \sigma\ \wedge$
$\textit{IPC-params-c4}\ \ \textit{caller}\ \ \textit{partner}\ \ \wedge$
$\textit{IPC-params-c5 partner}\ \sigma\ \wedge$
$(\textit{WAIT-SEND}_{id}\ \sigma\ (\textit{IPC WAIT}\ (\textit{SEND caller partner msg})) =$
$\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$
$\textit{thread-list}\ := \textit{update-th-waiting caller}\ (\textit{thread-list}\ \sigma),$
$\textit{error-codes}\ := \textit{NO-ERRORS}|\!)))$
by (*auto simp add*: *WAIT-SEND*$_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-SEND*$_{id}$*-obvious1*:
$(\textit{error-codes}\ (\textit{WAIT-SEND}_{id}\ \sigma\ (\textit{IPC WAIT}\ (\textit{SEND caller partner msg}))) = \textit{ERROR-IPC error-IPC}) =$
$((\neg\ \textit{IPC-send-comm-check-st}_{id}\ \textit{caller partner}\ \sigma\ \longrightarrow$
$(\textit{WAIT-SEND}_{id}\ \sigma\ (\textit{IPC WAIT}\ (\textit{SEND caller partner msg})))=$
$\sigma\ (\!|\textit{current-thread} := \textit{caller},$
$\textit{thread-list}\ := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$
$\textit{error-codes}\ := \textit{ERROR-IPC error-IPC-1-in-WAIT-SEND}|\!)\ \wedge$
$\textit{error-IPC} = \textit{error-IPC-1-in-WAIT-SEND})\ \wedge$
$(\textit{IPC-send-comm-check-st}_{id}\ \textit{caller partner}\ \sigma\ \longrightarrow$
$((\neg\ \textit{IPC-params-c4}\ \ \textit{caller}\ \ \textit{partner}\ \longrightarrow$
$(\textit{WAIT-SEND}_{id}\ \sigma\ (\textit{IPC WAIT}\ (\textit{SEND caller partner msg})))=$
$\sigma\ (\!|\textit{current-thread} := \textit{caller},$

```
        thread-list   := update-th-current caller (thread-list σ),
        error-codes   := ERROR-IPC error-IPC-3-in-WAIT-SEND|)) ∧
      error-IPC = error-IPC-3-in-WAIT-SEND) ∧
    (IPC-params-c4 caller partner ⟶
   ((¬ IPC-params-c5 partner σ ⟶
      (WAIT-SEND_id σ (IPC WAIT (SEND caller partner msg)))=
      update-state-wait-send-params5 σ caller ∧
     error-codes (update-state-wait-send-params5 σ caller) = ERROR-IPC error-IPC) ∧
    ¬ IPC-params-c5 partner σ)))))
 by (auto simp add: update-state-wait-send-params5-def WAIT-SEND_id-def
        split: errors.split split-if split-if-asm option.split-asm)
```

lemma *WAIT-SEND$_{id}$-obvious2*:

```
   ¬(error-codes (WAIT-SEND_id σ (IPC WAIT (SEND caller partner msg))) = ERROR-MEM error-IPC)
 by (auto simp add:WAIT-SEND_id-def split: errors.split split-if split-if-asm option.split-asm)
```

lemma *BUF-SEND$_{id}$-obvious0*:

```
   (error-codes (BUF-SEND_id σ (IPC BUF (SEND caller partner msg))) = NO-ERRORS) =
  (IPC-buf-check-st_id caller partner σ ∧
   BUF-SEND_id σ (IPC BUF (SEND caller partner msg)) =
     σ(|current-thread := caller,
      resource    := update-list (resource σ)
                        (zip  ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                        ((own-vmem-adr o the o thread-list σ) partner))
                        (map ((the o (fst o Rep-memory) (resource σ))) msg)),
      thread-list  := update-th-ready caller
                  (update-th-ready partner
                  (thread-list σ)),
       error-codes := NO-ERRORS|))
 by (auto simp add:  BUF-SEND_id-def)
```

lemma *BUF-SEND$_{id}$-obvious1*:

```
   (error-codes (BUF-SEND_id σ (IPC BUF (SEND caller partner msg))) =
    ERROR-IPC error-IPC-1-in-BUF-SEND) =
  (¬ IPC-buf-check-st_id caller partner σ ∧
    BUF-SEND_id σ (IPC BUF (SEND caller partner msg)) =
   σ(|current-thread := caller ,
     thread-list    := update-th-current caller (thread-list σ),
     error-codes    := ERROR-IPC error-IPC-1-in-BUF-SEND|))
 by (auto simp add:  BUF-SEND_id-def)
```

lemma *MAP-SEND$_{id}$-obvious0*:

```
   (error-codes (MAP-SEND_id σ (IPC MAP (SEND caller partner msg))) = error) =
    (error = NO-ERRORS ∧
    MAP-SEND_id σ (IPC MAP (SEND caller partner msg)) =
    σ(|current-thread := caller,
      resource    := init-share-list (resource σ)
                 (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                        ((own-vmem-adr o the o thread-list σ) partner))),
      thread-list  := update-th-ready caller
                 (update-th-ready partner
                 (thread-list σ)),
       error-codes := NO-ERRORS|))
 by (auto simp add:  MAP-SEND_id-def)
```

lemma *DONE-SEND$_{id}$-obvious0*:
(*error-codes* (*exec-action$_{id}$* σ (*IPC DONE* (*SEND caller partner msg*))) = *error*) =
((*exec-action$_{id}$* σ (*IPC DONE* (*SEND caller partner msg*)))= σ $\wedge$ *error-codes* σ = *error*)
by *simp*

lemma *PREP-RECV$_{id}$-obvious0*:
(*error-codes* (*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*))) = *NO-ERRORS*) =
(*exec-action$_{id}$-Mon-prep-fact0 caller partner* σ *msg*$\wedge$
exec-action$_{id}$-Mon-prep-fact1 caller partner σ $\wedge$
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
)
by (*auto simp add*: *PREP-RECV$_{id}$-def exec-action$_{id}$-Mon-prep-fact0-def*
exec-action$_{id}$-Mon-prep-fact1-def
split: *errors.split split-if split-if-asm*)

lemma *PREP-RECV$_{id}$-obvious1*:
(*error-codes* (*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*))) = *ERROR-MEM error-mem*) =
($\neg$((*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)) $\wedge$
error-mem = *not-valid-receiver-addr-in-PREP-RECV* $\wedge$
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*|)))
by (*auto simp add*: *PREP-RECV$_{id}$-def split*: *errors.split split-if split-if-asm*)

lemma *PREP-RECV$_{id}$-obvious2*:
(*error-codes* (*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*))) = *ERROR-IPC error-IPC*) =
($\neg$(*exec-action$_{id}$-Mon-prep-fact0 caller partner* σ *msg* $\wedge$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\wedge$
IPC-params-c2 ((*the o thread-list* σ) *partner*) $\wedge$
$\neg$*IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) $\wedge$
error-IPC = *error-IPC-22-in-PREP-RECV* $\wedge$
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-RECV*|))) $\longrightarrow$
(*exec-action$_{id}$-Mon-prep-fact0 caller partner* σ *msg* $\wedge$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\wedge$
$\neg$*IPC-params-c2* ((*the o thread-list* σ) *partner*) $\wedge$
error-IPC = *error-IPC-23-in-PREP-RECV* $\wedge$
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-RECV*|)))
)
by (*auto simp add*: *PREP-RECV$_{id}$-def exec-action$_{id}$-Mon-prep-fact0-def*
exec-action$_{id}$-Mon-prep-fact1-def exec-action$_{id}$-Mon-prep-fact2-def
split: *errors.split split-if split-if-asm*)

lemma *WAIT-RECV*$_{id}$*-obvious0*:
(*error-codes* (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*))) = *NO-ERRORS*) =
(*IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\wedge$
IPC-params-c4 caller partner $\wedge$
IPC-params-c5 partner σ $\wedge$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*auto simp add*: *WAIT-RECV*$_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV*$_{id}$*-obvious1*:
(*error-codes* (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*))) = *ERROR-IPC error-IPC*) =
(($\neg$ *IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))=
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|) $\wedge$
error-IPC = *error-IPC-1-in-WAIT-RECV*) $\wedge$
(*IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\longrightarrow$
(($\neg$ *IPC-params-c4 caller partner* $\longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))=
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|) $\wedge$
error-IPC = *error-IPC-3-in-WAIT-RECV*) $\wedge$
($\neg$ $\neg$ *IPC-params-c4 caller partner* $\longrightarrow$
(($\neg$ *IPC-params-c5 partner* σ $\longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))=
update-state-wait-recv-params5 σ *caller* $\wedge$
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) $\wedge$
$\neg$ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-recv-params5-def WAIT-RECV*$_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV*$_{id}$*-obvious2*:
$\neg$(*error-codes* (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*))) = *ERROR-MEM error-mem*)
by (*auto simp add*: *WAIT-RECV*$_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *BUF-RECV*$_{id}$*-obvious0*:
(*error-codes* (*BUF-RECV*$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*))) = *NO-ERRORS*) =
(*IPC-buf-check-st*$_{id}$ *caller partner* σ $\wedge$
BUF-RECV$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),

error-codes := *NO-ERRORS*⦈)
by (*auto simp add*: *BUF-RECV*$_{id}$*-def*)

lemma *BUF-RECV*$_{id}$*-obvious1*:
(*error-codes* (*BUF-RECV*$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*))) =
ERROR-IPC error-IPC-1-in-BUF-RECV) =
($\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ $\wedge$
BUF-RECV$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) =
σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*⦈)
by (*auto simp add*: *BUF-RECV*$_{id}$*-def*)

lemma *MAP-RECV*$_{id}$*-obvious0*:
(*error-codes* (*MAP-RECV*$_{id}$ σ (*IPC MAP* (*RECV caller partner msg*))) = *error*) =
(*error* = *NO-ERRORS* $\wedge$
MAP-RECV$_{id}$ σ (*IPC MAP* (*RECV caller partner msg*)) =
σ⦇*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*⦈)
by (*auto simp add*: *MAP-RECV*$_{id}$*-def*)

lemma *DONE-RECV*$_{id}$*-obvious0*:
(*error-codes* (*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*))) = *error*) =
((*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*)))= σ $\wedge$ *error-codes* σ = *error*)
by *simp*

4.17.3 Symbolic Execution Rules for Error Codes field on Pure-level

lemma *PREP-SEND*$_{id}$*-Pure-obvious0*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *NO-ERRORS* $\Longrightarrow$ *P*) $\equiv$
(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* &&&
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ &&&
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈)$\Longrightarrow$ *P*)
find-theorems *name*:*Pure*.
apply (*rule equal-intr-rule*)
apply (*elim meta-impE*)
apply (*drule conjunctionD2*)
apply (*drule conjunctionD2*)
apply (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
exec-action$_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)
done

lemma *PREP-SEND*$_{id}$*-Pure-obvious1*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *ERROR-MEM error-mem* $\Longrightarrow$*P*)

$\equiv$
($\neg$((*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)) &&&
error-mem = *not-valid-sender-addr-in-PREP-SEND* &&&
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|))
$\Longrightarrow$ *P*
)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*: *PREP-SEND*$_{id}$*-def split*: *errors.split split-if split-if-asm*)

lemma *WAIT-SEND*$_{id}$*-Pure-obvious0*:
(*error-codes* (*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*))) = *NO-ERRORS* $\Longrightarrow$ *P*) $\equiv$
(*IPC-send-comm-check-st*$_{id}$ *caller partner* σ &&&
IPC-params-c4 caller partner &&&
IPC-params-c5 partner σ &&&
(*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
$\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
apply (*drule conjunctionD2*)+
by (*auto simp add*:*WAIT-SEND*$_{id}$*-def split*: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-SEND*$_{id}$*-Pure-obvious1*:
(*error-codes* (*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*))) = *ERROR-IPC error-IPC* $\Longrightarrow$ *P*) $\equiv$
(($\neg$ *IPC-send-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
(*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*)))=
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*|) &&&
error-IPC = *error-IPC-1-in-WAIT-SEND*) &&&
(*IPC-send-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
(($\neg$ *IPC-params-c4 caller partner* $\Longrightarrow$
(*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*)))=
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*|) &&&
error-IPC = *error-IPC-3-in-WAIT-SEND*) &&&
(*IPC-params-c4 caller partner* $\Longrightarrow$
(($\neg$ *IPC-params-c5 partner* σ $\Longrightarrow$
(*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*)))= *update-state-wait-send-params5* σ *caller*
&&&
error-codes (*update-state-wait-send-params5* σ *caller*) = *ERROR-IPC error-IPC*) &&&
$\neg$ *IPC-params-c5 partner* σ))))
$\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*simp-all add*: *update-state-wait-send-params5-def WAIT-SEND*$_{id}$*-def*
split: *errors.split split-if split-if-asm option.split option.split-asm*)

lemma *DONE-SEND$_{id}$-Pure-obvious0*:
$(error\text{-}codes\ (exec\text{-}action_{id}\ \sigma\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))) = error \Longrightarrow P) \equiv$
$((exec\text{-}action_{id}\ \sigma\ (IPC\ DONE\ (SEND\ caller\ partner\ msg)))= \sigma \Longrightarrow error\text{-}codes\ \sigma = error \Longrightarrow P)$
by *simp*

lemma *PREP-RECV$_{id}$-Pure-obvious0*:
$(error\text{-}codes\ (PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ ) = NO\text{-}ERRORS \Longrightarrow P) \equiv$
$(exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|))$
$\Longrightarrow P)$
apply (*rule equal-intr-rule*)
by (*auto simp add*: *PREP-RECV$_{id}$-def exec-action$_{id}$-Mon-prep-fact0-def*
exec-action$_{id}$-Mon-prep-fact1-def
split: *errors.split split-if split-if-asm*)

lemma *PREP-RECV$_{id}$-Pure-obvious1*:
$(error\text{-}codes\ (PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))) = ERROR\text{-}MEM\ error\text{-}mem \Longrightarrow P)$
$\equiv$
$(\neg((list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg))\ \&\&\&$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ \&\&\&$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV|))$
$\Longrightarrow P)$
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*: *PREP-RECV$_{id}$-def split*: *errors.split split-if split-if-asm*)

lemma *WAIT-RECV$_{id}$-Pure-obvious0*:
$(error\text{-}codes\ (WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))) = NO\text{-}ERRORS \Longrightarrow P) \equiv$
$(IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \&\&\&$
$IPC\text{-}params\text{-}c4\ \ caller\ \ partner\ \&\&\&$
$IPC\text{-}params\text{-}c5\ partner\ \sigma\ \&\&\&$
$(WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg)) =$
$\sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|))$
$\Longrightarrow P)$
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*:*WAIT-RECV$_{id}$-def split*: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV$_{id}$-Pure-obvious1*:
$(error\text{-}codes\ (WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))) = ERROR\text{-}IPC\ error\text{-}IPC \Longrightarrow P)\equiv$
$((\neg\ IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$

(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))=
 σ(|*current-thread* := *caller* ,
 thread-list := *update-th-current caller* (*thread-list* σ),
 error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|) &&&
 error-IPC = *error-IPC-1-in-WAIT-RECV*) &&&
(*IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
((¬ *IPC-params-c4 caller partner* $\Longrightarrow$
 (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))= σ(|*current-thread* := *caller* ,
 thread-list := *update-th-current caller* (*thread-list* σ),
 error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|) &&&
 error-IPC = *error-IPC-3-in-WAIT-RECV*) &&&
 (*IPC-params-c4 caller partner* $\Longrightarrow$
((¬ *IPC-params-c5 partner* σ $\Longrightarrow$
 (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))= *update-state-wait-recv-params5* σ *caller*
&&&
 error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) &&&
 ¬ *IPC-params-c5 partner* σ)))) $\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*simp-all add*: *update-state-wait-recv-params5-def WAIT-RECV*$_{id}$*-def*
 split: *errors.split split-if split-if-asm list.split-asm*)

lemma *DONE-RECV*$_{id}$*-Pure-obvious0*:
 (*error-codes* (*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*))) = *error* $\Longrightarrow$ *P*) $\equiv$
 ((*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*)))= σ $\Longrightarrow$ *error-codes* σ = *error* $\Longrightarrow$ *P*)
by *simp*

4.17.4 Symbolic Execution of Action Informations Field

lemma *act-info-obvious0*:
 (*act-info* (*th-flag* (*update-state caller* σ *f error*)) =
 (*act-info* (*th-flag* σ))(*caller* := *None*)) =
 (*act-info* (*state*$_{id}$*.th-flag* σ) = (*act-info* (*state*$_{id}$*.th-flag* σ))(*caller* := *None*))
by *simp*

lemma *act-info-obvious1*:
 act-info (*th-flag* (*update-state caller* (*init-act-info caller partner* σ) *f error*)) =
 ((*act-info* (*th-flag* σ)) (*caller* := *None*, *partner* := *None*))
by *simp*

lemma *act-info-obvious2*:
 act-info (*th-flag* (*update-state caller* (*remove-caller-error caller* σ) *f error*)) =
 ((*act-info* (*th-flag* σ)) (*caller* := *None*))
by *simp*

lemma *act-info-prep-send-obvious0*:
 act-info (*th-flag* (*PREP-SEND*$_{id}$ (*init-act-info caller partner* σ)
 (*IPC PREP* (*SEND caller partner msg*)))) =
 (*act-info* (*state*$_{id}$*.th-flag* σ))(*caller* := *None*, *partner* := *None*)

by $(simp\ add: PREP\text{-}SEND_{id}\text{-}def)$

lemma *act-info-prep-send-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(PREP\text{-}SEND_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|\!))$
$(IPC\ PREP\ (SEND\ caller\ partner\ msg)))))$
by $(simp\ add: PREP\text{-}SEND_{id}\text{-}def)$

lemma *act-info-wait-send-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (WAIT\text{-}SEND_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ WAIT\ (SEND\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None)$
by $(simp\ add: WAIT\text{-}SEND_{id}\text{-}def\ split: option.split)$

lemma *act-info-wait-send-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(WAIT\text{-}SEND_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|\!))$
$(IPC\ WAIT\ (SEND\ caller\ partner\ msg)))))$
by $(simp\ add: WAIT\text{-}SEND_{id}\text{-}def\ split: option.split)$

lemma *act-info-buf-send-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (BUF\text{-}SEND_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ BUF\ (SEND\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None)$
by $(simp\ add: BUF\text{-}SEND_{id}\text{-}def)$

lemma *act-info-buf-send-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(BUF\text{-}SEND_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|\!))$
$(IPC\ BUF\ (SEND\ caller\ partner\ msg)))))$
by $(simp\ add: BUF\text{-}SEND_{id}\text{-}def)$

lemma *act-info-done-send-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (exec\text{-}action_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ DONE\ (SEND\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None)$
by *simp*

lemma *act-info-done-send-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(exec\text{-}action_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|\!))$

$(IPC\ DONE\ (SEND\ caller\ partner\ msg)))))$
by *simp*

lemma *act-info-prep-recv-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (PREP\text{-}RECV_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ PREP\ (RECV\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None)$
by $(simp\ add: PREP\text{-}RECV_{id}\text{-}def)$

lemma *act-info-prep-recv-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(PREP\text{-}RECV_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|))$
$(IPC\ PREP\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add: PREP\text{-}RECV_{id}\text{-}def)$

lemma *act-info-wait-recv-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (WAIT\text{-}RECV_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ WAIT\ (RECV\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None)$
by $(simp\ add: WAIT\text{-}RECV_{id}\text{-}def\ split: option.split)$

lemma *act-info-wait-recv-obvious1*:
$(act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(WAIT\text{-}RECV_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|))$
$(IPC\ WAIT\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add: WAIT\text{-}RECV_{id}\text{-}def\ split: option.split)$

lemma *act-info-buf-recv-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (BUF\text{-}RECV_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ BUF\ (RECV\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None)$
by $(simp\ add: BUF\text{-}RECV_{id}\text{-}def)$

lemma *act-info-buf-recv-obvious1*:
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(BUF\text{-}RECV_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(|current\text{-}thread := caller,$
$thread\text{-}list \quad := th\text{-}list,$
$error\text{-}codes \quad := error|))$
$(IPC\ BUF\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add: BUF\text{-}RECV_{id}\text{-}def)$

lemma *act-info-done-recv-obvious0*:
$act\text{-}info\ (th\text{-}flag\ (exec\text{-}action_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ DONE\ (RECV\ caller\ partner\ msg)))) =$
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None)$
by *simp*

lemma *act-info-done-recv-obvious1*:
$(act\text{-}info\ (th\text{-}flag\ \sigma))(caller := None, partner := None) =$
$(act\text{-}info(th\text{-}flag(exec\text{-}action_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(|current\text{-}thread := caller,$
$thread\text{-}list\ := th\text{-}list,$
$error\text{-}codes\ := error|))$
$(IPC\ DONE\ (RECV\ caller\ partner\ msg)))))$
by *simp*

lemmas *atomic-action-normalizer-errors* =
PREP-RECV$_{id}$*-obvious0* *PREP-RECV*$_{id}$*-obvious1* *PREP-RECV*$_{id}$*-obvious2*
PREP-SEND$_{id}$*-obvious0* *PREP-SEND*$_{id}$*-obvious1* *PREP-SEND*$_{id}$*-obvious2*
WAIT-RECV$_{id}$*-obvious0* *WAIT-RECV*$_{id}$*-obvious1* *WAIT-RECV*$_{id}$*-obvious2*
WAIT-SEND$_{id}$*-obvious0* *WAIT-SEND*$_{id}$*-obvious1* *WAIT-SEND*$_{id}$*-obvious2*
BUF-RECV$_{id}$*-obvious0* *BUF-SEND*$_{id}$*-obvious0* *DONE-SEND*$_{id}$*-obvious0*
DONE-RECV$_{id}$*-obvious0*

lemmas *atomic-action-normalizer-errors-Pure* =
PREP-RECV$_{id}$*-Pure-obvious0* *PREP-RECV*$_{id}$*-Pure-obvious1*
PREP-SEND$_{id}$*-Pure-obvious0* *PREP-SEND*$_{id}$*-Pure-obvious1*
WAIT-RECV$_{id}$*-Pure-obvious0* *WAIT-RECV*$_{id}$*-Pure-obvious1*
WAIT-SEND$_{id}$*-Pure-obvious0*
DONE-SEND$_{id}$*-Pure-obvious0*
DONE-RECV$_{id}$*-Pure-obvious0*

lemmas *atomic-action-normalizer-act-info* =
act-info-obvious0 act-info-obvious1 act-info-obvious2
act-info-prep-send-obvious0 act-info-prep-recv-obvious0
act-info-wait-send-obvious0 act-info-wait-recv-obvious0
act-info-buf-send-obvious0 act-info-buf-recv-obvious0
act-info-done-send-obvious0 act-info-done-recv-obvious0

lemmas *atomic-action-normalizer* =
prep-send-obvious prep-recv-obvious wait-send-obvious wait-recv-obvious
buf-send-obvious buf-recv-obvious

lemmas *PREP-SEND*$_{id}$*-normalizer-hyps* =
thread-eq-def
exec-action$_{id}$*-Mon-prep-fact0-def* *exec-action*$_{id}$*-Mon-prep-fact1-def* *IPC-params-c1-def*
IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def is-part-addr-th-mem-def is-part-mem-th-def
is-part-addr-addr-def is-part-mem-def Product-Type.split-beta

lemmas *PREP-RECV*$_{id}$*-normalizer-hyps* =
thread-eq-def Product-Type.split-beta
exec-action$_{id}$*-Mon-prep-fact0-def* *exec-action*$_{id}$*-Mon-prep-fact1-def* *IPC-params-c1-def*
IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def is-part-addr-th-mem-def is-part-mem-th-def
is-part-addr-addr-def is-part-mem-def

lemmas *WAIT-SEND*$_{id}$*-normalizer-hyps* =
thread-eq-def Product-Type.split-beta
IPC-send-comm-check-st$_{id}$*-def* *IPC-params-c4-def* *IPC-buf-check-st*$_{id}$*-def*

lemmas *WAIT-RECV*$_{id}$*-normalizer-hyps* =

thread-eq-def Product-Type.split-beta
IPC-recv-comm-check-st$_{id}$-def IPC-params-c4-def IPC-buf-check-st$_{id}$-def

lemmas *BUF-SEND$_{id}$-normalizer-hyps* =
thread-eq-def Product-Type.split-beta HOL.split-if HOL.split-if-asm
upd-st-res-equiv$_{id}$-def update-th-smm-equiv-def
equiv-def sym-def refl-on-def

lemmas *BUF-RECV$_{id}$-normalizer-hyps* = *BUF-SEND$_{id}$-normalizer-hyps*

lemmas *splitter* =
option.split errors.split
split-if list.split

lemmas *splitter-asm* =
option.split-asm errors.split-asm
split-if-asm list.split-asm

4.18 *IPC* pre-conditions normalizer

lemmas *pre-conditions-defs* =
IPC-params-c1-def IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def IPC-params-c5-def
IPC-send-comm-check-st$_{id}$-def IPC-recv-comm-check-st$_{id}$-def IPC-buf-check-st$_{id}$-def
Product-Type.split-beta is-part-addr-th-mem-def is-part-addr-addr-def

end

theory *IPC-trace-normalizer*

imports *IPC-atomic-action-normalizer*

begin

4.19 The Core Theory for Symbolic Execution of $abort_{lift}$

4.19.1 mbind and ioprog fail

lemma *mbind$_{FailSave}$-ioprog-None1*:
assumes *ioprog-fail*: *ioprog a σ = None*
shows $mbind_{FailSave}$ $(a \# S)$ *ioprog σ = Some* $([], \sigma)$
using *assms*
by(*simp add*: *Product-Type.split-beta*)

lemma *mbind$_{FailSave}$-ioprog-None2*:
assumes *exec-fail*: $mbind_{FailSave}$ $(a \# S)$ *ioprog σ = Some* $([], \sigma)$
shows *ioprog a σ = None*
using *exec-fail*
by(*simp add*: *Product-Type.split-beta split*: *option.split-asm*)

lemma *mbind$_{FailSave}$-ioprog-None*:
(*ioprog a σ = None*) = ($mbind_{FailSave}$ $(a \# S)$ *ioprog σ = Some* $([], \sigma)$)
by (*auto simp*: *mbind$_{FailSave}$-ioprog-None1 mbind$_{FailSave}$-ioprog-None2*)

Here is a collection of generic symbolic execution rules for for our Monad-transformer $abort_{lift}$. They make the specific semantics of aborting atomic actions explicit on the level of a side-calculus.

lemma *abort-None1*:
 assumes *ioprog-fail*: *ioprog a* σ = *None*
 shows *mbind* (*a* # *S*)(*abort*$_{lift}$ *ioprog*) σ =
   Some ([], σ)
oops

lemma *abort-None2*:
 assumes *exec-fail* : *mbind* (*a* # *S*)(*abort*$_{lift}$ *ioprog*) σ =
   Some([], σ)
 shows *ioprog a* σ = *None*
proof (*cases a*)
 case (*IPC ipc-stage ipc-direction*)
 assume *hyp0*: *a* = *IPC ipc-stage ipc-direction*
 then show *?thesis*
 using *assms*
 proof (*cases ipc-stage*)
  case *PREP*
  assume *hyp1*:*ipc-stage* = *PREP*
  then show *?thesis*
  using *assms hyp0 hyp1*
  proof (*cases ipc-direction*)
   case (*SEND thread-id1 thread-id2 adresses*)
   assume *hyp2*: *ipc-direction* = *SEND thread-id1 thread-id2 adresses*
   then show *?thesis*
   using *assms hyp0 hyp1 hyp2*
   by (*simp-all add*: *Product-Type.split-beta*
     split: *split-if-asm option.split-asm errors.split-asm*)
  next
   case (*RECV thread-id1 thread-id2 adresses*)
   assume *hyp2*: *ipc-direction* = *RECV thread-id1 thread-id2 adresses*
   then show *?thesis*
   using *assms hyp0 hyp1 hyp2*
   by (*simp-all add*: *Product-Type.split-beta*
     split: *split-if-asm option.split-asm errors.split-asm*)
  qed
 next
  case *WAIT*
  assume *hyp1*:*ipc-stage* = *WAIT*
  then show *?thesis*
  using *assms hyp0 hyp1*
  proof (*cases ipc-direction*)
   case (*SEND thread-id1 thread-id2 adresses*)
   assume *hyp2*: *ipc-direction* = *SEND thread-id1 thread-id2 adresses*
   then show *?thesis*
   using *assms hyp0 hyp1 hyp2*
   by (*simp-all add*: *Product-Type.split-beta*
     split: *split-if-asm option.split-asm errors.split-asm*)
  next
   case (*RECV thread-id1 thread-id2 adresses*)
   assume *hyp2*: *ipc-direction* = *RECV thread-id1 thread-id2 adresses*
   then show *?thesis*
   using *assms hyp0 hyp1 hyp2*
   by (*simp-all add*: *Product-Type.split-beta*
     split: *split-if-asm option.split-asm errors.split-asm*)
  qed
 next
  case *BUF*

```
  assume hyp1:ipc-stage = BUF
  then show ?thesis
  using assms hyp0 hyp1
  proof (cases ipc-direction)
    case (SEND thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = SEND thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  next
    case (RECV thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = RECV thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  qed
next
  case MAP
  assume hyp1:ipc-stage = MAP
  then show ?thesis
  using assms hyp0 hyp1
  proof (cases ipc-direction)
    case (SEND thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = SEND thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  next
    case (RECV thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = RECV thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  qed
next
  case DONE
  assume hyp1: ipc-stage = DONE
  then show ?thesis
  using assms hyp0 hyp1
  proof (cases ipc-direction)
    case (SEND thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = SEND thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  next
    case (RECV thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = RECV thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
            split: split-if-asm option.split-asm errors.split-asm)
  qed
```

qed
qed

lemma *abort-None'*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ))
and *not-done-act*: *stages* $\neq$ *DONE*
and *ioprog-fail* : *ioprog* (*IPC stages* (*SEND caller partner msg*)) σ = *None*
shows (*abort*$_{lift}$ *ioprog*) (*IPC stages* (*SEND caller partner msg*)) σ = *None*
using *assms*
by(*simp add*: *split*: *p4-stage*$_{ipc}$.*split*,*safe*, *simp-all*)

lemma *abort-None''*:
assumes *not-in-err* : $\bigwedge$*caller*. *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ))
and *not-done-act*: *stages* $\neq$ *DONE*
and *ioprog-fail* : *ioprog* (*IPC stages direction*) σ = *None*
shows (*abort*$_{lift}$ *ioprog*) (*IPC stages direction*) σ = *None*
proof (*cases stages*)
case (*PREP*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *PREP* $\Longrightarrow$
caller $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *SEND thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *PREP* $\Longrightarrow$
caller $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *RECV thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed
next
case (*WAIT*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *WAIT* $\Longrightarrow$
caller $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$

$direction = SEND\ thread\text{-}id1\ thread\text{-}id2\ adresses \Longrightarrow$
$abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
$stages = WAIT \Longrightarrow$
$caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$stages \neq DONE \Longrightarrow$
$ioprog\ (IPC\ stages\ direction)\ \sigma = None \Longrightarrow$
$direction = RECV\ thread\text{-}id1\ thread\text{-}id2\ adresses \Longrightarrow$
$abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
by *simp*
qed
next
case (*BUF*)
then show $abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
$stages = BUF \Longrightarrow$
$caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$stages \neq DONE \Longrightarrow$
$ioprog\ (IPC\ stages\ direction)\ \sigma = None \Longrightarrow$
$direction = SEND\ thread\text{-}id1\ thread\text{-}id2\ adresses \Longrightarrow$
$abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
$stages = BUF \Longrightarrow$
$caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$stages \neq DONE \Longrightarrow$
$ioprog\ (IPC\ stages\ direction)\ \sigma = None \Longrightarrow$
$direction = RECV\ thread\text{-}id1\ thread\text{-}id2\ adresses \Longrightarrow$
$abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
by *simp*
qed
next
case (*MAP*)
then show $abort_{lift}\ ioprog\ (IPC\ stages\ direction)\ \sigma = None$
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
$stages = MAP \Longrightarrow$
$caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$stages \neq DONE \Longrightarrow$
$ioprog\ (IPC\ stages\ direction)\ \sigma = None \Longrightarrow$

direction = *SEND thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *MAP* $\Longrightarrow$
caller $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *RECV thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed
next
case (*DONE*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed

lemma *abort-None0*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *not-done-act*:*stages* $\neq$ *DONE*
and *ioprog-fail* :*ioprog* (*IPC stages* (*SEND caller partner msg*)) σ = *None*
shows (*abort*$_{lift}$ *ioprog*) (*IPC stages* (*SEND caller partner msg*)) σ =
ioprog (*IPC stages* (*SEND caller partner msg*)) σ
using *not-in-err not-done-act ioprog-fail*
by(*simp add*: *split*: *IPC-atomic-actions.p4-stage*$_{ipc}$.*split*,*safe*, *simp-all*)

lemma *abort-None1*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ))
and *ioprog-fail*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *None*
shows *mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some ([], σ)
using *assms*
by *simp*

lemma *mbind-exec-action*$_{id}$*-Mon-None*:
mbind (*a* # *S*) *exec-action*$_{id}$*-Mon* σ $\neq$ *None*
by(*rule Monads.mbind-nofailure*)

lemma *mbind-exec-action*$_{id}$*-Mon-Some*:
$\exists$ *outs* σ'. *mbind* (*a* # *S*) *exec-action*$_{id}$*-Mon* σ = *Some* (*outs*,σ')
by(*insert mbind-exec-action*$_{id}$*-Mon-None*, *auto*)

lemma *mbindef-exec-action*$_{id}$*-Mon-None*:
mbind (*a* # *S*) *exec-action*$_{id}$*-Mon* σ $\neq$ *None*
by(*rule mbind-exec-action*$_{id}$*-Mon-None*)

lemma *mbindef-exec-action*$_{id}$*-Mon-Some*:
$\exists$ *outs* σ'. *mbind* (*a* # *S*) *exec-action*$_{id}$*-Mon* σ = *Some* (*outs*,σ')
by (*auto*, *rule action*$_{ipc}$.*induct*, *simp split*: *option.split*)

4.19.2 Symbolic Execution Rules on PREP stage

lemma *abort-prep-send-obvious0*:
assumes *not-in-err* : $caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
shows $abort_{lift}\ ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(NO\text{-}ERRORS,\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
using *assms*
by *simp*

lemma *abort-prep-send-obvious1*:
assumes *not-in-err* :$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
shows $abort_{lift}\ ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem, (set\text{-}error\text{-}mem\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
using *assms*
by *simp*

lemma *abort-prep-send-obvious2*:
assumes *not-in-err* :$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $abort_{lift}\ ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC, (set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
using *assms*
by *simp*

lemma *abort-prep-send-obvious3*:
assumes *not-in-err* :$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-sucess*:$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
shows
$mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(NO\text{-}ERRORS\#\ fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
proof $(cases\ ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case $(None)$
then show *?thesis*
using *assms*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa, b)$
show *?thesis*
using *assms hyp0 hyp1*
proof $(case\text{-}tac\ aa)$
assume *hyp2*: $aa = NO\text{-}ERRORS$
show *?thesis*
using *assms hyp0 hyp1 hyp2*
by $(simp\ split: option.split)$
next
fix *error-memory*
assume *hyp3*: $aa = ERROR\text{-}MEM\ error\text{-}memory$

show *?thesis*
using *assms hyp0 hyp1 hyp3*
by *simp*
next
fix *error-IPC*
assume *hyp4*: $aa = $ *ERROR-IPC error-IPC*
show *?thesis*
using *assms hyp0 hyp1 hyp4*
by *simp*
qed
qed
qed

lemma *abort-prep-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC PREP*(*SEND caller partner msg*))σ = *Some*(*ERROR-MEM error-mem*,σ')
shows
mbind ((*IPC PREP* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem* #
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ σ' *error-mem msg*))))
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*None*)
then show *?thesis*
using *assms*
by *simp*
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
show *?thesis*
using *assms hyp0 hyp1*
proof (*case-tac aa*)
assume *hyp2*: $aa = $ *NO-ERRORS*
show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
next
fix *error-memory*
assume *hyp3*: $aa = $ *ERROR-MEM error-memory*
show *?thesis*
using *assms hyp0 hyp1 hyp3*
by (*simp split*: *option.split*)
next
fix *error-IPC*
assume *hyp4*: $aa = $ *ERROR-IPC error-IPC*
show *?thesis*
using *assms hyp0 hyp1 hyp4*
by *simp*
qed
qed
qed

lemma *abort-prep-send-obvious5*:
  assumes *not-in-err* : $caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
  and *ioprog-succes*: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
  shows $mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
    $Some(ERROR\text{-}IPC\ error\text{-}IPC\#\ fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
      $(set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))),$
      $snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
      $(set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
  case (*None*)
  assume *hyp0*: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = None$
  then show *?thesis*
  using *assms hyp0*
  by *simp*
next
  case (*Some a*)
  assume *hyp0*:$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ a$
  then show *?thesis*
  using *assms hyp0*
    proof (*cases a*)
     fix *aa b*
     assume *hyp1*: $a = (aa, b)$
     show *?thesis*
     using *assms hyp0 hyp1*
     proof(*case-tac aa*)
      assume *hyp2*: $aa = NO\text{-}ERRORS$
      show *?thesis*
      using *assms hyp0 hyp1 hyp2*
      by *simp*
     next
      fix *error-memory*
      assume *hyp3*: $aa = ERROR\text{-}MEM\ error\text{-}memory$
      show *?thesis*
      using *assms hyp0 hyp1 hyp3*
      by *simp*
     next
      fix *error-IPCa*
      assume *hyp4*: $aa = ERROR\text{-}IPC\ error\text{-}IPCa$
      show *?thesis*
      using *assms hyp0 hyp1 hyp4*
      proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
        $(set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$)
       case (*None*)
       assume *hyp5*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
        $(set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = None$
       show *?thesis*
       using *assms hyp0 hyp1 hyp4 hyp5*
       by *simp*
      next
       case (*Some ab*)
       assume *hyp6*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
        $(set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ ab$
       then show *?thesis*
       using *assms*
       by (*simp add*: *Product-Type.split-beta*)
      qed

```
    qed
  qed
qed

lemma abort-prep-send-obvious6:
 assumes in-err:caller ∈ dom (act-info (th-flag σ))
 shows   abort_lift ioprog (IPC PREP (SEND caller partner msg)) σ =
         Some(get-caller-error caller σ, σ)
 using assms
 by simp

lemma abort-prep-send-obvious7:
 assumes in-err: caller ∈ dom (act-info (th-flag σ))
 shows        mbind ((IPC PREP (SEND caller partner msg))#S) (abort_lift ioprog) σ =
              Some(get-caller-error caller σ#fst(the(mbind S (abort_lift ioprog) σ)),
                                   snd(the(mbind S (abort_lift ioprog) σ)))
 using assms
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
 case (None)
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
 then show ?thesis
 using assms
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa, b)
  then show ?thesis
  using assms hyp0
  by simp
 qed
qed

lemma abort-prep-send-obvious8:
 assumes A: ∀ act σ . ioprog act σ ≠ None
 shows   mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift ioprog) σ =
         (if caller ∈ dom (act-info (th-flag σ))
         then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift ioprog) σ)),
              snd(the(mbind S (abort_lift ioprog) σ)))

         else if  ioprog (IPC PREP (SEND caller partner msg)) σ = Some(NO-ERRORS, σ′)
            then Some(NO-ERRORS#
                   fst(the(mbind S (abort_lift ioprog) (error-tab-transfer caller σ σ′))),
                   snd(the(mbind S (abort_lift ioprog) (error-tab-transfer caller σ σ′))))
          else if ioprog (IPC PREP (SEND caller partner msg)) σ = Some(ERROR-MEM error-mem, σ′)
              then  Some(ERROR-MEM error-mem#
                     fst(the(mbind S (abort_lift ioprog)
                          (set-error-mem-preps caller partner σ σ′ error-mem msg)))
                   ,
                    snd(the(mbind S (abort_lift ioprog)
                        (set-error-mem-preps caller partner σ σ′ error-mem msg))))
              else if ioprog (IPC PREP (SEND caller partner msg)) σ = Some(ERROR-IPC error-IPC, σ′)
              then  Some(ERROR-IPC error-IPC#
                     fst(the(mbind S (abort_lift ioprog)
                          (set-error-ipc-preps caller partner σ σ′ error-IPC msg)))
                   ,
```

```
                    snd(the(mbind S (abort_lift  ioprog)
                        (set-error-ipc-preps caller partner σ σ' error-IPC msg))))
                else if  ioprog (IPC PREP (SEND caller partner msg)) σ = None
                   then Some([], σ)
                   else id (mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift  ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case (None)
  thus ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  thus ?thesis
  using A hyp0
  proof (cases a)
    fix aa b
    assume hyp0 : a = (aa, b)
    thus ?thesis
    using A hyp0
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      thus ?thesis
      by simp
    next
      case (Some ab)
      assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      thus ?thesis
      using A hyp0 hyp1
      proof (cases ab)
        fix ac ba
        assume hyp2: ab = (ac, ba)
        thus ?thesis
        using A hyp0 hyp1 hyp2
        proof (cases mbind_FailSave S (abort_lift ioprog)
          (set-error-ipc-preps caller partner σ σ' error-IPC msg))
          case None
          thus ?thesis
          by simp
        next
          case (Some ad)
          assume hyp3: mbind_FailSave S (abort_lift ioprog) (set-error-ipc-preps caller partner σ σ' error-IPC msg)
=
                    Some ad
          thus ?thesis
          using A hyp0 hyp1 hyp2 hyp3
          proof (cases ad)
            fix ae bb
            assume hyp4: ad = (ae, bb)
            thus ?thesis
            using A hyp0 hyp1 hyp2 hyp3 hyp4
            proof (cases mbind_FailSave S (abort_lift ioprog)
                      (set-error-mem-preps caller partner σ σ' error-mem msg))
              case None
              thus ?thesis
              by simp
            next
              case (Some af)
              assume hyp5: mbind_FailSave S (abort_lift ioprog)
```

(*set-error-mem-preps caller partner* σ σ' *error-mem msg*) = *Some af*
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases af*)
fix *ag bc*
assume *hyp6*: *af* = (*ag*, *bc*)
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
thus *?thesis*
by *simp*
next
case (*Some ah*)
assume *hyp7*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some ah*
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
proof (*cases ah*)
fix *ai bd*
assume *hyp8*: *ah*= (*ai*, *bd*)
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7 hyp8*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious8'*:
mbind ((*IPC PREP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
else (*case ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
Some(*NO-ERRORS*#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
Some(*ERROR-MEM error-mem*#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ σ' *error-mem msg*))))
| *Some*(*ERROR-IPC error-IPC*, σ')⇒
Some(*ERROR-IPC error-IPC*#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)

(*set-error-ipc-preps caller partner* σ σ' *error-IPC msg*))))
| *None* $\Rightarrow$ *Some*([], σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
thus *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
thus *?thesis*
proof –
{**have** *1*: *caller* $\in$ *dom* (*act-info* (*th-flag* σ)) $\longrightarrow$
(*case a of* (*outs*, σ'') $\Rightarrow$ *Some* (*get-caller-error caller* σ # *outs*, σ'')) =
Some (*get-caller-error caller* σ # *fst a*, *snd a*)
by (*simp add*: *Product-Type.split-beta*)
thus *?thesis*
using *hyp0 1*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
{ **case** *None*
thus *?thesis*
using *hyp0 1*
by *simp*
next
case (*Some aa*)
assume *hyp1*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some aa*
thus *?thesis*
using *hyp0 hyp1 1*
proof (*cases aa*)
fix *ab b*
assume *hyp2*: *aa* = (*ab*, *b*)
thus *?thesis*
using *hyp0 hyp1 hyp2 1*
proof (*cases ab*)
case *NO-ERRORS*
assume *hyp3*: *ab* = *NO-ERRORS*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 1*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *b*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp6*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *b*) = *Some ac*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp6 1*
proof (*cases a*)
fix *ad ba*
assume *hyp7*: *a* = (*ad*, *ba*)
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp7 hyp6 1*
proof (*cases ac*)
fix *ae bb*
assume *hyp8*: *ac* = (*ae*, *bb*)
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp7 hyp6 hyp8 1*
by *simp*
qed

qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*: *ab* = *ERROR-MEM error-memory*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *b*)
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *b* = *Some ac*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp6 1*
proof (*cases a*)
fix *ad ba*
assume *hyp7*: *a* = (*ad*, *ba*)
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 1*
proof (*cases ac*)
fix *ae bb*
assume *hyp8*: *ac* = (*ae*, *bb*)
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *b error-memory msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *b error-memory msg*) =
Some af
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 hyp9 1*
proof (*cases af*)
fix *ag bc*
assume *hyp10*: *af* = (*ag*, *bc*)
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 hyp9 hyp10 1*
by *simp*
qed
qed
qed
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp5*: *ab* = *ERROR-IPC error-IPC*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp5 1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *b*)
case *None*
thus *?thesis*
by *simp*

```
      next
       case (Some ac)
       assume hyp6: mbind_FailSave S (abort_lift ioprog) b = Some ac
       thus ?thesis
       using hyp0 hyp1 hyp2 hyp5 hyp6 1
       proof (cases a)
        fix ad ba
        assume hyp7: a = (ad, ba)
        thus ?thesis
        using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 1
        proof (cases ac)
         fix ae bb
         assume hyp8: ac = (ae, bb)
         thus ?thesis
         using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 1
         proof (cases mbind_FailSave S (abort_lift ioprog)
                 (set-error-ipc-preps caller partner σ b error-IPC msg))
          case None
          thus ?thesis
          by simp
         next
          case (Some af)
          assume hyp9: mbind_FailSave S (abort_lift ioprog)
                 (set-error-ipc-preps caller partner σ b error-IPC msg) =
                  Some af
          thus ?thesis
          using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 hyp9 1
          proof (cases af)
           fix ag bc
           assume hyp10:   af = (ag, bc)
           thus ?thesis
           using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 hyp9 hyp10 1
           by simp
          qed
         qed
        qed
       qed
      qed
     qed
    qed
   }qed
  }qed
qed
```

```
lemma abort-prep-send-obvious9:
 fst(the(mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift  ioprog) σ)) =
    (if caller ∈ dom (act-info (th-flag σ))
    then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))
    else (case ioprog (IPC PREP (SEND caller partner msg)) σ of Some(NO-ERRORS, σ')⇒
            NO-ERRORS#
            fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ σ')))
       | Some(ERROR-MEM error-mem, σ')⇒
            ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)
                         (set-error-mem-preps caller partner σ σ' error-mem msg)))
       | Some(ERROR-IPC error-IPC, σ')⇒
            ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
```

(set-error-ipc-preps caller partner σ σ′ error-IPC msg)))
| *None* ⇒ []))
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) *σ*)
case *None*
thus *?thesis*
using *assms*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ*)
case *None*
assume *hyp0*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) *σ* = *None*
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *None*
thus *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) *σ* = *None*
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *Some a*
thus *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
thus *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) *σ* = *Some a*
thus *?thesis*
using *hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ*)
case *None*
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *None*
thus *?thesis*
using *assms hyp1 hyp0*
by *simp*
next
case (*Some aa*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *Some aa*
thus *?thesis*
using *hyp0 hyp2 assms*
proof –
have *1*: (*caller* ∈ *dom* (*act-info* (*th-flag σ*)) ⟶
fst (*the* (*case aa of* (*outs*, *σ′′*) ⇒ *Some* (*get-caller-error caller σ* # *outs*, *σ′′*))) =
get-caller-error caller σ # *fst aa*)
proof (*cases aa*)
fix *a b*
assume *hyp3*: *aa* = (*a*, *b*)
thus *?thesis*
by *simp*
qed
thus *?thesis*
using *1 assms hyp0 hyp2*
proof (*cases a*)
fix *ab b*
assume *hyp3*:*a* = (*ab*, *b*)

thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof (*cases ab*)
  case (*NO-ERRORS*)
  thus *?thesis*
  using *hyp3 1 assms hyp0 hyp2*
  proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller σ b*))
    case *None*
    thus *?thesis*
    by *simp*
  next
    case (*Some ac*)
    assume *hyp4*:*ab* = *NO-ERRORS*
    assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller σ b*) = *Some ac*
    thus *?thesis*
    using *hyp3 hyp4 hyp5 1assms hyp0 hyp2*
    proof (*cases ac*)
      fix *ad ba*
      assume *hyp6*: *ac* = (*ad*, *ba*)
      thus *?thesis*
      using *hyp3 hyp4 hyp5 1 assms hyp0 hyp2*
      by *simp*
    qed
  qed
next
  case (*ERROR-MEM error-memory*)
  thus *?thesis*
  using *hyp3 1 assms hyp0 hyp2*
  proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
          (*set-error-mem-preps caller partner σ b error-memory msg*))
    case *None*
    thus *?thesis*
    by *simp*
  next
    case (*Some ac*)
    assume *hyp7*: *ab* = *ERROR-MEM error-memory*
    assume *hyp8*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
          (*set-error-mem-preps caller partner σ b error-memory msg*) = *Some ac*
    thus *?thesis*
    using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
    proof (*cases ac*)
      fix *ad ba*
      assume *hyp6*: *ac* = (*ad*, *ba*)
      thus *?thesis*
      using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
      by *simp*
    qed
  qed
next
  case (*ERROR-IPC error-IPC*)
  thus *?thesis*
  using *hyp3 1 assms hyp0 hyp2*
  proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
          (*set-error-ipc-preps caller partner σ b error-IPC msg*))
    case *None*
    thus *?thesis*
    by *simp*
  next

case (*Some ac*)
assume *hyp9*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner* σ *b error-IPC msg*) = *Some ac*
assume *hyp10*: *ab* = *ERROR-IPC error-IPC*
thus *?thesis*
using *assms hyp9 hyp10 hyp3 1 hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: *ac* = (*ad*, *ba*)
thus *?thesis*
using *hyp3 hyp9 hyp10 1 assms hyp0 hyp2*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious0*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-succes*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-prep-recv-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success* :*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-prep-recv-obvious2*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some (*ERROR-IPC error-IPC*, (*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-prep-recv-obvious3*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*# *fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*

next
  case (*Some a*)
  assume *hyp0*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
  then show *?thesis*
  using *assms hyp0*
  proof (*cases a*)
    fix *aa b*
    assume *hyp1*: *a* = (*aa*, *b*)
    then show *?thesis*
    using *assms hyp0 hyp1*
    by *simp*
  qed
qed

lemma *abort-prep-recv-obvious4*:
  assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
  and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
      Some(*ERROR-MEM error-mem*, σ')
  shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
    Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
      (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))),
    snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))))
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some a*)
  assume *hyp0*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
  then show *?thesis*
  using *assms hyp0*
  proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
      (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some aa*)
    assume *hyp1*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*) = *Some aa*
    then show *?thesis*
    using *assms hyp0 hyp1*
    proof (*cases aa*)
      fix *ab b*
      assume *hyp2*: *aa* = (*ab*, *b*)
      then show *?thesis*
      using *assms hyp0 hyp1 hyp2*
      by *simp*
    qed
  qed
qed

lemma *abort-prep-recv-obvious5*:
  assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
  and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
      Some(*ERROR-IPC error-IPC*, σ')
  shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
    Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)

(set-error-ipc-prepr caller partner σ σ′ error-IPC msg))),
snd(the(mbind S ($abort_{lift}$ ioprog)
(set-error-ipc-prepr caller partner σ σ′ error-IPC msg))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner σ σ′ error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some aa*)
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner σ σ′ error-IPC msg*) =
Some aa
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases aa*)
fix *ab b*
assume *hyp2*: *aa* = (*ab*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
qed

lemma *abort-prep-recv-obvious6*:
assumes *in-err*: *caller* ∈ *dom* (*act-info* (*th-flag σ*))
shows $abort_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) *σ* =
Some(*get-caller-error caller σ*, *σ*)
using *in-err*
by *simp*

lemma *abort-prep-recv-obvious7*:
assumes *in-err*:*caller* ∈ *dom* (*act-info* (*th-flag σ*))
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) *σ* =
Some(*get-caller-error caller σ*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) *σ*)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) *σ*)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)

then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-prep-recv-obvious8*:
$mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$

$else\ if\ \ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
$then\ Some(NO\text{-}ERRORS\#$
$fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$else\ if\ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
$then\ \ Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$else\ if\ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
$then\ \ Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$else\ if\ \ \ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = None$
$then\ Some([], \sigma)$
$else\ id\ (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma\ )$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa,b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ ab$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*

assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*) =*Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*:*ad* = (*ae*,*bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*:*af* = (*ag*,*bc*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious8'*:
mbind ((*IPC PREP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
else (*case ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')$\Rightarrow$
Some(*NO-ERRORS*#
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)

(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))))
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))))
| *None* $\Rightarrow$ *Some*([], σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a*= (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*,*ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*:*ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*)= *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*:*ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed

qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*: *ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*:*ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*: *ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*:*ad* = (*ae*,*bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious9*:
fst(*the*(*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ))
else (*case ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')$\Rightarrow$
NO-ERRORS#
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
ERROR-MEM error-mem#
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*)))
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
ERROR-IPC error-IPC#
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*)))
| *None* $\Rightarrow$ []))
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case *None*
thus *?thesis*
using *assms*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
assume *hyp0*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *None*
assume *hyp1*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *None*
thus *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *None*
assume *hyp1*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
thus *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
thus *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some a*
thus *?thesis*
using *hyp0*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
assume *hyp1*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *None*
thus *?thesis*
using *assms hyp1 hyp0*
by *simp*
next
case (*Some aa*)
assume *hyp2*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some aa*
thus *?thesis*
using *hyp0 hyp2 assms*
proof –

have *1*: $(caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \longrightarrow$
$fst\ (the\ (case\ aa\ of\ (outs, \sigma'') \Rightarrow Some\ (get\text{-}caller\text{-}error\ caller\ \sigma \ \#\ outs, \sigma''))) =$
$get\text{-}caller\text{-}error\ caller\ \sigma \ \#\ fst\ aa)$
proof (*cases aa*)
fix *a b*
assume *hyp3*: $aa = (a, b)$
thus *?thesis*
by *simp*
qed
thus *?thesis*
using *1 assms hyp0 hyp2*
proof (*cases a*)
fix *ab b*
assume *hyp3*:$a = (ab, b)$
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof (*cases ab*)
case (*NO-ERRORS*)
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ b)$)
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp4*:$ab = NO\text{-}ERRORS$
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ b) = Some\ ac$
thus *?thesis*
using *hyp3 hyp4 hyp5 1assms hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: $ac = (ad, ba)$
thus *?thesis*
using *hyp3 hyp4 hyp5 1 assms hyp0 hyp2*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}prepr\ caller\ partner\ \sigma\ b\ error\text{-}memory\ msg)$)
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp7*: $ab = ERROR\text{-}MEM\ error\text{-}memory$
assume *hyp8*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}prepr\ caller\ partner\ \sigma\ b\ error\text{-}memory\ msg) = Some\ ac$
thus *?thesis*
using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: $ac = (ad, ba)$
thus *?thesis*

using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *b error-IPC msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *b error-IPC msg*) = *Some ac*
assume *hyp10*: *ab* = *ERROR-IPC error-IPC*
thus *?thesis*
using *assms hyp9 hyp10 hyp3 1 hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: *ac* = (*ad*, *ba*)
thus *?thesis*
using *hyp3 hyp9 hyp10 1 assms hyp0 hyp2*
by *simp*
qed
qed
qed
qed
qed
qed
qed

4.19.3 Symbolic Execution rules on WAIT stage

lemma *abort-wait-send-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-wait-send-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-succes*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-waits caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-wait-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =

Some (*ERROR-IPC error-IPC*, (*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-wait-send-obvious3*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-sucess:ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*# *fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
using *assms*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success:ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a*= (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious5*:
assumes *not-in-err*: $caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))), snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious6*:
assumes *in-err*:$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $abort_{lift}\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(get\text{-}caller\text{-}error\ caller\ \sigma, \sigma)$
using *assms*
by *simp*

lemma *abort-wait-send-obvious7*:
assumes *in-err*:$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma = Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)), snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:$a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed

qed

lemma *abort-wait-send-obvious8*:
  mbind ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
    (*if caller* ∈ *dom* (*act-info* (*th-flag* σ))
    then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
        snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

    else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
      then Some(*NO-ERRORS*#
          fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
          snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
      else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
        then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
              (*set-error-mem-waits caller partner* σ σ' *error-mem msg*)))
          ,
          snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
            (*set-error-mem-waits caller partner* σ σ' *error-mem msg*))))
        else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
        then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
              (*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*)))
          ,
          snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
            (*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*))))
        else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *None*
          then Some([], σ)
          else id (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
 case *None*
 then show *?thesis*
 by *simp*
next
 case (*Some a*)
 assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
 then show *?thesis*
 using *assms hyp0*
 proof (*cases a*)
  fix *aa b*
  assume *hyp1*: *a* = (*aa*,*b*)
  then show *?thesis*
  using *assms hyp0 hyp1*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*))
   case *None*
   then show *?thesis*
   by *simp*
  next
   case (*Some ab*)
   assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*) = *Some ab*
   then show *?thesis*
   using *assms hyp0 hyp1 hyp2*
   proof (*cases ab*)
    fix *ac ba*
    assume *hyp3*: *ab* = (*ac*,*ba*)
    then show *?thesis*
    using *assms hyp0 hyp1 hyp2 hyp3*
    proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)

$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some ad)*
assume *hyp4*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) = Some\ ad$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof *(cases ad)*
fix *ae bb*
assume *hyp5*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some af)*
assume *hyp6*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') = Some\ af$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof *(cases af)*
fix *ag bc*
assume *hyp7*: $af = (ag, bc)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious8'*:
$mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$

$else\ (case\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$Some(NO\text{-}ERRORS\#$
$fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$

$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$|\ None \Rightarrow Some([], \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*:$a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case $(Some\ ab)$
assume *hyp2*: $ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof $(cases\ ab)$
fix *ac ba*
assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof $(cases\ ac)$
case *NO-ERRORS*
assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$
assume *hyp7*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba) = Some\ ad$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof $(cases\ ad)$
fix *ae bb*
assume *hyp8*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next

case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious9*:
fst(*the*(*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))

then get-caller-error caller σ#fst(the(mbind S (abort$_{lift}$ ioprog) σ))
else (case ioprog (IPC WAIT (SEND caller partner msg)) σ of Some(NO-ERRORS, σ′)⇒
NO-ERRORS#
fst(the(mbind S (abort$_{lift}$ ioprog) (error-tab-transfer caller σ σ′)))
| Some(ERROR-MEM error-mem, σ′)⇒
ERROR-MEM error-mem#fst(the(mbind S (abort$_{lift}$ ioprog)
(set-error-mem-waits caller partner σ σ′ error-mem msg)))
| Some(ERROR-IPC error-IPC, σ′)⇒
ERROR-IPC error-IPC#fst(the(mbind S (abort$_{lift}$ ioprog)
(set-error-ipc-waits caller partner σ σ′ error-IPC msg)))
| None ⇒ []))
by (*simp split*: *option.split errors.split*, *auto*)

lemma *abort-wait-recv-obvious0*:
assumes *not-in-err :caller ∉ dom (act-info (th-flag σ))*
and *ioprog-success:ioprog (IPC WAIT (RECV caller partner msg)) σ = Some(NO-ERRORS, σ′)*
shows *abort$_{lift}$ ioprog (IPC WAIT (RECV caller partner msg)) σ = Some(NO-ERRORS, (error-tab-transfer caller σ σ′))*
using *assms*
by *simp*

lemma *abort-wait-recv-obvious1*:
assumes *not-in-err :caller ∉ dom (act-info (th-flag σ))*
and *ioprog-success: ioprog (IPC WAIT (RECV caller partner msg)) σ =*
Some(ERROR-MEM error-mem, σ′)
shows *abort$_{lift}$ ioprog (IPC WAIT (RECV caller partner msg)) σ =*
Some (ERROR-MEM error-mem, (set-error-mem-waitr caller partner σ σ′ error-mem msg))
using *assms*
by *simp*

lemma *abort-wait-recv-obvious2*:
assumes *not-in-err :caller ∉ dom (act-info (th-flag σ))*
and *ioprog-success:ioprog (IPC WAIT (RECV caller partner msg)) σ =*
Some(ERROR-IPC error-IPC, σ′)
shows *abort$_{lift}$ ioprog (IPC WAIT (RECV caller partner msg)) σ =*
Some (ERROR-IPC error-IPC, (set-error-ipc-waitr caller partner σ σ′ error-IPC msg))
using *assms*
by *simp*

lemma *abort-wait-recv-obvious3*:
assumes *not-in-err: caller ∉ dom (act-info (th-flag σ))*
and *ioprog-success:ioprog (IPC WAIT (RECV caller partner msg)) σ = Some(NO-ERRORS, σ′)*
shows *mbind ((IPC WAIT (RECV caller partner msg))#S) (abort$_{lift}$ ioprog) σ =*
Some(NO-ERRORS# fst(the(mbind S (abort$_{lift}$ ioprog) (error-tab-transfer caller σ σ′))),
snd(the(mbind S (abort$_{lift}$ ioprog) (error-tab-transfer caller σ σ′))))
proof (*cases mbind$_{FailSave}$ S (abort$_{lift}$ ioprog) (error-tab-transfer caller σ σ′)*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind$_{FailSave}$ S (abort$_{lift}$ ioprog) (error-tab-transfer caller σ σ′) = Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*

assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious4*:
assumes *not-in-err* :$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
proof $(cases\ \ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious5*:
assumes *not-in-err*: $caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*

assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious6*:
assumes *in-err*:$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $abort_{lift}\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma, \sigma)$
using *assms*
by *simp*

lemma *abort-wait-recv-obvious7*:
assumes *in-err*:$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a= (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious8*:
$mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$

$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
$then\ Some(NO\text{-}ERRORS\#$
$fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
$then\ Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
$then\ Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None$

then Some$([], \sigma)$
else id $(mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma)$
)
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa,b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ab)$
assume *hyp2*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ ab$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof $(cases\ ab)$
fix *ac ba*
assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$
assume *hyp4*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) = Some\ ad$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ ad)$
fix *ae bb*
assume *hyp5*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ af)$
assume *hyp6*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') = Some\ af$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*

proof (*cases af*)
fix *ag bc*
assume *hyp7*: *af* = (*ag*, *bc*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious8′*:
mbind ((*IPC WAIT* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* ∈ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
else (*case ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ′)⇒
Some(*NO-ERRORS*#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ′))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ′))))
| *Some*(*ERROR-MEM error-mem*, σ′)⇒
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ′ *error-mem msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ′ *error-mem msg*))))
| *Some*(*ERROR-IPC error-IPC*, σ′)⇒
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ σ′ *error-IPC msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ σ′ *error-IPC msg*))))
| *None* ⇒ *Some*([], σ)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*

```
    by simp
  next
    case (Some ab)
    assume hyp2: ioprog (IPC WAIT (RECV caller partner msg)) σ = Some ab
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    proof (cases ab)
      fix ac ba
      assume hyp3: ab = (ac,ba)
      then show ?thesis
      using assms hyp0 hyp1 hyp2 hyp3
      proof (cases ac)
        case NO-ERRORS
        assume hyp4: ac = NO-ERRORS
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp7: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7
          proof (cases ad)
            fix ae bb
            assume hyp8: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8
            by simp
          qed
        qed
      next
        case (ERROR-MEM error-memory)
        assume hyp5:ac = ERROR-MEM error-memory
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp5
        proof (cases mbind_FailSave S (abort_lift ioprog)
                 (set-error-mem-waitr caller partner σ ba error-memory msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp9: mbind_FailSave S (abort_lift ioprog)
                 (set-error-mem-waitr caller partner σ ba error-memory msg) = Some ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9
          proof (cases ad)
            fix ae bb
            assume hyp10: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10
            by simp
          qed
        qed
      next
```

case (*ERROR-IPC error-IPC*)
assume *hyp6:ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious9*:
fst(*the*(*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ))
else (*case ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
NO-ERRORS#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
ERROR-MEM error-mem#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ' *error-mem msg*)))

| *Some*(*ERROR-IPC error-IPC*, σ')⇒
ERROR-IPC error-IPC#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (
set-error-ipc-waitr caller partner σ σ' *error-IPC msg*)))

| *None* ⇒ []))
by (*simp split*: *option.split errors.split*,*auto*)

4.19.4 Symbolic Execution rules on BUF stage

lemma *abort-buf-send-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-buf-send-obvious1*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-buf-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some (*ERROR-IPC error-IPC*, (*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-buf-send-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next

case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*)= *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-succes* : *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-dones caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-dones caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious6*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-buf-send-obvious7*:
assumes *in-err*: *caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*

by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious8*:
assumes *A*: $\forall$ *act* σ . *ioprog act* $\sigma \neq$ *None*
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*)))

,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*)))

,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *None*
then Some([], σ)
else id (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*

by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*) = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*: *af* = (*ag*, *bc*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious8'*:
mbind ((*IPC BUF* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),

$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$

else $(case\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS, \sigma')\Rightarrow$
$Some(NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$|\ \ None \Rightarrow Some([], \sigma)))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*:$a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case $(Some\ ab)$
assume *hyp2*: $ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof $(cases\ ab)$
fix *ac ba*
assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof $(cases\ ac)$
case *NO-ERRORS*
assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$

assume *hyp7*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed

qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious9*:
$fst(the(mbind\ (IPC\ BUF\ (SEND\ caller\ partner\ msg)\#S)(abort_{lift}\ \ ioprog)\ \sigma)) =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma))$
$else\ (case\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS, \sigma')\Rightarrow$
$NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$

$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$

$|\ None \Rightarrow []))$
by $(simp\ split{:}\ option.split\ errors.split, auto)$

lemma *abort-buf-recv-obvious0*:
assumes *not-in-err* $:caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*$:ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(NO\text{-}ERRORS, \sigma')$
shows $abort_{lift}\ \ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
using *assms*
by *simp*

lemma *abort-buf-recv-obvious1*:
assumes *not-in-err* $:caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success*$:ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
shows $abort_{lift}\ \ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem, (set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
using *assms*
by *simp*

lemma *abort-buf-recv-obvious2*:
assumes *not-in-err*$:\ caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-succes*$:\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $abort_{lift}\ \ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC, (set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
using *assms*
by *simp*

lemma *abort-buf-recv-obvious3*:
assumes *not-in-err* $:caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-success* $:\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$

shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious4*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))))
using *assms*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))),

snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious6*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows *abort*$_{lift}$ *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-buf-recv-obvious7*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious8*:
mbind ((*IPC BUF* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
else if ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))))
else if ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))))
else if ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *None*
then Some([], σ)
else id (*mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*) = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*) = *Some ad*

then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') = Some\ af$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*: $af = (ag, bc)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious8'*:
$mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$
$else\ (case\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS, \sigma')\Rightarrow$
$Some(NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$|\ None \Rightarrow Some([], \sigma)))$
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma$)
case *None*
then show *?thesis*

by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*

next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious9*:
fst(*the*(*mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ))
else (*case ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
NO-ERRORS#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
ERROR-MEM error-mem#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*)))
| *Some*(*ERROR-IPC error-IPC*, σ')⇒
ERROR-IPC error-IPC#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*)))

| *None* ⇒ []))
by(*simp split*: *option.split errors.split*,*auto*)

4.19.5 Symbolic Execution Rules on MAP stage

lemma *abort-map-send-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-map-send-obvious1*:
assumes *not-in-err* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-maps caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-map-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some (*ERROR-IPC error-IPC*, (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-map-send-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =

$Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
shows $mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some a)*
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof *(cases a)*
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious5*:
assumes *not-in-err* :$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
and *ioprog-succes* : $ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}dones\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some a)*
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}dones\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof *(cases a)*
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious6*:
assumes *in-err*:$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $abort_{lift}\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma =$

$Some(get\text{-}caller\text{-}error\ caller\ \sigma, \sigma)$
using *assms*
by *simp*

lemma *abort-map-send-obvious7*:
assumes *in-err*: $caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
shows $mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case *(Some a)*
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof *(cases a)*
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious8*:
assumes *A*: $\forall\ act\ \sigma\ .\ ioprog\ act\ \sigma \neq None$
shows $mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma)))$

$else\ if\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
$then\ Some(NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$else\ if\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
$then\ Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$else\ if\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
$then\ Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$

,
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$else\ if\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = None$
$then\ Some([], \sigma)$
$else\ id\ (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)\ \sigma))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case *(Some a)*

assume *hyp0*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
  fix *aa b*
  assume *hyp1*: *a* = (*aa*,*b*)
  then show *?thesis*
  using *assms hyp0 hyp1*
  proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
      (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ab*)
    assume *hyp2*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
      (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*) = *Some ab*
    then show *?thesis*
    using *assms hyp0 hyp1 hyp2*
    proof (*cases ab*)
      fix *ac ba*
      assume *hyp3*: *ab* = (*ac*,*ba*)
      then show *?thesis*
      using *assms hyp0 hyp1 hyp2 hyp3*
      proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-mem-maps caller partner* σ σ' *error-mem msg*))
        case *None*
        then show *?thesis*
        by *simp*
      next
        case (*Some ad*)
        assume *hyp4*:$mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
          (*set-error-mem-maps caller partner* σ σ' *error-mem msg*) = *Some ad*
        then show *?thesis*
        using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
        proof (*cases ad*)
          fix *ae bb*
          assume *hyp5*: *ad* = (*ae*, *bb*)
          then show *?thesis*
          using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
          proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
            case *None*
            then show *?thesis*
            by *simp*
          next
            case (*Some af*)
            assume *hyp6*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
            then show *?thesis*
            using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
            proof (*cases af*)
              fix *ag bc*
              assume *hyp7*: *af* = (*ag*, *bc*)
              then show *?thesis*
              using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
              by *simp*
            qed
          qed
        qed

qed
qed
qed
qed
qed

lemma *abort-map-send-obvious8′*:
$mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$

$else\ (case\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$Some(NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
$,$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
$,$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$|\ \ None \Rightarrow Some([], \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:$a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*

using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
  case *NO-ERRORS*
  assume *hyp4*: $ac = $ *NO-ERRORS*
  then show *?thesis*
  using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)
    assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
    then show *?thesis*
    using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
    proof (*cases ad*)
      fix *ae bb*
      assume *hyp8*: $ad = (ae, bb)$
      then show *?thesis*
      using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
      by *simp*
    qed
  qed
next
  case (*ERROR-MEM error-memory*)
  assume *hyp5*:*ac* = *ERROR-MEM error-memory*
  then show *?thesis*
  using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-maps caller partner* σ *ba error-memory msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)
    assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
    then show *?thesis*
    using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
    proof (*cases ad*)
      fix *ae bb*
      assume *hyp10*: $ad = (ae, bb)$
      then show *?thesis*
      using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
      by *simp*
    qed
  qed
next
  case (*ERROR-IPC error-IPC*)
  assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
  then show *?thesis*
  using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next

case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-send-obvious9*:
fst(*the*(*mbind* (*IPC MAP* (*SEND caller partner msg*)#*S*)($abort_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ))
else (*case ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
NO-ERRORS#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
ERROR-MEM error-mem#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ σ' *error-mem msg*)))

| *Some*(*ERROR-IPC error-IPC*, σ')⇒
ERROR-IPC error-IPC#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)))

| *None* ⇒ []))
by (*simp split*: *option.split errors.split*,*auto*)

lemma *abort-map-recv-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-map-recv-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-map-recv-obvious2*:
assumes *not-in-err*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-succes*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some (*ERROR-IPC error-IPC*, (*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-map-recv-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success* : *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious4*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))))
using *assms*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*

using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious6*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows *abort*$_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-map-recv-obvious7*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)

then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious8*:
mbind ((*IPC MAP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *None*
then Some([], σ)
else id (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*) = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*

assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*: $af = (ag, bc)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-recv-obvious8'*:
mbind ((*IPC MAP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
else (*case ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)))
,

snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))))
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))))
| *None* $\Rightarrow$ *Some*([], σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*

```
      qed
    qed
  next
    case (ERROR-MEM error-memory)
    assume hyp5:ac = ERROR-MEM error-memory
    then show ?thesis
    using assms hyp0 hyp1 hyp2 hyp3 hyp5
    proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-mapr caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp9: mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-mapr caller partner σ ba error-memory msg) = Some ad
      then show ?thesis
      using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9
      proof (cases ad)
        fix ae bb
        assume hyp10: ad = (ae, bb)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10
        by simp
      qed
    qed
  next
    case (ERROR-IPC error-IPC)
    assume hyp6:ac = ERROR-IPC error-IPC
    then show ?thesis
    using assms hyp0 hyp1 hyp2 hyp3 hyp6
    proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-ipc-mapr caller partner σ ba error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp11: mbind_FailSave S (abort_lift ioprog)
              (set-error-ipc-mapr caller partner σ ba error-IPC msg) = Some ad
      then show ?thesis
      using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11
      proof (cases ad)
        fix ae bb
        assume hyp12: ad = (ae, bb)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
        by simp
      qed
    qed
  qed
 qed
qed
qed
qed
```

lemma *abort-map-recv-obvious9*:
$fst(the(mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma)) =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma))$
$else\ (case\ ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma\ of\ Some(NO\text{-}ERRORS,\ \sigma')\Rightarrow$
$NO\text{-}ERRORS\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma')\Rightarrow$
$ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')\Rightarrow$
$ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
$|\ \ None \Rightarrow []))$
by$(simp\ split\text{:}\ option.split\ errors.split,auto)$

4.19.6 Symbolic Execution Rules rules on DONE stage

lemma *abort-done-send-obvious0*:
assumes *not-in-err*:
$caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
assumes $ioprog\text{-}success\text{:}ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
shows $abort_{lift}\ \ ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ \sigma)$
using *assms*
by $(simp\ split\text{:}option.split)$

lemma *abort-done-send-obvious1*:
assumes $not\text{-}in\text{-}err\text{:}caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and $exec\text{-}success\text{:}\ mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$
$Some(out'',\sigma'')$
and $ioprog\text{-}success\text{:}ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
and $exec\text{-}success'\text{:}mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma = Some(out',\sigma')$
shows $\sigma' = \sigma''$
using *assms*
by *auto*

lemma *abort-done-send-obvious2*:
assumes $not\text{-}in\text{-}err\text{:}caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and $exec\text{-}success\text{:}\ mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$
$Some(out'',\sigma'')$
and $ioprog\text{-}success\text{:}ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
shows $mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma = Some(out',\sigma') \Longrightarrow out'' = (NO\text{-}ERRORS\#out')$
using *assms*
by *auto*

lemma *abort-done-send-obvious3*:
assumes $in\text{-}err\text{:}caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
shows $abort_{lift}\ \ ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma,\ remove\text{-}caller\text{-}error\ caller\ \sigma)$
using *assms*
by *simp*

lemma *abort-done-send-obvious4*:
assumes $in\text{-}err\text{:}caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and $exec\text{-}success\text{:}mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$
$Some(out'',\sigma'')$
shows $hd\ out'' = get\text{-}caller\text{-}error\ caller\ \sigma$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma))$

case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp*, *elim conjE*, *simp add*: *HOL.eq-sym-conv*)
qed
qed

lemma *abort-done-send-obvious5*:
assumes *in-err*:*caller* $\in$ *dom* ((*act-info o th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ = *Some*(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (σ(|*th-flag* := (*th-flag* σ) (|*act-info* := ((*act-info* (*th-flag* σ)) (*caller* := *None*))|)|)) = *Some*(*out'*,σ')
shows *out''* = *the* (((*act-info o th-flag*) σ) *caller*) #*out'*
using *assms*
by *simp*

lemma *abort-done-send-obvious6*:
assumes *in-err*:*caller* $\in$ *dom* (*act-info* (*th-flag* σ))
and *exec-success*: *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ = *Some*(*out''*,σ'')
and *exec-success'*: *mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) = *Some*(*out'*,σ')
shows $\sigma'' = \sigma'$
using *assms*
by *simp*

lemma *abort-done-send-obvious7*:
assumes *exec-success* : *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ = *Some* (*out'*,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ $\neq$ *None*
shows(*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') $\Rightarrow$ $\sigma' = \sigma''$)
else (*case mbind S* ($abort_{lift}$ *ioprog*) σ
of Some (*out''*,σ'') $\Rightarrow$ $\sigma' = \sigma''$))
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
assume *hyp0*: *caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next

case (*Some a*)
assume *hyp1*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case *False*
assume *hyp0*: *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *auto*
qed
qed
qed

lemma *abort-done-send-obvious8*:
assumes *execu-success* : *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
Some (*out'*,σ')
and *ioprog-success*: *ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*
shows
(*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') $\Rightarrow$ *out'* = (*get-caller-error caller* σ #*out''*))
else (*case mbind S* ($abort_{lift}$ *ioprog*) σ
of Some (*out''*,σ'') $\Rightarrow$ *out'* = (*NO-ERRORS*#*out''*)))
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
assume *hyp0*: *caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next

case (*Some a*)
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case *False*
assume *hyp0* : *caller* $\notin$ *dom* (*act-info* (*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *auto*
qed
qed
qed

lemma *abort-done-send-obvious9*:
mbind ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then Some (*get-caller-error caller* σ#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))))
else (*case ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ *of None* $\Rightarrow$ *Some* ([], σ)
| *Some* (*out'*, σ') $\Rightarrow$
Some (*NO-ERRORS*# (*fst o the*)(*mbind S* ($abort_{lift}$ *ioprog*) σ),
(*snd o the*)(*mbind S* ($abort_{lift}$ *ioprog*) σ))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *hyp0*

proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*simp add*: *split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-send-obvious10*:
(*fst o the*)(*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then get-caller-error caller σ#
(*fst o the*)(*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
else
(*case ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ *of*
None $\Rightarrow$ []
| *Some* (*out'*, σ')$\Rightarrow$ *NO-ERRORS*# (*fst o the*)(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*

proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*simp split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-recv-obvious0*:
assumes *no-inerr*:$caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and *ioprog-success*:$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
shows $abort_{lift}\ ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma)$
using *assms*
by (*simp split*:*option.split*)

lemma *abort-done-recv-obvious1*:
assumes *not-in-err*:$caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and *exec-success*:$mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma = Some(out'',\sigma'')$
and *ioprog-success*:$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
shows $mbind\ S\ (abort_{lift}\ ioprog)\ \sigma = Some(out',\sigma') \Longrightarrow \sigma' = \sigma''$
using *assms*
by *auto*

lemma *abort-done-recv-obvious2*:
assumes *not-inerr* :$caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and *exec-success* :$mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma = Some(out'',\sigma'')$
and *ioprog-success*:$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
shows $mbind\ S\ (abort_{lift}\ ioprog)\ \sigma = Some(out',\sigma') \Longrightarrow out'' = (NO\text{-}ERRORS\#out')$
using *assms*
by *auto*

lemma *abort-done-recv-obvious3*:
assumes *in-err*: $caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
shows $abort_{lift}\ ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma = Some(get\text{-}caller\text{-}error\ caller\ \sigma, remove\text{-}caller\text{-}error\ caller\ \sigma)$
using *assms*
by *simp*

lemma *abort-done-recv-obvious4*:
assumes *in-err*:$caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
and *exec-success*:$mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ ioprog)\ \sigma = Some(out'',\sigma'')$
shows $hd\ out'' = get\text{-}caller\text{-}error\ caller\ \sigma$
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma) = Some\ a$
then show *?thesis*

using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp*, *elim conjE*, *simp add*: *HOL.eq-sym-conv*)
qed
qed

lemma *abort-done-recv-obvious5*:
assumes *in-err*:*caller* ∈ *dom* ((*act-info o th-flag*) σ)
and *exec-success*: *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) = *Some*(*out'*,σ')
shows *out''* = (*get-caller-error caller* σ #*out'*)
using *assms*
by *simp*

lemma *abort-done-recv-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*act-info o th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some(*out'*,σ')
shows σ'' = σ'
using *assms*
by *simp*

lemma *abort-done-recv-obvious7*:
assumes *exec-success*: *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
Some (*out'*,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None*
shows (*if caller* ∈ *dom* ((*act-info o th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') ⇒ σ' = σ'')
else (*case mbind S* ($abort_{lift}$ *ioprog*) σ
of Some (*out''*,σ'') ⇒ σ' = σ''))
proof (*cases caller* ∈ *dom* ((*act-info o th-flag*) σ))
case *True*
assume *hyp0*: *caller* ∈ *dom* ((*act-info o th-flag*) σ)
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: $a = (aa, b)$
then show *?thesis*

using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case *False*
assume *hyp0*: *caller* $\notin$ *dom* ((*act-info o th-flag*) σ)
then show *?thesis*
using *assms hyp0*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *auto*
qed
qed
qed

lemma *abort-done-recv-obvious8*:
assumes *exec-success* : *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
Some (*out*$'$,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*
shows (*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then (*case mbind S* (*abort*$_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out*$''$,σ'') $\Rightarrow$ *out*$'$ = (*get-caller-error caller* σ #*out*$''$))
else (*case mbind S* (*abort*$_{lift}$ *ioprog*) σ
of *Some* (*out*$''$,σ'') $\Rightarrow$ *out*$'$ = (*NO-ERRORS*#*out*$''$)))
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
assume *hyp0*: *caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*

using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case *False*
assume *hyp0*: $caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
then show *?thesis*
using *assms hyp0*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case $(Some\ a)$
assume *hyp1*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0 hyp1*
proof $(cases\ a)$
fix *aa b*
assume *hyp2*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *auto*
qed
qed
qed

lemma *abort-done-recv-obvious9*:
$mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\ \sigma)$
$then\ Some\ ((get\text{-}caller\text{-}error\ caller\ \sigma\#$
$fst(the(mbind\ S\ (abort_{lift}\ ioprog)\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))),$
$snd(the(mbind\ S\ (abort_{lift}\ ioprog)\ (remove\text{-}caller\text{-}error\ caller\ \sigma))))$
$else(case\ ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma\ of\ None \Rightarrow Some\ ([], \sigma)$
$|\ Some\ (out', \sigma') \Rightarrow$
$Some\ (NO\text{-}ERRORS\#\ (fst\ o\ the)(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma),$
$(snd\ o\ the)(mbind\ S\ (abort_{lift}\ ioprog)\ \sigma))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma) =$
$Some\ a$
then show *?thesis*
using *hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*

next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*simp split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-recv-obvious10*:
fst(*the*(*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*) σ)
then (*get-caller-error caller* σ#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))))
else
(*case ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ *of*
None $\Rightarrow$ []
| *Some* (*out*′, σ′)$\Rightarrow$ *NO-ERRORS*# (*fst o the*)(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
by (*simp split*: *option.split*)

lemmas *trace-normalizer-errors-case* =
abort-prep-send-obvious9 abort-prep-recv-obvious9 abort-wait-send-obvious9
abort-wait-recv-obvious9 abort-buf-send-obvious9 abort-buf-recv-obvious9
abort-done-send-obvious10 abort-done-recv-obvious10

end

theory *IPC-symbolic-exec-rewriting*
imports *IPC-trace-normalizer*
begin

4.20 Rewriting Rules for Symbolic Execution of Sequence Test Scheme

4.20.1 Symbolic Execution Rules for PREP stage

lemma *abort-prep-send-obvious10*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*)σ)
then (σ $\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ′) $\Rightarrow$
(*error-tab-transfer caller* σ σ′) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*))
| *Some*(*ERROR-MEM error-mem*, σ′)$\Rightarrow$
((*set-error-mem-preps caller partner* σ σ′ *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))
| *Some*(*ERROR-IPC error-IPC*, σ′)$\Rightarrow$
((*set-error-ipc-preps caller partner* σ σ′ *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))

| None ⇒ ($\sigma \models$ (*P* [])))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*

proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
  (*set-error-mem-preps caller partner* σ *ba error-memory msg*))
 case *None*
 then show *?thesis*
 by *simp*
next
 case (*Some ad*)
 assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
  (*set-error-mem-preps caller partner* σ *ba error-memory msg*) = *Some ad*
 then show *?thesis*
 using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
 proof (*cases ad*)
  fix *ae bb*
  assume *hyp6*: *ad* = (*ae*, *bb*)
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
  by(*simp add*: *valid-SE-def bind-SE-def*)
 qed
qed
next
 case (*ERROR-IPC error-IPC*)
 assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
 then show *?thesis*
 using *hyp0 hyp1 hyp2 hyp3 hyp4*
 proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
  (*set-error-ipc-preps caller partner* σ *ba error-IPC msg*))
  case *None*
  then show *?thesis*
  by *simp*
 next
  case (*Some ad*)
  assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
   (*set-error-ipc-preps caller partner* σ *ba error-IPC msg*) = *Some ad*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
  proof (*cases ad*)
   fix *ae bb*
   assume *hyp6*: *ad* = (*ae*, *bb*)
   then show *?thesis*
   using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
   by(*simp add*: *valid-SE-def bind-SE-def*)
  qed
 qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious12*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*)σ)
then ($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ *of*
 Some(*NO-ERRORS*, σ') $\Rightarrow$
 ((*error-tab-transfer caller* σ σ') $\models$
 (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))$\wedge$
 (((*act-info o th-flag*) σ) *caller* = *None*) $\wedge$

$((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))\ caller \wedge$
$(th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$((set\text{-}error\text{-}mem\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner)$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner)$

$|\ None \Rightarrow (\sigma \models (P\ [])))))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume $hyp0$: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof $(cases\ a)$
fix *aa b*
assume $hyp1$: $a = (aa\ , b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by $(simp\ add\text{:}\ valid\text{-}SE\text{-}def\ bind\text{-}SE\text{-}def)$
next
case $(Some\ ab)$
assume $hyp2$: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof $(cases\ ab)$
fix *ac ba*
assume $hyp3$:$ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof $(cases\ ac)$
case *NO-ERRORS*
assume $hyp4$: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$

```
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show  ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: valid-SE-def bind-SE-def)
      qed
    qed
  next
    case (ERROR-MEM error-memory)
    assume hyp4:ac = ERROR-MEM error-memory
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog)
                (set-error-mem-preps caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog)
                (set-error-mem-preps caller partner σ ba error-memory msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show  ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(simp add: valid-SE-def bind-SE-def)
      qed
    qed
  next
    case (ERROR-IPC error-IPC)
    assume hyp4:ac = ERROR-IPC error-IPC
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog)
                (set-error-ipc-preps caller partner σ ba error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog)
                (set-error-ipc-preps caller partner σ ba error-IPC msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
```

assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious10''*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)); P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))) \wedge$
$((\forall\, a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS, \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}memory, \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}preps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ , b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac, ba)$
then show *?thesis*

```
  using hyp0 hyp1 hyp2 hyp3
  proof (cases ac)
   case NO-ERRORS
   assume hyp4: ac = NO-ERRORS
   then show ?thesis
   using  hyp0 hyp1 hyp2 hyp3 hyp4
   proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
    case None
    then show ?thesis
    by simp
   next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
     fix ae bb
     assume hyp6: ad = (ae, bb)
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
     by(simp add: valid-SE-def bind-SE-def )
    qed
   qed
  next
   case (ERROR-MEM error-memory)
   assume hyp4:ac = ERROR-MEM error-memory
   then show ?thesis
   using  hyp0 hyp1 hyp2 hyp3 hyp4
   proof (cases mbind_FailSave S (abort_lift ioprog)
          (set-error-mem-preps caller partner σ ba error-memory msg))
    case None
    then show ?thesis
    by simp
   next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift ioprog)
          (set-error-mem-preps caller partner σ ba error-memory msg) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
     fix ae bb
     assume hyp6: ad = (ae, bb)
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
     by(simp add: valid-SE-def bind-SE-def)
    qed
   qed
  next
   case (ERROR-IPC error-IPC)
   assume hyp4:ac = ERROR-IPC error-IPC
   then show ?thesis
   using  hyp0 hyp1 hyp2 hyp3 hyp4
   proof (cases mbind_FailSave S (abort_lift ioprog)
          (set-error-ipc-preps caller partner σ ba error-IPC msg))
    case None
    then show ?thesis
    by simp
   next
```

case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner σ ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious10′*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*)) =
((*caller* ∈ *dom* ((*act-info o th-flag*)σ) ⟶
(σ ⊨ (*outs* ← (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*get-caller-error caller σ # outs*)))) ∧
(*caller* ∉ *dom* ((*act-info o th-flag*)σ) ⟶
(∀ *a b*. (*a* = *NO-ERRORS* ⟶
$exec\text{-}action_{id}$-*Mon* (*IPC PREP* (*SEND caller partner msg*)) σ =
Some (*NO-ERRORS*, *b*) ⟶
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈ ⊨
(*outs* ← (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));*P* (*NO-ERRORS # outs*)))) ∧
(∀ *error-memory*. *a* = *ERROR-MEM error-memory* ⟶
$exec\text{-}action_{id}$-*Mon* (*IPC PREP* (*SEND caller partner msg*)) σ =
Some (*ERROR-MEM error-memory*, *b*) ⟶
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM error-memory*,
th-flag := *th-flag* σ
⦇*act-info* := ((*act-info o th-flag*)σ)
(*caller* ↦ (*ERROR-MEM error-memory*),
partner ↦ (*ERROR-MEM error-memory*))⦈⦈
⊨ (*outs* ← (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*ERROR-MEM error-memory # outs*)))) ∧
(∀ *error-IPC*. *a* = *ERROR-IPC error-IPC* ⟶
$exec\text{-}action_{id}$-*Mon* (*IPC PREP* (*SEND caller partner msg*)) σ =
Some (*ERROR-IPC error-IPC*, *b*) ⟶
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
$state_{id}$.*th-flag* := *th-flag* σ
⦇*act-info* := ((*act-info o th-flag*)σ)
(*caller* ↦ (*ERROR-IPC error-IPC*),
partner ↦ (*ERROR-IPC error-IPC*))⦈⦈
⊨ (*outs* ← (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*ERROR-IPC error-IPC# outs*)))))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*) σ)
case *None*
then show *?thesis*

```
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases exec-action_id-Mon (IPC PREP (SEND caller partner msg)) σ)
   case None
   then show ?thesis
   using assms hyp0 hyp1
   by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
  next
   case (Some ab)
   assume hyp2: exec-action_id-Mon (IPC PREP (SEND caller partner msg)) σ = Some ab
   then show ?thesis
   using  hyp0 hyp1 hyp2
   proof (cases ab)
    fix ac ba
    assume hyp3:ab = (ac, ba)
    then show ?thesis
    using hyp0 hyp1 hyp2 hyp3
    proof (cases ac)
     case NO-ERRORS
     assume hyp4: ac = NO-ERRORS
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4
     proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) ba)
      case None
      then show ?thesis
      by simp
     next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon) ba = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
       fix ae bb
       assume hyp6: ad = (ae, bb)
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
       proof (cases error-codes ba)
        case NO-ERRORS
        assume hyp7:error-codes ba = NO-ERRORS
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
         by (auto simp add: PREP-SEND_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
               split: split-if-asm)
       next
        case (ERROR-MEM error-memory)
        assume hyp7:error-codes ba = ERROR-MEM error-memory
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
        by (auto simp add: PREP-SEND_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
```

```
          split: split-if-asm )
      next
        case (ERROR-IPC error-IPC)
        assume hyp7:error-codes ba = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
        by (auto simp add: PREP-SEND_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
          split: split-if-asm )
      qed
    qed
  qed
next
  case (ERROR-MEM error-memory)
  assume hyp4:ac = ERROR-MEM error-memory
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
          (set-error-mem-preps caller partner σ ba error-memory msg))
    case None
    then show ?thesis
    by simp
  next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
            (set-error-mem-preps caller partner σ ba error-memory msg) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
      by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
               PREP-SEND_id-def
        split      : errors.split option.split split-if-asm)
    qed
  qed
next
  case (ERROR-IPC error-IPC)
  assume hyp4:ac = ERROR-IPC error-IPC
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
          (set-error-ipc-preps caller partner σ ba error-IPC msg))
    case None
    then show ?thesis
    by simp
  next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
            (set-error-ipc-preps caller partner σ ba error-IPC msg) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
```

by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
PREP-SEND$_{id}$*-def*
split : *errors.split option.split split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.\ (exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$((b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND|\!) \wedge$
$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$error\text{-}memory = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ error\text{-}memory,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$((b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg\ IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND) \vee$
$(b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg\ IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND)) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma$

$(\!|$*act-info* $:= ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))))))$

by (*auto simp add*: *abort-prep-send-obvious10′ exec-action*$_{id}$*-Mon-prep-send-obvious3*
exec-action$_{id}$*-Mon-prep-send-obvious4 exec-action*$_{id}$*-Mon-prep-send-obvious5*)

lemma *abort-prep-recv-obvious10*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)); P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}prepr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
$|\ \ None \Rightarrow (\sigma \models (P\ []))))$
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some ad*)
  assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
  proof (*cases ad*)
    fix *ae bb*
    assume *hyp6*: *ad* = (*ae*, *bb*)
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
    by(*auto simp add*: *valid-SE-def bind-SE-def*)
  qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some ad*)
  assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
  proof (*cases ad*)
    fix *ae bb*
    assume *hyp6*: *ad* = (*ae*, *bb*)
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
    by(*simp add*: *valid-SE-def bind-SE-def*)
  qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some ad*)
  assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
        (*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) = *Some ad*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*

```
      proof (cases ad)
       fix ae bb
       assume hyp6: ad = (ae, bb)
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
       by(simp add: valid-SE-def bind-SE-def)
      qed
     qed
    qed
   qed
  qed
 qed
qed
```

```
lemma abort-prep-recv-obvious12:
 (σ ⊨ (outs ← (mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift ioprog)); P outs)) =
  (if caller ∈ dom ((act-info o th-flag)σ)
   then (σ ⊨ (outs ← (mbind S(abort_lift ioprog)); P (get-caller-error caller σ # outs)))
   else (case ioprog (IPC PREP (RECV caller partner msg)) σ of
        Some(NO-ERRORS, σ') ⇒
       ((error-tab-transfer caller σ σ') ⊨
        (outs ← (mbind S(abort_lift ioprog)); P (NO-ERRORS # outs)))∧
       (((act-info o th-flag) σ) caller = None) ∧
        ((act-info o th-flag) σ) caller =
        ((act-info o th-flag) (error-tab-transfer caller σ σ')) caller ∧
         (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
       | Some(ERROR-MEM error-mem, σ')⇒
     ((set-error-mem-prepr caller partner σ σ' error-mem msg)
        ⊨ (outs ← (mbind S (abort_lift ioprog)); P (ERROR-MEM error-mem # outs)))∧
     (((act-info o th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller =
       Some (ERROR-MEM error-mem))∧
     (((act-info o th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner =
       Some (ERROR-MEM error-mem)) ∧
     (((act-info o th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller =
      ((act-info o th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
       | Some(ERROR-IPC error-IPC, σ')⇒
       ((set-error-ipc-prepr caller partner σ σ' error-IPC msg)⊨
        (outs ← (mbind S(abort_lift ioprog)); P ( ERROR-IPC error-IPC# outs)))∧
       (((act-info o th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
         Some (ERROR-IPC error-IPC))∧
       (((act-info o th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner =
         Some (ERROR-IPC error-IPC)) ∧
       (((act-info o th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
        ((act-info o th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)
      | None ⇒ (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
 case None
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
```

using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
 case *None*
 then show *?thesis*
 using *assms hyp0 hyp1*
 by (*simp add*: *valid-SE-def bind-SE-def*)
next
 case (*Some ab*)
 assume *hyp2*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some ab*
 then show *?thesis*
 using *hyp0 hyp1 hyp2*
 proof (*cases ab*)
  fix *ac ba*
  assume *hyp3*:*ab* = (*ac*, *ba*)
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3*
  proof (*cases ac*)
   case *NO-ERRORS*
   assume *hyp4*: *ac* = *NO-ERRORS*
   then show *?thesis*
   using *hyp0 hyp1 hyp2 hyp3 hyp4*
   proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
    case *None*
    then show *?thesis*
    by *simp*
   next
    case (*Some ad*)
    assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
    proof (*cases ad*)
     fix *ae bb*
     assume *hyp6*: *ad* = (*ae*, *bb*)
     then show *?thesis*
     using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
     by(*auto simp add*: *valid-SE-def bind-SE-def*)
    qed
   qed
  next
   case (*ERROR-MEM error-memory*)
   assume *hyp4*:*ac* = *ERROR-MEM error-memory*
   then show *?thesis*
   using *hyp0 hyp1 hyp2 hyp3 hyp4*
   proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
    case *None*
    then show *?thesis*
    by *simp*
   next
    case (*Some ad*)
    assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
    proof (*cases ad*)
     fix *ae bb*
     assume *hyp6*: *ad* = (*ae*, *bb*)
     then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious10*$''$:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *None* $\longrightarrow$ ($\sigma \models$ (*P* []))) $\wedge$
(($\forall$ *a* σ'.
(*a* = *NO-ERRORS* $\longrightarrow$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, σ') $\longrightarrow$
((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) $\wedge$
($\forall$ *error-memory*.
a = *ERROR-MEM error-memory* $\longrightarrow$
ioprog (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-MEM error-memory*, σ') $\longrightarrow$
((*set-error-mem-prepr caller partner* σ σ' *error-memory msg*)$\models$
(*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) $\wedge$
($\forall$ *error-IPC*.
a = *ERROR-IPC error-IPC* $\longrightarrow$
ioprog (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, σ') $\longrightarrow$
((*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*

then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*

by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious10'*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)); *P* (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(($\forall$ *a b*.
(*a* = *NO-ERRORS* $\longrightarrow$
$exec\text{-}action_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) $\longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*)))) $\wedge$

($\forall$ *error-memory*.
a = *ERROR-MEM error-memory* $\longrightarrow$
exec-action$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-MEM error-memory*, *b*) $\longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM error-memory*,
state$_{id}$*.th-flag* := *th-flag* σ
(|*act-info* := ((*act-info o th-flag*)σ)
(*caller* $\mapsto$ (*ERROR-MEM error-memory*),
partner $\mapsto$ (*ERROR-MEM error-memory*))|)|)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*ERROR-MEM error-memory* # *outs*)))) $\wedge$
($\forall$ *error-IPC*. *a* = *ERROR-IPC error-IPC* $\longrightarrow$
exec-action$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) $\longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$*.th-flag* := *th-flag* σ
(|*act-info* := ((*act-info o th-flag*)σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC*),
partner $\mapsto$ (*ERROR-IPC error-IPC*))|)|)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*ERROR-IPC error-IPC*# *outs*)))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by(*simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*

```
      by simp
    next
      case (Some ad)
     assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon) (error-tab-transfer caller σ ba) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        proof (cases error-codes ba)
          case NO-ERRORS
          assume hyp7:error-codes ba = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
          by (auto simp add: PREP-RECV_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
                  split: split-if-asm)
        next
          case (ERROR-MEM error-memory)
          assume hyp7:error-codes ba = ERROR-MEM error-memory
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
          by (auto simp add: PREP-RECV_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
                  split: split-if-asm )
        next
          case (ERROR-IPC error-IPC)
          assume hyp7:error-codes ba = ERROR-IPC error-IPC
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
          by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
                  split: split-if-asm )
        qed
      qed
    qed
  next
    case (ERROR-MEM error-memory)
    assume hyp4:ac = ERROR-MEM error-memory
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
              (set-error-mem-prepr caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                (set-error-mem-prepr caller partner σ ba error-memory msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                    PREP-RECV_id-def
```

split : *errors.split option.split split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
PREP-RECV$_{id}$*-def*
split : *errors.split option.split split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-recv-obvious11*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*)) =
((*caller* ∈ *dom* ((*act-info o th-flag*)σ) ⟶
(σ ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*get-caller-error caller* σ # *outs*)))) ∧
(*caller* ∉ *dom* ((*act-info o th-flag*)σ) ⟶
((∀ *a b*.
(*a* = *NO-ERRORS* ⟶
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ ⟶
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈ ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-memory*.
a = *ERROR-MEM error-memory* ⟶
((*b*= σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*⦈∧
¬(*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*) ∧
error-memory = *not-valid-receiver-addr-in-PREP-RECV*)) ⟶
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),

$$
\begin{array}{l}
\quad error\text{-}codes \ := ERROR\text{-}MEM\ error\text{-}memory,\\
\quad state_{id}.th\text{-}flag := th\text{-}flag\ \sigma\\
\qquad (\!|act\text{-}info := ((act\text{-}info\ o\ th\text{-}flag)\sigma)\\
\qquad (caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),\\
\qquad partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)|\!)\\
\qquad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge\\
(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow\\
\quad ((b = \sigma(\!|current\text{-}thread := caller\ ,\\
\qquad thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),\\
\qquad error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV|\!)\wedge\\
\quad exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge\\
\quad \neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge\\
\quad IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge\\
\quad \neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge\\
\quad error\text{-}IPC = error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV) \vee\\
\quad (b = \sigma(\!|current\text{-}thread := caller\ ,\\
\qquad thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),\\
\qquad error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV|\!)\wedge\\
\quad exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge\\
\quad \neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge\\
\quad \neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge\\
\quad error\text{-}IPC = error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV)) \longrightarrow\\
\quad (\sigma(\!|current\text{-}thread := caller,\\
\qquad thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),\\
\qquad error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC,\\
\qquad state_{id}.th\text{-}flag := th\text{-}flag\ \sigma\\
\qquad\quad (\!|act\text{-}info := ((act\text{-}info\ o\ th\text{-}flag)\sigma)\\
\qquad\quad (caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),\\
\qquad\quad partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)|\!)\\
\qquad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))
\end{array}
$$

by $(auto\ simp\ add{:}\ abort\text{-}prep\text{-}recv\text{-}obvious10'\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious3$
$\quad exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious4\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious5)$

4.20.2 Symbolic Execution Rules for WAIT stage

lemma *abort-wait-send-obvious10*:

$$
\begin{array}{l}
(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =\\
\ (if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)\\
\ \ then\ \ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))\\
\ \ else\ (case\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma\ of\\
\qquad Some(NO\text{-}ERRORS, \sigma') \Rightarrow (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\\
\qquad\qquad\qquad \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))\\
\quad |\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow\\
\qquad ((set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)\\
\qquad\ \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))\\
\quad |\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow\\
\qquad ((set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)\\
\qquad\ \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))\\
\\
\quad |\ None \Rightarrow (\sigma \models (P\ []))))
\end{array}
$$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
 case *None*
 then show *?thesis*
 by *simp*
next
 case (*Some a*)
 assume $hyp0{:}\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
 then show *?thesis*

```
using hyp0
proof (cases a)
 fix aa b
 assume hyp1: a = (aa , b)
 then show ?thesis
 using hyp0 hyp1
 proof (cases ioprog (IPC WAIT (SEND caller partner msg)) σ)
  case None
  then show ?thesis
  using assms hyp0 hyp1
  by (simp add: valid-SE-def bind-SE-def)
 next
  case (Some ab)
  assume hyp2: ioprog (IPC WAIT (SEND caller partner msg)) σ = Some ab
  then show ?thesis
  using  hyp0 hyp1 hyp2
  proof (cases ab)
   fix ac ba
   assume hyp3:ab = (ac, ba)
   then show ?thesis
   using hyp0 hyp1 hyp2 hyp3
   proof (cases ac)
    case NO-ERRORS
    assume hyp4: ac = NO-ERRORS
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
     case None
     then show ?thesis
     by simp
    next
     case (Some ad)
     assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
     proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
      by(simp add: valid-SE-def bind-SE-def)
     qed
    qed
   next
    case (ERROR-MEM error-memory)
    assume hyp4:ac = ERROR-MEM error-memory
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-waits caller partner σ ba error-memory msg))
     case None
     then show ?thesis
     by simp
    next
     case (Some ad)
     assume hyp5: mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-waits caller partner σ ba error-memory msg) = Some ad
     then show ?thesis
```

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious12*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*)σ)
then ($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ') $\Rightarrow$
((*error-tab-transfer caller* σ σ')$\models$
(*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))$\wedge$
(((*act-info o th-flag*) σ) *caller* = *None*) $\wedge$
((*act-info o th-flag*) σ) *caller* =
((*act-info o th-flag*) (*error-tab-transfer caller* σ σ')) *caller* $\wedge$
(*th-flag* σ = *th-flag* (*error-tab-transfer caller* σ σ'))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
((*set-error-mem-waits caller partner* σ σ' *error-mem msg*) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *caller* =
Some (*ERROR-MEM error-mem*))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *partner* =
Some (*ERROR-MEM error-mem*)) $\wedge$

$(((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}mem\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}mem\ msg}))\ \mathit{caller} =$
$((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}mem\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}mem\ msg}))\ \mathit{partner})$
$\mid \mathit{Some}(\mathit{ERROR\text{-}IPC\ error\text{-}IPC}, \sigma') \Rightarrow$
$((\mathit{set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}IPC\ msg}) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{ioprog}));\ P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC}\#\ \mathit{outs})))\wedge$
$(((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}IPC\ msg}))\ \mathit{caller} =$
$\mathit{Some}\ (\mathit{ERROR\text{-}IPC\ error\text{-}IPC}))\wedge$
$(((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}IPC\ msg}))\ \mathit{partner} =$
$\mathit{Some}\ (\mathit{ERROR\text{-}IPC\ error\text{-}IPC}))\ \wedge$
$(((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}IPC\ msg}))\ \mathit{caller} =$
$((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma'\ \mathit{error\text{-}IPC\ msg}))\ \mathit{partner})$

$\mid \mathit{None} \Rightarrow (\sigma \models (P\ [])))))$

proof $(\mathit{cases\ mbind}_{FailSave}\ S\ (\mathit{abort}_{lift}\ \mathit{ioprog})\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $\mathit{mbind}_{FailSave}\ S\ (\mathit{abort}_{lift}\ \mathit{ioprog})\ \sigma = \mathit{Some}\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ , b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) $\sigma = \mathit{Some}\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: $ac = \mathit{NO\text{-}ERRORS}$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(\mathit{cases\ mbind}_{FailSave}\ S\ (\mathit{abort}_{lift}\ \mathit{ioprog})\ (\mathit{error\text{-}tab\text{-}transfer\ caller}\ \sigma\ \mathit{ba}))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $\mathit{mbind}_{FailSave}\ S\ (\mathit{abort}_{lift}\ \mathit{ioprog})\ (\mathit{error\text{-}tab\text{-}transfer\ caller}\ \sigma\ \mathit{ba}) = \mathit{Some}\ \mathit{ad}$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*

assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed

qed

lemma *abort-wait-send-obvious10*$''$:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))) \wedge$
$((\forall\, a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
case *None*

then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious10′*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \land$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.$
$(a = NO\text{-}ERRORS \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ b) \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \land$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ b) \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $exec\text{-}action_{id}\text{-}Mon\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma$)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by(*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def$ *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $exec\text{-}action_{id}\text{-}Mon\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*

using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: $ac = $ *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *WAIT-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*

then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: $exec\text{-}action_{id}$-*Mon-def valid-SE-def bind-SE-def*
$WAIT\text{-}SEND_{id}$-*def*
split : *errors.split option.split option.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: $exec\text{-}action_{id}$-*Mon-def valid-SE-def bind-SE-def*
$WAIT\text{-}SEND_{id}$-*def*
split : *errors.split option.split option.split-asm split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious11*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\forall$ *a b*. ($IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\wedge$

$IPC\text{-}params\text{-}c4\ caller\ partner \wedge IPC\text{-}params\text{-}c5\ partner\ \sigma \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS \# outs)))) \wedge$
$(\forall\ error\text{-}IPC.$
$($
$\neg\ IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller\ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\#\ outs)))) \wedge$
$(a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\neg\ IPC\text{-}params\text{-}c4\ \ caller\ \ partner\ \longrightarrow$
$b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND|\!) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND) \wedge$
$(IPC\text{-}params\text{-}c4\ \ caller\ \ partner\ \longrightarrow$
$((\neg\ IPC\text{-}params\text{-}c5\ partner\ \sigma \longrightarrow$
$b = update\text{-}state\text{-}wait\text{-}send\text{-}params5\ \sigma\ caller \wedge$
$error\text{-}codes\ (update\text{-}state\text{-}wait\text{-}send\text{-}params5\ \sigma\ caller) = ERROR\text{-}IPC\ error\text{-}IPC\ ) \wedge$
$\neg\ IPC\text{-}params\text{-}c5\ partner\ \sigma\ ))) \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller\ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$

by $(auto\ simp\ add{:}\ abort\text{-}wait\text{-}send\text{-}obvious10'\ exec\text{-}action_{id}\text{-}Mon\text{-}wait\text{-}send\text{-}obvious3$
$exec\text{-}action_{id}\text{-}Mon\text{-}wait\text{-}send\text{-}obvious4)$

lemma *abort-wait-recv-obvious10*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ \ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma \# outs)))$
$else\ (case\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$(error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS \# outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem \# outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$

```
    | None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC WAIT (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC WAIT (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
            assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
            proof (cases ad)
              fix ae bb
              assume hyp6: ad = (ae, bb)
              then show ?thesis
              using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
              by(simp add: valid-SE-def bind-SE-def)
            qed
          qed
        next
          case (ERROR-MEM error-memory)
          assume hyp4:ac = ERROR-MEM error-memory
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog)
```

$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some ad)*
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof *(cases ad)*
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case *(ERROR-IPC error-IPC)*
assume *hyp4*:$ac = ERROR\text{-}IPC\ error\text{-}IPC$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case *(Some ad)*
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof *(cases ad)*
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious12*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller = None)\ \wedge$
$((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller =$

((*act-info o th-flag*) (*error-tab-transfer caller* σ σ')) *caller* $\wedge$
(*th-flag* σ = *th-flag* (*error-tab-transfer caller* σ σ'))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
((*set-error-mem-waitr caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *caller* =
Some (*ERROR-MEM error-mem*))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *partner* =
Some (*ERROR-MEM error-mem*)) $\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *caller* =
((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *partner*)
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
((*set-error-ipc-waitr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))$\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *caller* =
Some (*ERROR-IPC error-IPC*))$\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *partner* =
Some (*ERROR-IPC error-IPC*)) $\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *caller* =
((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *partner*)
| *None* $\Rightarrow$ (σ $\models$ (*P* [])))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*

by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious10''*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))) \wedge$
$((\forall\ a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*

proof (*cases ac*)
  case *NO-ERRORS*
  assume *hyp4*: *ac* = *NO-ERRORS*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)
    assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
    proof (*cases ad*)
      fix *ae bb*
      assume *hyp6*: *ad* = (*ae*, *bb*)
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
      by(*simp add*: *valid-SE-def bind-SE-def*)
    qed
  qed
next
  case (*ERROR-MEM error-memory*)
  assume *hyp4*:*ac* = *ERROR-MEM error-memory*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)
    assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-waitr caller partner* σ *ba error-memory msg*) = *Some ad*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
    proof (*cases ad*)
      fix *ae bb*
      assume *hyp6*: *ad* = (*ae*, *bb*)
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
      by(*simp add*: *valid-SE-def bind-SE-def*)
    qed
  qed
next
  case (*ERROR-IPC error-IPC*)
  assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)

assume *hyp5*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious10′*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\forall$ *a b*. (*a* = *NO-ERRORS* $\longrightarrow$ *exec-action*$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ =
Some (*NO-ERRORS*, *b*) $\longrightarrow$
((σ⦇*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ ⦈)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*)))) $\wedge$
($\forall$ *error-IPC*. *a* = *ERROR-IPC error-IPC* $\longrightarrow$
exec-action$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) $\longrightarrow$
((σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$*.th-flag* := *state*$_{id}$*.th-flag* σ
⦇*act-info* := *act-info* (*state*$_{id}$*.th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC*),
partner $\mapsto$ (*ERROR-IPC error-IPC*))⦈⦈)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-IPC error-IPC*# *outs*)))))))
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases exec-action*$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ)

```
  case None
  then show ?thesis
  using assms hyp0 hyp1
  by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
 next
  case (Some ab)
  assume hyp2: exec-action_id-Mon (IPC WAIT (RECV caller partner msg)) σ = Some ab
  then show ?thesis
  using  hyp0 hyp1 hyp2
  proof (cases ab)
   fix ac ba
   assume hyp3:ab = (ac, ba)
   then show ?thesis
   using hyp0 hyp1 hyp2 hyp3
   proof (cases ac)
    case NO-ERRORS
    assume hyp4: ac = NO-ERRORS
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) (error-tab-transfer caller σ ba))
     case None
     then show ?thesis
     by simp
    next
     case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon) (error-tab-transfer caller σ ba) = Some ad
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
     proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
      proof (cases error-codes ba)
       case NO-ERRORS
       assume hyp7:error-codes ba = NO-ERRORS
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
       by (auto simp add: WAIT-RECV_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
               split: split-if-asm option.split-asm)
      next
       case (ERROR-MEM error-memory)
       assume hyp7:error-codes ba = ERROR-MEM error-memory
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
       by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
                 split: split-if-asm )
      next
       case (ERROR-IPC error-IPC)
       assume hyp7:error-codes ba = ERROR-IPC error-IPC
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
       by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
               split: split-if-asm )
      qed
     qed
    qed
   next
```

case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
WAIT-RECV$_{id}$*-def*
split : *errors.split option.split option.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
WAIT-RECV$_{id}$*-def*
split : *errors.split option.split option.split-asm split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious11*:

$$
\begin{array}{l}
(\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ ((\mathit{IPC\ WAIT}\ (\mathit{RECV\ caller\ partner\ msg}))\#S) \\
\qquad\qquad\qquad (\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\mathit{\text{-}Mon})); P\ \mathit{outs})) = \\
\ ((\mathit{caller} \in \mathit{dom}\ ((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\sigma) \longrightarrow \\
\quad (\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\mathit{\text{-}Mon})); \\
\qquad\qquad P\ (\mathit{get\text{-}caller\text{-}error\ caller}\ \sigma\ \#\ \mathit{outs})))) \wedge \\
\quad (\mathit{caller} \notin \mathit{dom}\ ((\mathit{act\text{-}info}\ o\ \mathit{th\text{-}flag})\sigma) \longrightarrow \\
\quad (\forall\, a\ b.\ (\mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \wedge \\
\qquad\quad \mathit{IPC\text{-}params\text{-}c4\ caller\ partner} \wedge \mathit{IPC\text{-}params\text{-}c5\ partner}\ \sigma\ \ \longrightarrow \\
\quad\ ((\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}waiting\ caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes}\ \ := \mathit{NO\text{-}ERRORS}, \\
\qquad \mathit{th\text{-}flag}\qquad := \mathit{th\text{-}flag}\ \sigma|\!)) \\
\quad\ \ \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\mathit{\text{-}Mon})); P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs})))) \wedge \\
\quad\ (\forall\, \mathit{error\text{-}IPC}. \\
\quad\ (\\
\quad\ \neg\ \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\quad\ \ \mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}current\ caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\quad\ \ \mathit{error\text{-}codes}\ \ := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}, \\
\quad\ \ \mathit{th\text{-}flag}\qquad := \mathit{th\text{-}flag}\ \sigma \\
\quad\ \ (\!|\mathit{act\text{-}info} := \mathit{act\text{-}info}\ (\mathit{state}_{id}.\mathit{th\text{-}flag}\ \sigma) \\
\quad\ \ (\mathit{caller}\ \ \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}), \\
\quad\ \ \ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}))|\!)|\!) \models \\
\quad\ \ (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\mathit{\text{-}Mon})); \\
\qquad\quad P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}\#\ \mathit{outs})))) \wedge \\
\quad\ (a = \mathit{ERROR\text{-}IPC\ error\text{-}IPC} \longrightarrow \\
\quad\ \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \longrightarrow \\
\quad\ \ ((\neg\ \mathit{IPC\text{-}params\text{-}c4}\ \ \mathit{caller}\ \ \mathit{partner} \longrightarrow \\
\qquad b = \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad\quad \mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}current\ caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad\quad \mathit{error\text{-}codes}\ \ := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}|\!) \wedge \\
\qquad \mathit{error\text{-}IPC} = \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}) \wedge \\
\quad\ \ (\mathit{IPC\text{-}params\text{-}c4}\ \ \mathit{caller\ partner}\ \ \longrightarrow \\
\quad\ \ ((\neg\ \mathit{IPC\text{-}params\text{-}c5\ partner}\ \sigma \longrightarrow \\
\qquad b = \mathit{update\text{-}state\text{-}wait\text{-}recv\text{-}params5}\ \sigma\ \mathit{caller} \wedge \\
\qquad \mathit{error\text{-}codes}\ (\mathit{update\text{-}state\text{-}wait\text{-}recv\text{-}params5}\ \sigma\ \mathit{caller}) = \mathit{ERROR\text{-}IPC\ error\text{-}IPC}) \wedge \\
\quad\ \ \neg\ \mathit{IPC\text{-}params\text{-}c5\ partner}\ \sigma\ ))) \longrightarrow \\
\quad\ ((\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}current\ caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes}\ \ := \mathit{ERROR\text{-}IPC\ error\text{-}IPC}, \\
\qquad \mathit{state}_{id}.\mathit{th\text{-}flag} := \mathit{state}_{id}.\mathit{th\text{-}flag}\ \sigma \\
\qquad (\!|\mathit{act\text{-}info} := \mathit{act\text{-}info}\ (\mathit{state}_{id}.\mathit{th\text{-}flag}\ \sigma) \\
\qquad (\mathit{caller}\ \ \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC}), \\
\qquad\ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC}))|\!)|\!)) \\
\qquad\quad \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\mathit{\text{-}Mon})); \\
\qquad\qquad\quad P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC}\#\ \mathit{outs}))))))))
\end{array}
$$

by (*auto simp add*: *abort-wait-recv-obvious10′* $\mathit{exec\text{-}action}_{id}$*-Mon-wait-recv-obvious3* $\mathit{exec\text{-}action}_{id}$*-Mon-wait-recv-obvious4*)

4.20.3 Symbolic Execution Rules for BUF stage

lemma *abort-buf-send-obvious10*:

$$
\begin{array}{l}
(\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ ((\mathit{IPC\ BUF}\ (\mathit{SEND\ caller\ partner\ msg}))\#S)(\mathit{abort}_{lift}\ \ \mathit{ioprog})); P\ \mathit{outs})) = \\
\ (\mathit{if\ caller} \in \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \\
\ \ \mathit{then}\ (\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{ioprog})); P\ (\mathit{get\text{-}caller\text{-}error\ caller}\ \sigma\ \#\ \mathit{outs}))) \\
\ \ \mathit{else}\ (\mathit{case\ ioprog}\ (\mathit{IPC\ BUF}\ (\mathit{SEND\ caller\ partner\ msg}))\ \sigma\ \mathit{of} \\
\qquad \mathit{Some}(\mathit{NO\text{-}ERRORS}, \sigma') \Rightarrow \\
\quad\ \ (\mathit{error\text{-}tab\text{-}transfer\ caller}\ \sigma\ \sigma') \models
\end{array}
$$

$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))$
$\mid Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))$
$\mid Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
$\mid None \Rightarrow (\sigma \models (P\ [])))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
 case *None*
 then show *?thesis*
 by *simp*
next
 case (*Some a*)
 assume $hyp0$: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
 then show *?thesis*
 using *hyp0*
 proof (*cases a*)
  fix *aa b*
  assume $hyp1$: $a = (aa\ ,\ b)$
  then show *?thesis*
  using *hyp0 hyp1*
  proof $(cases\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma)$
   case *None*
   then show *?thesis*
   using *assms hyp0 hyp1*
   by (*simp add*: *valid-SE-def bind-SE-def*)
  next
   case (*Some ab*)
   assume $hyp2$: $ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
   then show *?thesis*
   using *hyp0 hyp1 hyp2*
   proof (*cases ab*)
    fix *ac ba*
    assume $hyp3$:$ab = (ac,\ ba)$
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3*
    proof (*cases ac*)
     case *NO-ERRORS*
     assume $hyp4$: $ac = NO\text{-}ERRORS$
     then show *?thesis*
     using *hyp0 hyp1 hyp2 hyp3 hyp4*
     proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
      case *None*
      then show *?thesis*
      by *simp*
     next
      case (*Some ad*)
      assume $hyp5$: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba) = Some\ ad$
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
      proof (*cases ad*)
       fix *ae bb*
       assume $hyp6$: $ad = (ae,\ bb)$
       then show *?thesis*
       using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
       by(*simp add*: *valid-SE-def bind-SE-def*)
      qed

qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:ac = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:ac = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious12*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))

then $(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
else $(case\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller = None) \wedge$
$((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))\ caller \wedge$
$(th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner)$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner)$
$|\ None \Rightarrow (\sigma \models (P\ [])))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ , b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)

case *NO-ERRORS*
assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:$ac = ERROR\text{-}MEM\ error\text{-}memory$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:$ac = ERROR\text{-}IPC\ error\text{-}IPC$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$

```
            (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
       fix ae bb
       assume hyp6: ad = (ae, bb)
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
       by(simp add: valid-SE-def bind-SE-def)
      qed
     qed
    qed
   qed
  qed
 qed
qed

lemma abort-buf-send-obvious10'':
 (σ ⊨ (outs ← (mbind ((IPC BUF (SEND caller partner msg))#S)(abort_lift  ioprog)); P outs)) =
  ((caller ∈ dom ((act-info o th-flag)σ) ⟶
   (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ # outs)))) ∧
   (caller ∉ dom ((act-info o th-flag)σ) ⟶
   (ioprog (IPC BUF (SEND caller partner msg)) σ = None ⟶ (σ ⊨ (P []))) ∧
   ((∀ a σ'.
     (a = NO-ERRORS ⟶ ioprog (IPC BUF (SEND caller partner msg)) σ =  Some (NO-ERRORS, σ') ⟶
     ((error-tab-transfer caller σ σ') ⊨
      (outs ← (mbind S(abort_lift  ioprog));P (NO-ERRORS # outs)))) ∧
     (∀ error-memory. a = ERROR-MEM error-memory ⟶
      ioprog (IPC BUF (SEND caller partner msg)) σ = Some (ERROR-MEM error-memory, σ') ⟶
      ((set-error-mem-bufs caller partner σ σ' error-memory msg)
        ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-memory # outs)))) ∧
     (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
      ioprog (IPC BUF (SEND caller partner msg)) σ = Some (ERROR-IPC error-IPC, σ') ⟶
      ((set-error-ipc-bufs caller partner σ σ' error-IPC msg)
        ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC# outs))))))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
 case None
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases ioprog (IPC BUF (SEND caller partner msg)) σ)
   case None
   then show ?thesis
   using assms hyp0 hyp1
   by (simp add: valid-SE-def bind-SE-def)
  next
   case (Some ab)
   assume hyp2: ioprog (IPC BUF (SEND caller partner msg)) σ = Some ab
   then show ?thesis
```

```
using hyp0 hyp1 hyp2
proof (cases ab)
 fix ac ba
 assume hyp3:ab = (ac, ba)
 then show ?thesis
 using hyp0 hyp1 hyp2 hyp3
 proof (cases ac)
  case NO-ERRORS
  assume hyp4: ac = NO-ERRORS
  then show ?thesis
  using hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
   case None
   then show ?thesis
   by simp
  next
   case (Some ad)
   assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
   then show ?thesis
   using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
   proof (cases ad)
    fix ae bb
    assume hyp6: ad = (ae, bb)
    then show ?thesis
    using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
    by(simp add: valid-SE-def bind-SE-def )
   qed
  qed
 next
  case (ERROR-MEM error-memory)
  assume hyp4:ac = ERROR-MEM error-memory
  then show ?thesis
  using hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift ioprog)
         (set-error-mem-bufs caller partner σ ba error-memory msg))
   case None
   then show ?thesis
   by simp
  next
   case (Some ad)
   assume hyp5: mbind_FailSave S (abort_lift ioprog)
          (set-error-mem-bufs caller partner σ ba error-memory msg) = Some ad
   then show ?thesis
   using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
   proof (cases ad)
    fix ae bb
    assume hyp6: ad = (ae, bb)
    then show ?thesis
    using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
    by(simp add: valid-SE-def bind-SE-def)
   qed
  qed
 next
  case (ERROR-IPC error-IPC)
  assume hyp4:ac = ERROR-IPC error-IPC
  then show ?thesis
  using hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift ioprog)
```

$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof $(cases\ ad)$
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by$(simp\ add\colon valid\text{-}SE\text{-}def\ bind\text{-}SE\text{-}def)$
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious10′*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.\ (a = NO\text{-}ERRORS \longrightarrow exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (NO\text{-}ERRORS, b) \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$resource\ := update\text{-}list\ (resource\ \sigma)$
$(zip\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ := NO\text{-}ERRORS,$
$th\text{-}flag\ := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, b) \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller\ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma)$
case *None*
then show *?thesis*

```
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases exec-action_id-Mon (IPC BUF (SEND caller partner msg)) σ)
   case None
   then show ?thesis
   using assms hyp0 hyp1
   by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
  next
   case (Some ab)
   assume hyp2: exec-action_id-Mon (IPC BUF (SEND caller partner msg)) σ = Some ab
   then show ?thesis
   using  hyp0 hyp1 hyp2
   proof (cases ab)
    fix ac ba
    assume hyp3:ab = (ac, ba)
    then show ?thesis
    using hyp0 hyp1 hyp2 hyp3
    proof (cases ac)
     case NO-ERRORS
     assume hyp4: ac = NO-ERRORS
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4
     proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) ba)
      case None
      then show ?thesis
      by simp
     next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon) ba = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
       fix ae bb
       assume hyp6: ad = (ae, bb)
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
       proof (cases error-codes ba)
        case NO-ERRORS
        assume hyp7:error-codes ba = NO-ERRORS
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
        by (auto simp add: BUF-SEND_id-def valid-SE-def bind-SE-def exec-action_id-Mon-def
            split: split-if-asm option.split-asm)
       next
        case (ERROR-MEM error-memory)
        assume hyp7:error-codes ba = ERROR-MEM error-memory
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
        by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
```

```
          split: split-if-asm )
      next
        case (ERROR-IPC error-IPC)
        assume hyp7:error-codes ba = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
        by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
            split: split-if-asm )
      qed
    qed
  qed
next
  case (ERROR-MEM error-memory)
  assume hyp4:ac = ERROR-MEM error-memory
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
         (set-error-mem-bufs caller partner σ ba error-memory msg))
    case None
    then show ?thesis
    by simp
  next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
           (set-error-mem-bufs caller partner σ ba error-memory msg) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
      by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                BUF-SEND_id-def
         split        : errors.split option.split list.split-asm split-if-asm)
    qed
  qed
next
  case (ERROR-IPC error-IPC)
  assume hyp4:ac = ERROR-IPC error-IPC
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4
  proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
         (set-error-ipc-bufs caller partner σ ba error-IPC msg))
    case None
    then show ?thesis
    by simp
  next
    case (Some ad)
    assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
           (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some ad
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
    proof (cases ad)
      fix ae bb
      assume hyp6: ad = (ae, bb)
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
```

by(*auto simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*
BUF-SEND$_{id}$-def
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.\ (a = NO\text{-}ERRORS \longrightarrow IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\sigma(|current\text{-}thread := caller,$
$resource := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(a =\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND \longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\sigma(|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes :=\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller\ \mapsto (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND),$
$partner \mapsto (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND))|)|))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ \ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\#\ outs)))))))$
by (*simp add*: *abort-buf-send-obvious10′ exec-action$_{id}$-Mon-buf-send-obvious3*, *auto*)

lemma *abort-buf-recv-obvious10*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ \ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$(error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$

$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
$\mid None \Rightarrow (\sigma \models (P\ [])))))$
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof $(cases\ ad)$
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by$(simp\ add{:}\ valid\text{-}SE\text{-}def\ bind\text{-}SE\text{-}def)$
qed
qed
next
case $(ERROR\text{-}IPC\ error\text{-}IPC)$
assume *hyp4*:$ac = ERROR\text{-}IPC\ error\text{-}IPC$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ ad)$
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof $(cases\ ad)$
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by$(simp\ add{:}\ valid\text{-}SE\text{-}def\ bind\text{-}SE\text{-}def)$
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious12*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ \ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller = None)\ \wedge$

$((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))\ caller \wedge$
$(th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem)) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner)$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC)) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner)$
$|\ None \Rightarrow (\sigma \models (P\ []))))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
 case *None*
 then show *?thesis*
 by *simp*
next
 case (*Some a*)
 assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
 then show *?thesis*
 using *hyp0*
 proof (*cases a*)
  fix *aa b*
  assume *hyp1*: $a = (aa\ ,\ b)$
  then show *?thesis*
  using *hyp0 hyp1*
  proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
   case *None*
   then show *?thesis*
   using *assms hyp0 hyp1*
   by (*simp add*: *valid-SE-def bind-SE-def*)
  next
   case (*Some ab*)
   assume *hyp2*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) $\sigma = Some\ ab$
   then show *?thesis*
   using *hyp0 hyp1 hyp2*
   proof (*cases ab*)
    fix *ac ba*
    assume *hyp3*:$ab = (ac, ba)$
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3*
    proof (*cases ac*)
     case *NO-ERRORS*
     assume *hyp4*: $ac = NO\text{-}ERRORS$
     then show *?thesis*
     using *hyp0 hyp1 hyp2 hyp3 hyp4*
     proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
      case *None*

then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious10''*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))) \wedge$
$((\forall\, a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some\ (NO\text{-}ERRORS,\ \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*

```
using hyp0 hyp1 hyp2 hyp3
proof (cases ac)
 case NO-ERRORS
 assume hyp4: ac = NO-ERRORS
 then show ?thesis
 using  hyp0 hyp1 hyp2 hyp3 hyp4
 proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
  case None
  then show ?thesis
  by simp
 next
  case (Some ad)
  assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
  proof (cases ad)
   fix ae bb
   assume hyp6: ad = (ae, bb)
   then show ?thesis
   using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
   by(simp add: valid-SE-def bind-SE-def )
  qed
 qed
next
 case (ERROR-MEM error-memory)
 assume hyp4:ac = ERROR-MEM error-memory
 then show ?thesis
 using  hyp0 hyp1 hyp2 hyp3 hyp4
 proof (cases mbind_FailSave S (abort_lift ioprog)
        (set-error-mem-bufr caller partner σ ba error-memory msg))
  case None
  then show ?thesis
  by simp
 next
  case (Some ad)
  assume hyp5: mbind_FailSave S (abort_lift ioprog)
        (set-error-mem-bufr caller partner σ ba error-memory msg) = Some ad
  then show ?thesis
  using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
  proof (cases ad)
   fix ae bb
   assume hyp6: ad = (ae, bb)
   then show ?thesis
   using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
   by(simp add: valid-SE-def bind-SE-def)
  qed
 qed
next
 case (ERROR-IPC error-IPC)
 assume hyp4:ac = ERROR-IPC error-IPC
 then show ?thesis
 using  hyp0 hyp1 hyp2 hyp3 hyp4
 proof (cases mbind_FailSave S (abort_lift ioprog)
        (set-error-ipc-bufr caller partner σ ba error-IPC msg))
  case None
  then show ?thesis
  by simp
 next
```

case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious10′*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
($\forall$ *a b*. (*a* = *NO-ERRORS* $\longrightarrow$ $exec\text{-}action_{id}$*-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some (*NO-ERRORS*, *b*) $\longrightarrow$
((σ⦇*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*)))) $\wedge$
($\forall$ *error-IPC*. *a* = *ERROR-IPC error-IPC* $\longrightarrow$
$exec\text{-}action_{id}$*-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) $\longrightarrow$
((σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
$state_{id}$*.th-flag* := $state_{id}$*.th-flag* σ
⦇*act-info* := *act-info* ($state_{id}$*.th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC*),
partner $\mapsto$ (*ERROR-IPC error-IPC*))⦈⦈)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*));
P (*ERROR-IPC error-IPC*# *outs*)))))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*) σ = *Some a*
then show *?thesis*

using *hyp0*
proof (*cases a*)
  fix *aa b*
  assume *hyp1*: $a = (aa\ ,\ b)$
  then show *?thesis*
  using *hyp0 hyp1*
  proof (*cases exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ)
    case *None*
    then show *?thesis*
    using *assms hyp0 hyp1*
    by(*simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*)
  next
    case (*Some ab*)
    assume *hyp2*: *exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
    then show *?thesis*
    using *hyp0 hyp1 hyp2*
    proof (*cases ab*)
      fix *ac ba*
      assume *hyp3*:$ab = (ac, ba)$
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3*
      proof (*cases ac*)
        case *NO-ERRORS*
        assume *hyp4*: *ac* = *NO-ERRORS*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4*
        proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *ba*)
          case *None*
          then show *?thesis*
          by *simp*
        next
          case (*Some ad*)
          assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *ba* = *Some ad*
          then show *?thesis*
          using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
          proof (*cases ad*)
            fix *ae bb*
            assume *hyp6*: $ad = (ae, bb)$
            then show *?thesis*
            using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
            proof (*cases error-codes ba*)
              case *NO-ERRORS*
              assume *hyp7*:*error-codes ba* = *NO-ERRORS*
              then show *?thesis*
              using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
              by (*auto simp add*: *BUF-RECV$_{id}$-def valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
                  split: *split-if-asm*)
            next
              case (*ERROR-MEM error-memory*)
              assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
              then show *?thesis*
              using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
              by (*auto simp add*: *valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
                  split: *split-if-asm*)
            next
              case (*ERROR-IPC error-IPC*)
              assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
              then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-mem-bufr caller partner σ ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-mem-bufr caller partner σ ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*
BUF-RECV$_{id}$-def
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-ipc-bufr caller partner σ ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-ipc-bufr caller partner σ ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*
BUF-RECV$_{id}$-def
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed

qed
qed
qed
qed
qed

lemma *abort-buf-recv-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.\ (a = NO\text{-}ERRORS \longrightarrow IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV \longrightarrow$
$\neg IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller\ \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV))|\!)|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\#\ outs)))))))$
by $(simp\ add:\ abort\text{-}buf\text{-}recv\text{-}obvious10'\ exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}recv\text{-}obvious3,\ auto)$

4.20.4 Symbolic Execution Rules for MAP stage

lemma *abort-map-send-obvious10*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS,\ \sigma') \Rightarrow$
$(error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma')\Rightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
$|\ None \Rightarrow (\sigma \models (P\ [])))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*

```
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases ioprog (IPC MAP (SEND caller partner msg)) σ)
   case None
   then show ?thesis
   using assms hyp0 hyp1
   by (simp add: valid-SE-def bind-SE-def)
  next
   case (Some ab)
   assume hyp2: ioprog (IPC MAP (SEND caller partner msg)) σ = Some ab
   then show ?thesis
   using  hyp0 hyp1 hyp2
   proof (cases ab)
    fix ac ba
    assume hyp3:ab = (ac, ba)
    then show ?thesis
    using hyp0 hyp1 hyp2 hyp3
    proof (cases ac)
     case NO-ERRORS
     assume hyp4: ac = NO-ERRORS
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4
     proof (cases mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer caller σ ba))
      case None
      then show ?thesis
      by simp
     next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer caller σ ba) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
       fix ae bb
       assume hyp6: ad = (ae, bb)
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
       by(simp add: valid-SE-def bind-SE-def)
      qed
     qed
    next
     case (ERROR-MEM error-memory)
     assume hyp4:ac = ERROR-MEM error-memory
     then show ?thesis
     using  hyp0 hyp1 hyp2 hyp3 hyp4
     proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-maps caller partner σ ba error-memory msg))
      case None
      then show ?thesis
```

by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-send-obvious12*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then ($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ') $\Rightarrow$
((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\wedge$
(((*act-info o th-flag*) σ) *caller* = *None*) $\wedge$
(((*act-info o th-flag*) σ) *caller* =
((*act-info o th-flag*) (*error-tab-transfer caller* σ σ')) *caller*) $\wedge$
(*th-flag* σ = *th-flag* (*error-tab-transfer caller* σ σ'))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$

$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner)$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')\Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC))\ \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$((act\text{-}info\ o\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner)$
$|\ None \Rightarrow (\sigma \models (P\ [])))))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some a*)
  assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
  then show *?thesis*
  using *hyp0*
  proof (*cases a*)
    fix *aa b*
    assume *hyp1*: $a = (aa\ ,\ b)$
    then show *?thesis*
    using *hyp0 hyp1*
    proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ)
      case *None*
      then show *?thesis*
      using *assms hyp0 hyp1*
      by (*simp add*: *valid-SE-def bind-SE-def*)
    next
      case (*Some ab*)
      assume *hyp2*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) $\sigma = Some\ ab$
      then show *?thesis*
      using *hyp0 hyp1 hyp2*
      proof (*cases ab*)
        fix *ac ba*
        assume *hyp3*:$ab = (ac,\ ba)$
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3*
        proof (*cases ac*)
          case *NO-ERRORS*
          assume *hyp4*: $ac = NO\text{-}ERRORS$
          then show *?thesis*
          using *hyp0 hyp1 hyp2 hyp3 hyp4*
          proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba))$
            case *None*
            then show *?thesis*
            by *simp*
          next
            case (*Some ad*)

```
      assume hyp5: mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer caller σ ba) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: valid-SE-def bind-SE-def)
      qed
    qed
  next
    case (ERROR-MEM error-memory)
    assume hyp4:ac = ERROR-MEM error-memory
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-mem-maps caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog)
                (set-error-mem-maps caller partner σ ba error-memory msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(simp add: valid-SE-def bind-SE-def)
      qed
    qed
  next
    case (ERROR-IPC error-IPC)
    assume hyp4:ac = ERROR-IPC error-IPC
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift ioprog)
              (set-error-ipc-maps caller partner σ ba error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift ioprog)
                (set-error-ipc-maps caller partner σ ba error-IPC msg) = Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(simp add: valid-SE-def bind-SE-def)
      qed
```

qed
qed
qed
qed
qed
qed

lemma *abort-map-send-obvious10''*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)); P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))) \wedge$
$((\forall a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS, \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}memory, \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: $ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:$ac = ERROR\text{-}MEM\ error\text{-}memory$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg$))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:$ac = ERROR\text{-}IPC\ error\text{-}IPC$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg$))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*

```
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(simp add: valid-SE-def bind-SE-def)
      qed
     qed
    qed
   qed
  qed
 qed
qed

lemma abort-map-send-obvious10':
 (σ ⊨ (outs ← (mbind ((IPC MAP (SEND caller partner msg))#S)
              (abort_lift  exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom ((act-info o th-flag)σ) ⟶
    (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
              P (get-caller-error caller σ # outs)))) ∧
   (caller ∉ dom ((act-info o th-flag)σ) ⟶
    (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC MAP (SEND caller partner msg)) σ =
                 Some (NO-ERRORS, b) ⟶
     ((σ(|current-thread := caller,
        resource      := init-share-list (resource σ)
                         (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                              ((own-vmem-adr o the o thread-list σ) partner))),
        thread-list    := update-th-ready caller
                       (update-th-ready partner
                       (thread-list σ)),
        error-codes   := NO-ERRORS,
        th-flag       := th-flag σ|))
      ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS # outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
 case None
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa , b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases exec-action_id-Mon (IPC MAP (SEND caller partner msg)) σ)
   case None
   then show ?thesis
   using assms hyp0 hyp1
   by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
  next
   case (Some ab)
   assume hyp2: exec-action_id-Mon (IPC MAP (SEND caller partner msg)) σ = Some ab
   then show ?thesis
   using  hyp0 hyp1 hyp2
   proof (cases ab)
```

fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
  case *NO-ERRORS*
  assume *hyp4*: *ac* = *NO-ERRORS*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) *ba*)
    case *None*
    then show *?thesis*
    by *simp*
  next
    case (*Some ad*)
    assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) *ba* = *Some ad*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
    proof (*cases ad*)
      fix *ae bb*
      assume *hyp6*: *ad* = (*ae*, *bb*)
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
      proof (*cases error-codes ba*)
        case *NO-ERRORS*
        assume *hyp7*:*error-codes ba* = *NO-ERRORS*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
        by (*auto simp add*: *MAP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
            split: *split-if-asm option.split-asm*)
      next
        case (*ERROR-MEM error-memory*)
        assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
        by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
            split: *split-if-asm*)
      next
        case (*ERROR-IPC error-IPC*)
        assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
        by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
            split: *split-if-asm*)
      qed
    qed
  qed
next
  case (*ERROR-MEM error-memory*)
  assume *hyp4*:*ac* = *ERROR-MEM error-memory*
  then show *?thesis*
  using *hyp0 hyp1 hyp2 hyp3 hyp4*
  proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
      (*set-error-mem-maps caller partner* σ *ba error-memory msg*))
    case *None*
    then show *?thesis*
    by *simp*
  next

case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
MAP-SEND$_{id}$*-def*
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
MAP-SEND$_{id}$*-def*
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-send-obvious11*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*)) =
((*caller* ∈ *dom* ((*act-info o th-flag*)σ) ⟶
(σ ⊨ (*outs* ← (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*)))) ∧
(*caller* ∉ *dom* ((*act-info o th-flag*)σ) ⟶
(∀ *a b*. (*a* = *NO-ERRORS* ⟶
((σ⦇*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)

((*own-vmem-adr o the o thread-list* σ) *partner*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*)))))))
by (*simp add*: *abort-map-send-obvious10*$'$ *exec-action*$_{id}$*-Mon-map-send-obvious3*)

lemma *abort-map-recv-obvious10*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then (σ $\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ') $\Rightarrow$
(*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
((*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
((*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))
| *None* $\Rightarrow$ (σ $\models$ (*P* []))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume $hyp5\colon mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ ba) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume $hyp6\colon ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume $hyp4{:}ac = ERROR\text{-}MEM\ error\text{-}memory$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg$))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume $hyp5\colon mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}memory\ msg) = Some\ ad$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume $hyp6\colon ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume $hyp4{:}ac = ERROR\text{-}IPC\ error\text{-}IPC$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof ($cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
($set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg$))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume $hyp5\colon mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ ba\ error\text{-}IPC\ msg) = Some\ ad$
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-recv-obvious12*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* (*act-info* (*th-flag* σ))
then ($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ') $\Rightarrow$
(((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))$\wedge$
(((*act-info o th-flag*) σ) *caller* = *None*) $\wedge$
(((*act-info o th-flag*) σ) *caller* =
((*act-info o th-flag*) (*error-tab-transfer caller* σ σ')) *caller*) $\wedge$
(*th-flag* σ = *th-flag* (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
(((*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *caller* =
Some (*ERROR-MEM error-mem*))$\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *partner* =
Some (*ERROR-MEM error-mem*)) $\wedge$
(((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *caller* =
((*act-info o th-flag*) (*set-error-mem-maps caller partner* σ σ' *error-mem msg*)) *partner*))
| *Some*(*ERROR-IPC error-IPC*, σ')$\Rightarrow$
(((*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))$\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *caller* =
Some (*ERROR-IPC error-IPC*))$\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *partner* =
Some (*ERROR-IPC error-IPC*)) $\wedge$
(((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *caller* =
((*act-info o th-flag*) (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)) *partner*))
| *None* $\Rightarrow$ ($\sigma \models$ (*P* []))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)

then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
  case *None*
  then show *?thesis*
  using *assms hyp0 hyp1*
  by (*simp add*: *valid-SE-def bind-SE-def*)
next
  case (*Some ab*)
  assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
  then show *?thesis*
  using *hyp0 hyp1 hyp2*
  proof (*cases ab*)
    fix *ac ba*
    assume *hyp3*:*ab* = (*ac*, *ba*)
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3*
    proof (*cases ac*)
      case *NO-ERRORS*
      assume *hyp4*: *ac* = *NO-ERRORS*
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4*
      proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
        case *None*
        then show *?thesis*
        by *simp*
      next
        case (*Some ad*)
        assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
        proof (*cases ad*)
          fix *ae bb*
          assume *hyp6*: *ad* = (*ae*, *bb*)
          then show *?thesis*
          using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
          by(*auto simp add*: *valid-SE-def bind-SE-def*)
        qed
      qed
    next
      case (*ERROR-MEM error-memory*)
      assume *hyp4*:*ac* = *ERROR-MEM error-memory*
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4*
      proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
            (*set-error-mem-maps caller partner* σ *ba error-memory msg*))
        case *None*
        then show *?thesis*
        by *simp*
      next
        case (*Some ad*)
        assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
            (*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
        proof (*cases ad*)
          fix *ae bb*
          assume *hyp6*: *ad* = (*ae*, *bb*)

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-recv-obvious10''*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
((*caller* $\in$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*act-info o th-flag*)σ) $\longrightarrow$
(*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *None* $\longrightarrow$ (σ $\models$ (*P* []))) $\wedge$
(($\forall$ *a* σ'.
(*a* = *NO-ERRORS* $\longrightarrow$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, σ') $\longrightarrow$
((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) $\wedge$
($\forall$ *error-memory*. *a* = *ERROR-MEM error-memory* $\longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-MEM error-memory*, σ') $\longrightarrow$
((*set-error-mem-maps caller partner* σ σ' *error-memory msg*) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) $\wedge$
($\forall$ *error-IPC*. *a* = *ERROR-IPC error-IPC* $\longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, σ') $\longrightarrow$
((*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next

case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
 fix *aa b*
 assume *hyp1*: *a* = (*aa* , *b*)
 then show *?thesis*
 using *hyp0 hyp1*
 proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
  case *None*
  then show *?thesis*
  using *assms hyp0 hyp1*
  by (*simp add*: *valid-SE-def bind-SE-def*)
 next
  case (*Some ab*)
  assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
  then show *?thesis*
  using *hyp0 hyp1 hyp2*
  proof (*cases ab*)
   fix *ac ba*
   assume *hyp3*:*ab* = (*ac*, *ba*)
   then show *?thesis*
   using *hyp0 hyp1 hyp2 hyp3*
   proof (*cases ac*)
    case *NO-ERRORS*
    assume *hyp4*: *ac* = *NO-ERRORS*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4*
    proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
     case *None*
     then show *?thesis*
     by *simp*
    next
     case (*Some ad*)
     assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
     then show *?thesis*
     using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
     proof (*cases ad*)
      fix *ae bb*
      assume *hyp6*: *ad* = (*ae*, *bb*)
      then show *?thesis*
      using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
      by(*simp add*: *valid-SE-def bind-SE-def*)
     qed
    qed
   next
    case (*ERROR-MEM error-memory*)
    assume *hyp4*:*ac* = *ERROR-MEM error-memory*
    then show *?thesis*
    using *hyp0 hyp1 hyp2 hyp3 hyp4*
    proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
        (*set-error-mem-maps caller partner* σ *ba error-memory msg*))
     case *None*
     then show *?thesis*
     by *simp*
    next
     case (*Some ad*)

```
        assume hyp5: mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-maps caller partner σ ba error-memory msg) = Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
         fix ae bb
         assume hyp6: ad = (ae, bb)
         then show ?thesis
         using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
         by(simp add: valid-SE-def bind-SE-def)
        qed
       qed
      next
       case (ERROR-IPC error-IPC)
       assume hyp4:ac = ERROR-IPC error-IPC
       then show ?thesis
       using  hyp0 hyp1 hyp2 hyp3 hyp4
       proof (cases mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-maps caller partner σ ba error-IPC msg))
        case None
        then show ?thesis
        by simp
       next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-maps caller partner σ ba error-IPC msg) = Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
         fix ae bb
         assume hyp6: ad = (ae, bb)
         then show ?thesis
         using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
         by(simp add: valid-SE-def bind-SE-def)
        qed
       qed
      qed
     qed
    qed
   qed
qed

lemma abort-map-recv-obvious10':
 (σ ⊨ (outs ← (mbind ((IPC MAP (RECV caller partner msg))#S)
                (abort_lift  exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom ((act-info o th-flag)σ) ⟶
   (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
               P (get-caller-error caller σ # outs)))) ∧
  (caller ∉ dom ((act-info o th-flag)σ) ⟶
   (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC MAP (RECV caller partner msg)) σ =
                    Some (NO-ERRORS, b) ⟶
    ((σ(|current-thread := caller,
      resource      := init-share-list (resource σ)
                        (zip msg ((sorted-list-of-set.F o dom  o fst o Rep-memory)
                              ((own-vmem-adr o the o thread-list σ) caller))),
      thread-list    := update-th-ready caller
                     (update-th-ready partner
                     (thread-list σ)),
```

error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|))
$\models$ (*outs* $\leftarrow$ (*mbind* S(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*)))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by(*simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) *ba*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) *ba* = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *MAP-RECV*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)

next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*
MAP-RECV$_{id}$-def
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae, bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*
MAP-RECV$_{id}$-def
split : *errors.split option.split list.split-asm split-if-asm*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-recv-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\, a\ b.\ (a = NO\text{-}ERRORS \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$resource := init\text{-}share\text{-}list\ (resource\ \sigma)$
$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (NO\text{-}ERRORS\ \#\ outs)))))))$
by (*simp add*:*abort-map-recv-obvious10$'$ exec-action$_{id}$-Mon-map-recv-obvious3*)

4.20.5 Symbolic Execution Rules for DONE stage

lemma *abort-done-send-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)); P\ outs)) =$
$(if\ caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma)$
$then\ \ ((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (if\ ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs)))$
$else\ (\sigma \models (P\ [\,]))))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma) =$
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)

fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ S ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ S ($abort_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*auto simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-send-obvious12*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* ((*act-info o th-flag*)σ)
then (((((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\wedge$
(((*act-info o th-flag*) (*remove-caller-error caller* σ)) *caller* = *None*) $\wedge$
caller $\neq$ *partner* $\wedge$
(((*act-info o th-flag*) σ) *partner* =
((*act-info o th-flag*) (*remove-caller-error caller* σ)) *partner*)) $\vee$
(((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\wedge$
(((*act-info o th-flag*) (*remove-caller-error caller* σ)) *caller* = *None*) $\wedge$
caller = *partner* $\wedge$
(((*act-info o th-flag*) (*remove-caller-error caller* σ)) *partner* = *None*)))
else (*if ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*
then ($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))
else ($\sigma \models$ (*P* []))))
proof (*cases* $mbind_{FailSave}$ S ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ S ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ S ($abort_{lift}$ *ioprog*) σ)
case *None*

then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*auto simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-send-obvious11'*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
((*caller* ∈ *dom* ((*act-info o th-flag*)σ) ⟶
((*remove-caller-error caller* σ) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))) ∧
(*caller* ∉ *dom* ((*act-info o th-flag*)σ) ∧
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ ≠ *None* ⟶
(σ ⊨(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))))∧
(*caller* ∉ *dom* (*act-info* (*th-flag* σ)) ∧
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ = *None* ⟶
(σ ⊨(*P*[]))))
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*

by (*simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-recv-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ \ ((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else$
$(if\ ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
$then\ (\sigma \models (outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs)))$
$else\ (\sigma \models (P\ []))))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma) =$
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*auto simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-recv-obvious12*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))$
$then\ \ ((((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ caller = None) \qquad \wedge$

$caller \neq partner \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ \sigma)\ partner =$
$((act\text{-}info\ o\ th\text{-}flag)\ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ partner)) \quad \vee$

$(((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ caller = None) \quad \wedge$
$caller = partner \wedge$
$(((act\text{-}info\ o\ th\text{-}flag)\ \ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ partner = None)))$
else
$(if\ ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
$then\ (\sigma \models (outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs)))$
$else\ (\sigma \models (P\ [])))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma))$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume $hyp0$: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)(remove\text{-}caller\text{-}error\ caller\ \sigma) =$
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac, ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*auto simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-recv-obvious11'*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$((caller \in dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((act\text{-}info\ o\ th\text{-}flag)\sigma) \wedge$
$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \wedge$
$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ []))))$

proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some a*)
  assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) =
      Some a
  then show *?thesis*
  using *hyp0*
  proof (*cases a*)
    fix *aa b*
    assume *hyp1*: *a* = (*aa*, *b*)
    then show *?thesis*
    using *hyp0 hyp1*
    proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
      case *None*
      then show *?thesis*
      by *simp*
    next
      case (*Some ab*)
      assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some ab*
      then show *?thesis*
      using *hyp0 hyp1 hyp2*
      proof (*cases ab*)
        fix *ac ba*
        assume *hyp3*: *ab* = (*ac*, *ba*)
        then show *?thesis*
        using *hyp0 hyp1 hyp2 hyp3*
        by (*simp add*: *valid-SE-def bind-SE-def split*:*option.split*)
      qed
    qed
  qed
qed

lemmas *trace-normalizer-errors-TestGen* =
  abort-prep-send-obvious10 abort-prep-recv-obvious10 abort-wait-send-obvious10
  abort-wait-recv-obvious10 abort-buf-send-obvious10 abort-buf-recv-obvious10
  abort-done-send-obvious11 abort-done-recv-obvious11 valid-SE-def bind-SE-def
  unit-SE-def

lemmas *trace-normalizer-errors-exec-conj-imp-TestGen* =
  abort-prep-send-obvious10′ abort-prep-recv-obvious10′ abort-wait-send-obvious10′
  abort-wait-recv-obvious10′ abort-buf-send-obvious10′ abort-buf-recv-obvious10′
  abort-done-send-obvious11′ abort-done-recv-obvious11′

end

theory *IPC-symbolic-exec-intros*
imports *IPC-symbolic-exec-rewriting*
begin

4.21 Introduction Rules for Sequence Testing Scheme

4.21.1 Introduction Rules for PREP stage

lemma *abort-prep-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
and *not-in-err-state1*:
$\bigwedge a\ b.\ caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$a = NO\text{-}ERRORS\ \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS, b)\ \Longrightarrow$
$(\sigma(|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$
and *not-in-err-state2*:
$\bigwedge a\ b\ error\text{-}memory.\ caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}MEM\ error\text{-}memory\ \ \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}memory, b)\ \ \Longrightarrow$
$(\sigma(|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ error\text{-}memory,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|)|)$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))$
and *not-in-err-state3*:
$\bigwedge a\ b\ error\text{-}IPC.\ caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}IPC\ error\text{-}IPC\ \ \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, b)\ \ \Longrightarrow$
$(\sigma(|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|)|)$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
shows $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
using *assms*
by (*simp add*: *abort-prep-send-obvious10'*)

lemma *abort-prep-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
and *not-in-err-state1*:

$\bigwedge b.$ *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*)))
and *not-in-err-state2*:
$\bigwedge$ *b error-memory*.
caller $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some (*ERROR-MEM error-memory*, *b*) $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM error-memory*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-MEM error-memory*),
partner $\mapsto$ (*ERROR-MEM error-memory*))|)|)
$\models$ (*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-MEM error-memory* # *outs*)))
and *not-in-err-state3*:
$\bigwedge$ *b error-IPC*. *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(|*act-info* := *act-info* (*state*$_{id}$.*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC*),
partner $\mapsto$ (*ERROR-IPC error-IPC*))|)|)
$\models$ (*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*ERROR-IPC error-IPC*# *outs*)))
shows (σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
using *assms*
by (*simp add*: *abort-prep-recv-obvious10'*)

4.21.2 Introduction rules for WAIT stage

lemma *abort-wait-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
σ $\models$ (*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*))
and *not-in-err-state1*:
$\bigwedge a\, b.$ *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
a = *NO-ERRORS* $\Longrightarrow$
exec-action$_{id}$-*Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) $\Longrightarrow$
σ(|*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))
and *not-in-err-state3*:
$\bigwedge$*a b error-IPC*. *caller* $\notin$ *dom* (*act-info* (*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
a = *ERROR-IPC error-IPC* $\Longrightarrow$
exec-action$_{id}$-*Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) $\Longrightarrow$
σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ

$(\!|$*act-info* := *act-info* $($*state*$_{id}$.*th-flag* $\sigma)$
$($*caller* $\mapsto$ $($*ERROR-IPC error-IPC*$)$,
partner $\mapsto$ $($*ERROR-IPC error-IPC*$))|\!)|\!)$
$\models$ $($*outs* $\leftarrow$ $($*mbind* $(S)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$; *P* $($ *ERROR-IPC error-IPC*# *outs*$))$
shows σ $\models$ $($*outs* $\leftarrow$ $($*mbind* $(($*IPC WAIT* $($*SEND caller partner msg*$))$#*S*$)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$; *P* *outs*$)$
using *assms*
by $($*simp add*: *abort-wait-send-obvious10′*$)$

lemma *abort-wait-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller $\in$ *dom* $($*act-info* $($*th-flag* $\sigma))$ $\Longrightarrow$
σ $\models$ $($*outs* $\leftarrow$ $($*mbind* $(S)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$; *P* $($*get-caller-error caller* σ # *outs*$))$
and *not-in-err-state1*:
$\bigwedge$*a b*. *caller* $\notin$ *dom* $($*act-info* $($*state*$_{id}$.*th-flag* $\sigma))$ $\Longrightarrow$
a = *NO-ERRORS* $\Longrightarrow$
exec-action$_{id}$-*Mon* $($*IPC WAIT* $($*RECV caller partner msg*$))$ σ = *Some* $($*NO-ERRORS*, *b*$)$ $\Longrightarrow$
$\sigma(\!|$*current-thread* := *caller*,
thread-list := *update-th-waiting caller* $($*thread-list* $\sigma)$,
error-codes := *NO-ERRORS*$|\!)$ $\models$
$($*outs* $\leftarrow$ $($*mbind* $(S)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$;*P* $($*NO-ERRORS* # *outs*$))$
and *not-in-err-state2*:
$\bigwedge$*a b error-IPC*. *caller* $\notin$ *dom* $($*act-info* $($*state*$_{id}$.*th-flag* $\sigma))$ $\Longrightarrow$
a = *ERROR-IPC error-IPC* $\Longrightarrow$
exec-action$_{id}$-*Mon* $($*IPC WAIT* $($*RECV caller partner msg*$))$ σ = *Some* $($*ERROR-IPC error-IPC*, *b*$)$ $\Longrightarrow$
$\sigma(\!|$*current-thread* := *caller*,
thread-list := *update-th-current caller* $($*thread-list* $\sigma)$,
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
$(\!|$*act-info* := *act-info* $($*state*$_{id}$.*th-flag* $\sigma)$
$($*caller* $\mapsto$ $($*ERROR-IPC error-IPC*$)$,
partner $\mapsto$ $($*ERROR-IPC error-IPC*$))|\!)|\!)$
$\models$ $($*outs* $\leftarrow$ $($*mbind* $(S)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$; *P* $($ *ERROR-IPC error-IPC*# *outs*$))$
shows σ $\models$ $($*outs* $\leftarrow$ $($*mbind* $(($*IPC WAIT* $($*RECV caller partner msg*$))$#*S*$)($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$; *P* *outs*$)$
using *assms*
by $($*auto simp*: *abort-wait-recv-obvious10′ in-err-state*$)$

4.21.3 Introduction rules rules for BUF stage

lemma *abort-buf-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller $\in$ *dom* $($*act-info* $($*th-flag* $\sigma))$ $\Longrightarrow$
σ $\models$ $($*outs* $\leftarrow$ $($*mbind* *S*$($*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*$))$;
P $($*get-caller-error caller* σ # *outs*$))$
and *not-in-err-state1*:
$\bigwedge$*a b*. *caller* $\notin$ *dom* $($*act-info* $($*th-flag* $\sigma))$ $\Longrightarrow$
a = *NO-ERRORS* $\Longrightarrow$
exec-action$_{id}$-*Mon* $($*IPC BUF* $($*SEND caller partner msg*$))$ σ = *Some* $($*NO-ERRORS*, *b*$)$ $\Longrightarrow$
$\sigma(\!|$*current-thread* := *caller*,
resource := *update-list* $($*resource* $\sigma)$
$($*zip* $(($*sorted-list-of-set.F o dom o fst o Rep-memory*$)$
$(($*own-vmem-adr o the o thread-list* $\sigma)$ *partner*$))$
$($*map* $(($*the o* $($*fst o Rep-memory*$)$ $($*resource* $\sigma)))$ *msg*$))$,
thread-list := *update-th-ready caller*
$($*update-th-ready partner*

$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS \rparr \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS \# outs))$
and *not-in-err-state2*:
$\bigwedge a\ b\ error\text{-}IPC.\ caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}IPC\ error\text{-}IPC \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, b) \Longrightarrow$
$\sigma \lparr current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$\lparr act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))\rparr\rparr \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (ERROR\text{-}IPC\ error\text{-}IPC \# outs))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by (*auto simp* : *abort-buf-send-obvious10'*)

lemma *abort-buf-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma \# outs))$
and *not-in-err-state1*:
$\bigwedge a\ b.\ caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$a = NO\text{-}ERRORS \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS, b) \Longrightarrow$
$\sigma \lparr current\text{-}thread := caller,$
$resource := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\rparr \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS \# outs))$
and *not-in-err-state2*:
$\bigwedge a\ b\ error\text{-}IPC.\ caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}IPC\ error\text{-}IPC \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC, b) \Longrightarrow$
$\sigma \lparr current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$\lparr act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))\rparr\rparr \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (ERROR\text{-}IPC\ error\text{-}IPC \# outs))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by (*auto simp*: *abort-buf-recv-obvious10'*)

4.21.4 Introduction rules for MAP stage

lemma *abort-map-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$\mathit{caller} \in \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\mathit{get\text{-}caller\text{-}error}\ \mathit{caller}\ \sigma\ \#\ \mathit{outs}))$
and *not-in-err-state1*:
$\bigwedge a\ b.\ \mathit{caller} \notin \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$a = \mathit{NO\text{-}ERRORS}\ \ \Longrightarrow$
$\mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}\ (\mathit{IPC}\ \mathit{MAP}\ (\mathit{SEND}\ \mathit{caller}\ \mathit{partner}\ \mathit{msg}))\ \sigma = \mathit{Some}\ (\mathit{NO\text{-}ERRORS}, b)\ \ \Longrightarrow$
$\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{resource}\quad := \mathit{init\text{-}share\text{-}list}\ (\mathit{resource}\ \sigma)$
$(\mathit{zip}\ \mathit{msg}\ ((\mathit{sorted\text{-}list\text{-}of\text{-}set}.F\ o\ \mathit{dom}\ \ o\ \mathit{fst}\ o\ \mathit{Rep\text{-}memory})$
$((\mathit{own\text{-}vmem\text{-}adr}\ o\ \mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \mathit{partner}))),$
$\mathit{thread\text{-}list}\ := \mathit{update\text{-}th\text{-}ready}\ \mathit{caller}$
$(\mathit{update\text{-}th\text{-}ready}\ \mathit{partner}$
$(\mathit{thread\text{-}list}\ \sigma)),$
$\mathit{error\text{-}codes} := \mathit{NO\text{-}ERRORS}|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ (S)(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs}))$
shows $\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ ((\mathit{IPC}\ \mathit{MAP}\ (\mathit{SEND}\ \mathit{caller}\ \mathit{partner}\ \mathit{msg}))\#S)$
$(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));\ P\ \mathit{outs})$
using *assms*
by $(\mathit{auto\ simp} : \mathit{abort\text{-}map\text{-}send\text{-}obvious10}')$

lemma *abort-map-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$\mathit{caller} \in \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\mathit{get\text{-}caller\text{-}error}\ \mathit{caller}\ \sigma\ \#\ \mathit{outs}))$
and *not-in-err-state1*:
$\bigwedge a\ b.\ \mathit{caller} \notin \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$a = \mathit{NO\text{-}ERRORS}\ \ \Longrightarrow$
$\mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}\ (\mathit{IPC}\ \mathit{MAP}\ (\mathit{RECV}\ \mathit{caller}\ \mathit{partner}\ \mathit{msg}))\ \sigma = \mathit{Some}\ (\mathit{NO\text{-}ERRORS}, b)\ \ \Longrightarrow$
$\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{resource}\quad := \mathit{init\text{-}share\text{-}list}\ (\mathit{resource}\ \sigma)$
$(\mathit{zip}\ \mathit{msg}\ ((\mathit{sorted\text{-}list\text{-}of\text{-}set}.F\ o\ \mathit{dom}\ \ o\ \mathit{fst}\ o\ \mathit{Rep\text{-}memory})$
$((\mathit{own\text{-}vmem\text{-}adr}\ o\ \mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \mathit{caller}))),$
$\mathit{thread\text{-}list}\ := \mathit{update\text{-}th\text{-}ready}\ \mathit{caller}$
$(\mathit{update\text{-}th\text{-}ready}\ \mathit{partner}$
$(\mathit{thread\text{-}list}\ \sigma)),$
$\mathit{error\text{-}codes}\ \ := \mathit{NO\text{-}ERRORS}|\!)\models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ (S)(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs}))$
shows $\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ ((\mathit{IPC}\ \mathit{MAP}\ (\mathit{RECV}\ \mathit{caller}\ \mathit{partner}\ \mathit{msg}))\#S)$
$(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));\ P\ \mathit{outs})$
using *assms*
by $(\mathit{auto\ simp}: \mathit{abort\text{-}map\text{-}recv\text{-}obvious10}')$

4.21.5 Introduction rules for DONE stage

lemma *abort-done-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$(\mathit{caller} \in \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$(\mathit{remove\text{-}caller\text{-}error}\ \mathit{caller}\ \sigma) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ (S)(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));\ P\ (\mathit{get\text{-}caller\text{-}error}\ \mathit{caller}\ \sigma\ \#\ \mathit{outs})))$
and *not-in-err-state1*:
$(\mathit{caller} \notin \mathit{dom}\ (\mathit{act\text{-}info}\ (\mathit{state}_{id}.\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$

$\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by $(simp\ add\colon abort\text{-}done\text{-}send\text{-}obvious11\ exec\text{-}action_{id}\text{-}Mon\text{-}def)$

lemma *abort-done-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))$
and *not-in-err-state1*:
$caller \notin dom\ (act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by $(simp\ add\colon abort\text{-}done\text{-}recv\text{-}obvious11\ exec\text{-}action_{id}\text{-}Mon\text{-}def)$

end

theory *IPC-symbolic-exec-elims*
imports *IPC-symbolic-exec-rewriting IPC-symbolic-exec-intros ../../../../src/TestLib*
begin

4.22 Elimination rules for Symbolic Execution of a Test Specification

lemma *threa-table-obvious*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) = \ (act\text{-}info\ (th\text{-}flag\ \sigma)\ caller = None)$
by *auto*

lemma *threa-table-obvious′*:
$(act\text{-}info\ (th\text{-}flag\ \sigma)\ caller = None) = \ (caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
by *auto*

4.22.1 Symbolic Execution rules for PREP SEND

HOL representation

lemma *abort-prep-send-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')$
$\models (outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$

$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-send-HOL-elim21*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$

$(\sigma(\!|$*current-thread* $:=$ *caller*,
thread-list $:=$ *update-th-current caller* (*thread-list* σ),
error-codes $:=$ *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*,
state$_{id}$.*th-flag* $:=$ *th-flag* σ
$(\!|$*act-info* $:=$ (*act-info* (*th-flag* σ))
(*caller* $\mapsto$ (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*),
partner $\mapsto$ (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*))$|\!)|\!)$ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND* # *outs*)))$\Longrightarrow$ *Q*
and
not-in-err-exec31:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*)$\Longrightarrow$
IPC-params-c2 ((*the o thread-list* σ) *partner*)$\Longrightarrow$
$\neg$ *IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) $\Longrightarrow$
$(\sigma(\!|$*current-thread* $:=$ *caller*,
thread-list $:=$ *update-th-current caller* (*thread-list* σ),
error-codes $:=$ *ERROR-IPC error-IPC-22-in-PREP-SEND*,
th-flag $:=$ *th-flag* σ
$(\!|$*act-info* $:=$ *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-22-in-PREP-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-22-in-PREP-SEND*))$|\!)|\!)$ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-22-in-PREP-SEND*# *outs*)))$\Longrightarrow$*Q*
and
not-in-err-exec32:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\Longrightarrow$
$\neg$*IPC-params-c2* ((*the o thread-list* σ) *partner*) $\Longrightarrow$
$(\sigma(\!|$*current-thread* $:=$ *caller*,
thread-list $:=$ *update-th-current caller* (*thread-list* σ),
error-codes $:=$ *ERROR-IPC error-IPC-23-in-PREP-SEND*,
th-flag $:=$ *th-flag* σ
$(\!|$*act-info* $:=$ *act-info* (*state*$_{id}$.*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-23-in-PREP-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-23-in-PREP-SEND*))$|\!)|\!)$ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-23-in-PREP-SEND*# *outs*)))$\Longrightarrow$*Q*
and
not-in-err-exec33:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\Longrightarrow$
IPC-params-c2 ((*the o thread-list* σ) *partner*) $\Longrightarrow$
IPC-params-c6 caller ((*the o thread-list* σ) *partner*)$\Longrightarrow$
$(\sigma(\!|$*current-thread* $:=$ *caller*,
thread-list $:=$ *update-th-ready caller* (*thread-list* σ),
error-codes $:=$ *NO-ERRORS*$|\!)$ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*)))$\Longrightarrow$*Q*
shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-prep-send-mbindFSave-E*)
apply (*simp add*: *in-err-exec*)
apply (*simp add*: *exec-action*$_{id}$-*Mon-prep-send-obvious3*)
apply *auto*
apply (*erule contrapos-np*)

apply *simp*
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action$_{id}$-Mon-prep-send-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec2 exec-action$_{id}$-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action$_{id}$-Mon-prep-send-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

4.22.2 Symbolic Execution rules for PREP RECV

lemma *abort-prep-recv-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)); P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))

case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-recv-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* *S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ $\Longrightarrow$
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec2:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
¬*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
(σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*,
state$_{id}$*.th-flag* := *th-flag* σ
⦇*act-info* := (*act-info* (*th-flag* σ))
(*caller* $\mapsto$ (*ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*),
partner $\mapsto$ (*ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*))⦈⦈ $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-MEM not-valid-receiver-addr-in-PREP-RECV* # *outs*)))$\Longrightarrow$ *Q*
and
not-in-err-exec31:

$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$\neg\ IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\#\ outs))) \Longrightarrow Q$

and

not-in-err-exec32:

$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\#\ outs))) \Longrightarrow Q$

and

not-in-err-exec33:

$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

shows Q
apply (*insert valid-exec*)
apply (*elim abort-prep-recv-mbindFSave-E*)
apply (*simp add*: *in-err-exec*)
apply (*simp add*: $exec\text{-}action_{id}$*-Mon-prep-recv-obvious3*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: $exec\text{-}action_{id}$*-Mon-prep-recv-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)

apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec2 exec-action*$_{id}$*-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-recv-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.22.3 Symbolic Execution rules for WAIT SEND

lemma *abort-wait-send-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*

proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-send-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
IPC-params-c5 partner σ $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec21:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$\neg IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-SEND*# *outs*)))$\Longrightarrow$ *Q*
and
not-in-err-exec22:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
$\neg$*IPC-params-c4 caller partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*,

$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND\#\ outs))) \Longrightarrow Q$

and
not-in-err-exec23:

$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$(thread\text{-}list\ \sigma)\ caller = None \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND\#\ outs))) \Longrightarrow Q$

and
not-in-err-exec24:

$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$\exists\ th.\ (thread\text{-}list\ \sigma)\ caller = Some\ th \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND\#\ outs))) \Longrightarrow Q$

shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-wait-send-mbindFSave-E*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$*-Mon-wait-send-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def WAIT-SEND*$_{id}$*-def split*: *split-if-asm option.split-asm*)
apply (*simp only*: *exec-action*$_{id}$*-Mon-wait-send-obvious4*)
apply (*auto*)
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*elim not-in-err-exec23*)
apply *simp-all*
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)

apply $(simp)$
apply $(fold\ update\text{-}th\text{-}current.simps\ )$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add\colon not\text{-}in\text{-}err\text{-}exec22)$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(simp\ add\colon update\text{-}state\text{-}wait\text{-}send\text{-}params5\text{-}def\ split\colon option.split\text{-}asm\ split\text{-}if\text{-}asm)$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(fold\ update\text{-}th\text{-}current.simps\ )$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add\colon not\text{-}in\text{-}err\text{-}exec21)$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(simp\ add\colon update\text{-}state\text{-}wait\text{-}send\text{-}params5\text{-}def\ split\colon option.split\text{-}asm\ split\text{-}if\text{-}asm)$
apply $(simp\ add\colon exec\text{-}action_{id}\text{-}Mon\text{-}def)$
done

4.22.4 Symbolic Execution rules for WAIT RECV

lemma $abort\text{-}wait\text{-}recv\text{-}mbindFSave\text{-}E$:
assumes $valid\text{-}exec$:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and $in\text{-}err\text{-}state$:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and $not\text{-}in\text{-}err\text{-}state\text{-}Some1$:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and $not\text{-}in\text{-}err\text{-}state\text{-}Some2$:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and $not\text{-}in\text{-}err\text{-}state\text{-}Some3$:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and $not\text{-}in\text{-}err\text{-}state\text{-}None$:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case $True$
then show $?thesis$
using $valid\text{-}exec$
by $(subst\ (asm)\ abort\text{-}wait\text{-}recv\text{-}obvious10,\ elim\ in\text{-}err\text{-}state,\ simp)$
next
case $False$
then show $?thesis$

using *valid-exec*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-recv-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* *S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
IPC-params-c5 partner σ $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec21:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$\neg IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-RECV*# *outs*)))$\Longrightarrow$ *Q*
and
not-in-err-exec22:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}$ *caller partner* σ $\Longrightarrow$
$\neg$*IPC-params-c4 caller partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*,

th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-IPC error-IPC-3-in-WAIT-RECV*# *outs*)))$\Longrightarrow$*Q*

and

not-in-err-exec23:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-recv-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
$\neg$*IPC-params-c5 partner* σ $\Longrightarrow$
(*thread-list* σ) *caller* = *None* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-RECV*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-6-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-6-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-IPC error-IPC-6-in-WAIT-RECV*# *outs*)))$\Longrightarrow$*Q*

and

not-in-err-exec24:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-recv-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
$\neg$*IPC-params-c5 partner* σ $\Longrightarrow$
$\exists$ *th.* (*thread-list* σ) *caller* = *Some th* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-IPC error-IPC-5-in-WAIT-RECV*# *outs*)))$\Longrightarrow$*Q*

shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-wait-recv-mbindFSave-E*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$*-Mon-wait-recv-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def WAIT-RECV*$_{id}$*-def split*: *split-if-asm option.split-asm*)
apply (*simp only*: *exec-action*$_{id}$*-Mon-wait-recv-obvious4*)
apply (*auto*)
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*elim not-in-err-exec23*)
apply *simp-all*
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)

apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.22.5 Symbolic Execution rules for BUF SEND

lemma *abort-buf-send-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows *Q*
proof (*cases caller* ∈ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)

case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-buf-send-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow Q$
and
not-in-err-exec1:

caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-buf-check-st$_{id}$ *caller partner* $\sigma \Longrightarrow$
(σ(|*current-thread* := *caller*,
resource := *update-list* (*resource* σ)
(*zip* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))
(*map* ((*the o* (*fst o Rep-memory*) (*resource* σ))) *msg*)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow Q$
and
not-in-err-exec2:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
$\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-BUF-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-BUF-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*ERROR-IPC error-IPC-1-in-BUF-SEND*# *outs*)))$\Longrightarrow Q$
shows *Q*
using *assms*
apply (*rule abort-buf-send-mbindFSave-E*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: *exec-action*$_{id}$*-Mon-buf-send-obvious3*)+

apply (*simp add*: *not-in-err-exec2*)
apply (*simp-all add*: *exec-action*$_{id}$*-Mon-def*)
done

4.22.6 Symbolic Execution rules for BUF RECV

lemma *abort-buf-recv-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed

qed

lemma *abort-buf-recv-HOL-elim21*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \ \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller\ \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV))|\!)|\!)\ \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\#\ outs)))\Longrightarrow Q$
shows *Q*
using *assms*
apply (*rule abort-buf-recv-mbindFSave-E*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}recv\text{-}obvious3$)+
apply (*simp add*: *not-in-err-exec2*)
apply (*simp-all add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def$)
done

4.22.7 Symbolic Execution rules for MAP SEND

lemma *abort-map-send-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$

and *not-in-err-state-Some1*:
$\bigwedge\sigma'$.
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof $(cases\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-send-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-send-HOL-elim2*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma(|current\text{-}thread := caller,$

resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *partner*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
shows *Q*
using *assms*
apply (*rule abort-map-send-mbindFSave-E*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: *exec-action*$_{id}$*-Mon-map-send-obvious3*)+
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.22.8 Symbolic Execution rules for MAP RECV

lemma *abort-map-recv-mbindFSave-E*:
assumes *valid-exec*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(σ $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and *not-in-err-state-Some1*:
$\bigwedge\sigma'$.
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ') $\Longrightarrow$
((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-Some2*:
$\bigwedge\sigma'$ *error-mem*.
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ') $\Longrightarrow$
((*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
(σ $\models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*

then show *?thesis*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-recv-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-recv-HOL-elim2*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
shows *Q*
using *assms*
apply (*rule abort-map-recv-mbindFSave-E*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: *exec-action*$_{id}$-*Mon-map-recv-obvious3*)+
apply (*simp add*: *exec-action*$_{id}$-*Mon-def*)
done

4.22.9 Symbolic Execution rules for DONE SEND

lemma *abort-done-send-mbindFSave-E*:
assumes *valid-exec*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and *not-in-err-state-Some*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$

ioprog (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None* $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-done-send-obvious11* , *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-send-obvious11*, *simp only*: *False comp-apply*)
next
case *False*
then show *?thesis*
using *assms not-in-err-state-None*
by (*metis* (*mono-tags*) *comp-apply in-err-state False abort-done-send-obvious11*)
qed
qed

lemma *abort-done-send-HOL-elim1*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*))

and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(((*remove-caller-error caller* σ) $\models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*)
and
not-in-err-exec1:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
shows *Q*
using *assms*
by (*rule abort-done-send-mbindFSave-E*, *simp-all add*: *exec-action*$_{id}$*-Mon-def*)

4.22.10 Symbolic Execution rules for DONE SEND

lemma *abort-done-recv-mbindFSave-E*:
assumes *valid-exec*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and *not-in-err-state-Some*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None* $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$*Q*

and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows *Q*
proof (*cases caller* ∈ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-done-recv-obvious11* , *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None*)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-recv-obvious11*, *simp only*: *False*)
next
case *False*
then show *?thesis*
using *assms not-in-err-state-None*
by (*metis* (*mono-tags*) *in-err-state False abort-done-recv-obvious11*)
qed
qed

lemma *abort-done-recv-HOL-elim1*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(((remove\text{-}caller\text{-}error\ caller\ \sigma) \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q)$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
shows *Q*
using *assms*
by (*rule abort-done-recv-mbindFSave-E*, *simp-all add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def$)

4.23 Rules with detailed Constraints

4.23.1 Symbolic Execution rules for PREP SEND

HOL representation

lemma *abort-prep-send-mbindFSave-E′*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$

$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *elim not-in-err-state-None*)

qed
qed

lemma *abort-prep-send-HOL-elim21′*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma) = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\quad := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\quad := NO\text{-}ERRORS,$
$th\text{-}flag\qquad := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\quad := ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND,$
$th\text{-}flag\qquad := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := (act\text{-}info\ (th\text{-}flag\ \sigma))$
$(caller \mapsto (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND),$
$partner \mapsto (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ \#\ outs)))\Longrightarrow Q$
and
not-in-err-exec31:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\Longrightarrow$
$\neg\ IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$

$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND\#\ outs))) \Longrightarrow Q$

and

not-in-err-exec32:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \ \Longrightarrow$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND)\ \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\#\ outs))) \Longrightarrow Q$

and

not-in-err-exec33:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ \ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

shows Q

apply (*insert valid-exec*)
apply (*elim abort-prep-send-mbindFSave-E*$'$)
apply (*simp add*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$*-Mon-prep-send-obvious3*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*rule not-in-err-exec1*)
apply (*simp-all add*: *threa-table-obvious*$'$)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-send-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec2 exec-action*$_{id}$*-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-send-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*$'$)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.23.2 Symbolic Execution rules for PREP RECV

lemma *abort-prep-recv-mbindFSave-E*$'$:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$

Some (*ERROR-MEM error-mem*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-mem-maps caller partner* σ σ' *error-mem msg*))) *caller* =
(*act-info* (*th-flag* (*set-error-mem-maps caller partner* σ σ' *error-mem msg*))) *partner* $\Longrightarrow$
((*set-error-mem-waitr caller partner* σ σ' *error-mem msg*) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* $\Longrightarrow$
((*set-error-ipc-waitr caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC PREP* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
(σ $\models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-recv-HOL-elim21'*:
assumes
valid-exec: (σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$

$$
\begin{array}{l}
\textit{exec-action}_{id}\textit{-Mon-prep-fact1 caller partner } \sigma \Longrightarrow \\
(\textit{act-info } (\textit{ th-flag } \sigma))\ \textit{caller} = \textit{None} \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{error-tab-transfer caller } \sigma\ \sigma)))\ \textit{caller} = \\
(\textit{act-info } (\textit{th-flag } \sigma))\ \textit{caller} \Longrightarrow \\
\ \textit{th-flag } (\textit{error-tab-transfer caller } \sigma\ \sigma)\ = \textit{th-flag } \sigma \Longrightarrow \\
(\sigma(\!|\textit{current-thread} := \textit{caller}, \\
\quad \textit{thread-list} \quad := \textit{update-th-ready caller } (\textit{thread-list } \sigma), \\
\quad \textit{error-codes} \quad := \textit{NO-ERRORS}, \\
\quad \textit{th-flag} \qquad := \textit{th-flag } \sigma |\!) \models \\
\quad (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \textit{ exec-action}_{id}\textit{-Mon}));P\ (\textit{NO-ERRORS} \# \textit{outs}))) \Longrightarrow Q
\end{array}
$$

and

$$
\begin{array}{l}
\textit{not-in-err-exec2}: \\
\textit{caller} \notin \textit{dom } (\textit{act-info } (\textit{th-flag } \sigma)) \Longrightarrow \\
\neg\ \textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner } \sigma\ \textit{msg} \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-mem-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{not-valid-receiver-addr-in-PREP-RECV msg})))\ \textit{caller} = \\
\ \textit{Some } (\textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV}) \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-mem-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{not-valid-receiver-addr-in-PREP-RECV msg})))\ \textit{partner} = \\
\ \textit{Some } (\textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV}) \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-mem-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{not-valid-receiver-addr-in-PREP-RECV msg})))\ \textit{caller} = \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-mem-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{not-valid-receiver-addr-in-PREP-RECV msg})))\ \textit{partner} \Longrightarrow \\
(\sigma(\!|\textit{current-thread} := \textit{caller}, \\
\quad \textit{thread-list} \ := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\quad \textit{error-codes} \ := \textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV}, \\
\quad \textit{th-flag} \qquad := \textit{th-flag } \sigma \\
\quad (\!|\textit{act-info} := (\textit{act-info } (\textit{th-flag } \sigma)) \\
\quad (\textit{caller} \mapsto (\textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV}), \\
\quad\ \textit{partner} \mapsto (\textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV}))|\!)|\!) \models \\
\quad (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \textit{ exec-action}_{id}\textit{-Mon})); \\
\qquad\quad P\ (\textit{ERROR-MEM not-valid-receiver-addr-in-PREP-RECV} \# \textit{outs})))\Longrightarrow Q
\end{array}
$$

and

$$
\begin{array}{l}
\textit{not-in-err-exec31}: \\
\textit{caller} \notin \textit{dom } (\textit{act-info } (\textit{th-flag } \sigma)) \Longrightarrow \\
\textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner } \sigma\ \textit{msg} \Longrightarrow \\
\neg \textit{IPC-params-c1 } ((\textit{the } o \textit{ thread-list } \sigma)\ \ \textit{partner})\Longrightarrow \\
\textit{IPC-params-c2 } ((\textit{the } o \textit{ thread-list } \sigma)\ \ \textit{partner})\Longrightarrow \\
\neg\ \textit{IPC-params-c6 caller } ((\textit{the } o \textit{ thread-list } \sigma)\ \textit{partner}) \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-ipc-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{error-IPC-22-in-PREP-RECV msg})))\ \textit{caller} = \\
\textit{Some } (\textit{ERROR-IPC error-IPC-22-in-PREP-RECV}) \Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-ipc-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{error-IPC-22-in-PREP-RECV msg})))\ \textit{partner} = \\
\textit{Some } (\textit{ERROR-IPC error-IPC-22-in-PREP-RECV})\Longrightarrow \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-ipc-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{error-IPC-22-in-PREP-RECV msg})))\ \textit{caller} = \\
(\textit{act-info } (\textit{th-flag } (\textit{set-error-ipc-maps caller partner } \sigma\ \sigma \\
\qquad\qquad\qquad \textit{error-IPC-22-in-PREP-RECV msg})))\ \textit{partner} \Longrightarrow \\
(\sigma(\!|\textit{current-thread} := \textit{caller}, \\
\quad \textit{thread-list} \ := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\quad \textit{error-codes} \ := \textit{ERROR-IPC error-IPC-22-in-PREP-RECV}, \\
\quad \textit{th-flag} := \textit{th-flag } \sigma \\
\qquad (\!|\textit{act-info} := \textit{act-info } (\textit{th-flag } \sigma) \\
\qquad (\textit{caller} \mapsto (\textit{ERROR-IPC error-IPC-22-in-PREP-RECV}), \\
\qquad\ \textit{partner} \mapsto (\textit{ERROR-IPC error-IPC-22-in-PREP-RECV}))|\!)|\!) \models \\
\quad (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \textit{ exec-action}_{id}\textit{-Mon}));
\end{array}
$$

P ($ERROR\text{-}IPC$ $error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\#$ $outs$)))$\Longrightarrow Q$
and
not-in-err-exec32:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the \circ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$\neg IPC\text{-}params\text{-}c2\ ((the \circ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV)\Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (state_{id}.th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
P ($ERROR\text{-}IPC$ $error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\#$ $outs$)))$\Longrightarrow Q$
and
not-in-err-exec33:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the \circ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the \circ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c6\ caller\ ((the \circ thread\text{-}list\ \sigma)\ partner)\Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ \ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller$
$\Longrightarrow$
$th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))\Longrightarrow Q$
shows Q
apply (*insert valid-exec*)
apply (*elim abort-prep-recv-mbindFSave-E'*)
apply (*simp add*: *in-err-exec*)
apply (*simp only*: $exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious3$)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*rule not-in-err-exec1*)
apply (*simp-all add*: *threa-table-obvious'*)
apply (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious4$)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*

apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec2 exec-action*$_{id}$*-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-recv-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.23.3 Symbolic Execution rules for WAIT SEND

lemma *abort-wait-send-mbindFSave-E*′:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') =\ th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC)\Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$

$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$

and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$

shows Q

proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof $(cases\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma)$
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-send-HOL-elim21'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$

and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ := th\text{-}flag\ \sigma|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

and
not-in-err-exec21:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$

$\neg$*IPC-send-comm-check-st$_{id}$ caller partner* $\sigma \Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-SEND msg))) *caller* =
Some (*ERROR-IPC error-IPC-1-in-WAIT-SEND*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-SEND msg))) *partner* =
Some (*ERROR-IPC error-IPC-1-in-WAIT-SEND*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-SEND msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-SEND msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort$_{lift}$ exec-action$_{id}$-Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-SEND*# *outs*)))$\Longrightarrow$ *Q*

and

not-in-err-exec22:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-send-comm-check-st$_{id}$ caller partner $\sigma \Longrightarrow$
$\neg$*IPC-params-c4 caller partner* $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-SEND msg))) *caller* =
Some (*ERROR-IPC error-IPC-3-in-WAIT-SEND*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-SEND msg))) *partner* =
Some (*ERROR-IPC error-IPC-3-in-WAIT-SEND*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-SEND msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-SEND msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort$_{lift}$ exec-action$_{id}$-Mon*));
P (*ERROR-IPC error-IPC-3-in-WAIT-SEND*# *outs*)))$\Longrightarrow$*Q*

and

not-in-err-exec23:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-send-comm-check-st$_{id}$ caller partner $\sigma \Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
$\neg$*IPC-params-c5 partner* $\sigma \Longrightarrow$
(*thread-list* σ) *caller* = *None* $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-SEND msg))) *caller* =
Some (*ERROR-IPC error-IPC-6-in-WAIT-SEND*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-SEND msg))) *partner* =
Some (*ERROR-IPC error-IPC-6-in-WAIT-SEND*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ

error-IPC-6-in-WAIT-SEND msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-SEND msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-6-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-6-in-WAIT-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-6-in-WAIT-SEND*# *outs*)))$\Longrightarrow$*Q*
and
not-in-err-exec24:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-send-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
$\neg$*IPC-params-c5 partner* σ $\Longrightarrow$
$\exists$ *th*. (*thread-list* σ) *caller* = *Some th* $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *caller* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-SEND*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *partner* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-SEND*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-SEND*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-5-in-WAIT-SEND*# *outs*)))$\Longrightarrow$*Q*
shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-wait-send-mbindFSave-E*′)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-send-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$-*Mon-def* *WAIT-SEND*$_{id}$-*def split*: *split-if-asm option.split-asm*)
apply (*auto*)
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-send-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious*′)
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*simp-all add*: *not-in-err-exec23*)
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)

apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

4.23.4 Symbolic Execution rules for WAIT RECV

lemma *abort-wait-recv-mbindFSave-E′*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ \ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$

$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-recv-HOL-elim21'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \sigma\ \ \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ \ := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ \ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec21:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$

error-IPC-1-in-WAIT-RECV msg))) *caller* =
Some (*ERROR-IPC error-IPC-1-in-WAIT-RECV*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-RECV msg))) *partner* =
Some (*ERROR-IPC error-IPC-1-in-WAIT-RECV*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-RECV msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-1-in-WAIT-RECV msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-RECV*# *outs*)))$\Longrightarrow$ *Q*

and

not-in-err-exec22:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-recv-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
$\neg$*IPC-params-c4 caller partner* $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-RECV msg))) *caller* =
Some (*ERROR-IPC error-IPC-3-in-WAIT-RECV*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-RECV msg))) *partner* =
Some (*ERROR-IPC error-IPC-3-in-WAIT-RECV*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-RECV msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-3-in-WAIT-RECV msg))) *partner* $\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*,
th-flag := *th-flag* σ
(|*act-info* := *act-info* (*th-flag* σ)
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-3-in-WAIT-RECV*))|)|) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-3-in-WAIT-RECV*# *outs*)))$\Longrightarrow$*Q*

and

not-in-err-exec23:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
IPC-recv-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
$\neg$*IPC-params-c5 partner* σ $\Longrightarrow$
(*thread-list* σ) *caller* = *None* $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-RECV msg))) *caller* =
Some (*ERROR-IPC error-IPC-6-in-WAIT-RECV*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-RECV msg))) *partner* =
Some (*ERROR-IPC error-IPC-6-in-WAIT-RECV*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-6-in-WAIT-RECV msg))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ

$error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV\ msg))) \ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV,$
$th\text{-}flag \ := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec24:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$\exists\, th.\ (thread\text{-}list\ \sigma)\ caller = Some\ th \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV\ msg))) \ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV\ msg))) \ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV\ msg))) \ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV\ msg))) \ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV,$
$th\text{-}flag \ := th\text{-}flag\ \sigma$
$(\!|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV))|\!)|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV \#\ outs))) \Longrightarrow Q$
shows Q
apply (*insert valid-exec*)
apply (*elim abort-wait-recv-mbindFSave-E'*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: $exec\text{-}action_{id}\text{-}Mon\text{-}wait\text{-}recv\text{-}obvious3$)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def\ \ WAIT\text{-}RECV_{id}\text{-}def$ *split*: *split-if-asm option.split-asm*)
apply *auto*
apply (*simp only*: $exec\text{-}action_{id}\text{-}Mon\text{-}wait\text{-}recv\text{-}obvious4$)
apply (*auto*)
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*simp-all add*: *not-in-err-exec23*)
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec22*)

apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.23.5 Symbolic Execution rules for BUF SEND

lemma *abort-buf-send-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:

$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows *Q*
proof (*cases caller* ∈ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-buf-send-HOL-elim21′*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ := th\text{-}flag\ \sigma|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow$
$Rep\text{-}memory$
$(resource(\sigma(\!|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$

$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ \ := th\text{-}flag\ \sigma|))) =$
$Rep\text{-}memory\ (update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg))) \Longrightarrow Q$
and
$not\text{-}in\text{-}err\text{-}exec2$:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \ \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND)\Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND,$
$th\text{-}flag\ \ \ \ \ \ := th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller\ \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND))|)|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\#\ outs)))\Longrightarrow Q$
shows Q
using *assms*
apply $(rule\ abort\text{-}buf\text{-}send\text{-}mbindFSave\text{-}E')$
apply *simp*
apply *simp*
apply *simp+*
apply $(simp\ add{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}send\text{-}obvious3)+$
apply $(simp\ add{:}\ \ not\text{-}in\text{-}err\text{-}exec2\ )$
apply $(simp\text{-}all\ add{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}def)$
done

4.23.6 Symbolic Execution rules for BUF RECV

lemma $abort\text{-}buf\text{-}recv\text{-}mbindFSave\text{-}E'$:
assumes $valid\text{-}exec$:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and $in\text{-}err\text{-}state$:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and $not\text{-}in\text{-}err\text{-}state\text{-}Some1$:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$

$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$

and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$

and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$

shows Q
proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed

qed

lemma *abort-buf-recv-HOL-elim21'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow$
$Rep\text{-}memory$
$(resource(\sigma(|current\text{-}thread := caller,$
$resource\ \ \ := update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg)),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|))) =$
$Rep\text{-}memory\ (update\text{-}list\ (resource\ \sigma)$
$(zip\ \ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ caller))$
$(map\ ((the\ o\ (fst\ o\ Rep\text{-}memory)\ (resource\ \sigma)))\ msg))) \Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \ \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$

$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg))) \ partner \Longrightarrow$
$(\sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma$
$(|act\text{-}info := act\text{-}info\ (th\text{-}flag\ \sigma)$
$(caller \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV))|)|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\#\ outs)))\Longrightarrow Q$
shows Q
using *assms*
apply (*rule abort-buf-recv-mbindFSave-E'*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}recv\text{-}obvious3$)+
apply (*simp add*: *not-in-err-exec2*)
apply (*simp-all add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def$)
done

4.23.7 Symbolic Execution rules for MAP SEND

lemma *abort-map-send-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma\ \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$

$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof $(cases\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma)$
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-send-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-send-HOL-elim2'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \sigma\ \ \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource\ \ \ \ := init\text{-}share\text{-}list\ (resource\ \sigma)$
$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))),$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ \ \ := th\text{-}flag\ \sigma|\!) \models$

$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon)); P\ (NO\text{-}ERRORS \# outs))) \Longrightarrow$
Rep-memory
$(resource(\sigma(\!|current\text{-}thread := caller,$
$resource := init\text{-}share\text{-}list\ (resource\ \sigma)$
$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner))),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!))) =$
$Rep\text{-}memory\ (init\text{-}share\text{-}list\ (resource\ \sigma)$
$(zip\ msg\ ((sorted\text{-}list\text{-}of\text{-}set.F\ o\ dom\ \ o\ fst\ o\ Rep\text{-}memory)$
$((own\text{-}vmem\text{-}adr\ o\ the\ o\ thread\text{-}list\ \sigma)\ partner)))) \Longrightarrow Q$
shows Q
using *assms*
apply (*rule abort-map-send-mbindFSave-E′*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious3$)+
apply (*simp-all add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def$)
done

4.23.8 Symbolic Execution rules for MAP RECV

lemma *abort-map-recv-mbindFSave-E′*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)); P\ outs))$
and *in-err-state*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma \# outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(act\text{-}info\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(act\text{-}info\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog)); P\ (NO\text{-}ERRORS \# outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma =\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(act\text{-}info\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(act\text{-}info\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem \# outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$

ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
(*act-info* (*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* $\Longrightarrow$
((*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
(σ $\models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-recv-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-recv-HOL-elim2'*:
assumes
valid-exec: (σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind* *S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$
(*act-info* (*th-flag* σ)) *caller* = *None* $\Longrightarrow$
(*act-info* (*th-flag* (*error-tab-transfer caller* σ σ))) *caller* =
(*act-info* (*th-flag* σ)) *caller* $\Longrightarrow$
th-flag (*error-tab-transfer caller* σ σ) = *th-flag* σ $\Longrightarrow$
(σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*

(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$
Rep-memory
(*resource*(σ(|*current-thread* := *caller*,
resource := *init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*))),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|))) =
Rep-memory (*init-share-list* (*resource* σ)
(*zip msg* ((*sorted-list-of-set.F o dom o fst o Rep-memory*)
((*own-vmem-adr o the o thread-list* σ) *caller*)))) $\Longrightarrow$ *Q*
shows *Q*
using *assms*
apply (*rule abort-map-recv-mbindFSave-E'*)
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*: *exec-action*$_{id}$*-Mon-map-recv-obvious3*)+
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

4.23.9 Symbolic Execution rules for DONE SEND

lemma *abort-done-send-mbindFSave-E'*:
assumes *valid-exec*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state1*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$ *caller* $\neq$ *partner* $\Longrightarrow$
((*act-info* (*th-flag* σ)) *partner* =
((*act-info* (*th-flag* (*remove-caller-error caller* σ)))) *partner*) $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and *in-err-state2*:
caller $\in$ *dom* (*act-info* (*th-flag* σ)) $\Longrightarrow$ *caller* = *partner* $\Longrightarrow$
((*act-info* (*th-flag* (*remove-caller-error caller* σ)))) *partner* = *None* $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and *not-in-err-state-Some*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ $\neq$ *None* $\Longrightarrow$
(σ $\models$ (*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* (*act-info* (*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
(σ $\models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* (*act-info* (*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
apply (*subst* (*asm*) *abort-done-send-obvious12*, *simp*)

```
    apply (erule disjE)
    apply (erule conjE)+
    apply (simp add: in-err-state1)
    apply (erule conjE)+
    apply (simp add: in-err-state2)
    done
next
  case False
  assume hyp1: caller ∉ dom (act-info (th-flag σ))
  then show ?thesis
  proof (cases ioprog (IPC DONE (SEND caller partner msg)) σ ≠ None)
    case True
    then show ?thesis
    using assms
    by (subst (asm) abort-done-send-obvious11, simp only: False comp-apply)
  next
    case False
    then show ?thesis
    using valid-exec False hyp1
      apply (subst (asm) abort-done-send-obvious11)
      apply (simp only: if-False comp-apply split: bool.split-asm )
      apply (elim not-in-err-state-None)
      apply (erule contrapos-np)
      apply (simp-all)
      done
  qed
qed

lemma abort-done-send-HOL-elim1':
  assumes
    valid-exec: (σ ⊨ (outs ← (mbind ((IPC DONE (SEND caller partner msg))#S)
                      (abort_lift exec-action_id-Mon)); P outs))

  and in-err-state1:
    caller ∈ dom (act-info (th-flag σ)) ⟹ caller ≠ partner ⟹
     ((act-info (th-flag (remove-caller-error caller σ))) ) partner =
     (act-info (th-flag σ)) partner ⟹
     ((remove-caller-error caller σ) ⊨
     (outs ← (mbind S (abort_lift exec-action_id-Mon)); P (get-caller-error caller σ # outs)))
       ⟹ Q

  and in-err-state2:
     caller ∈ dom (act-info (th-flag σ)) ⟹  caller = partner ⟹
      ((act-info (th-flag (remove-caller-error caller σ))) ) partner = None ⟹
      ((remove-caller-error caller σ) ⊨
      (outs ← (mbind S (abort_lift exec-action_id-Mon)); P (get-caller-error caller σ # outs))) ⟹ Q

  and not-in-err-exec1:
     caller ∉ dom (act-info (th-flag σ)) ⟹
      (σ ⊨ (outs ← (mbind S(abort_lift exec-action_id-Mon));P (NO-ERRORS # outs))) ⟹ Q

  shows Q
  using valid-exec
  by (rule abort-done-send-mbindFSave-E',
     simp-all add: exec-action_id-Mon-def in-err-state1 in-err-state2 not-in-err-exec1)
```

4.23.10 Symbolic Execution rules for DONE SEND

lemma *abort-done-recv-mbindFSave-E′*:
 assumes *valid-exec*:
  $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$

 and *in-err-state1*:
  $caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow caller \neq partner \Longrightarrow$
  $((act\text{-}info\ (th\text{-}flag\ \sigma))\ partner =$
  $((act\text{-}info\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner) \Longrightarrow$
  $((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
  $(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
  $\Longrightarrow Q$

 and *in-err-state2*:
  $caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow\ caller = partner \Longrightarrow$
  $((act\text{-}info\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner = None \Longrightarrow$
  $((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
  $(outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$

 and *not-in-err-state-Some*:
  $(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
  $ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma \neq None \Longrightarrow$
  $(\sigma \models (outs \leftarrow (mbind\ S\ (abort_{lift}\ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

 and *not-in-err-state-None*:
  $(caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma))) \Longrightarrow$
  $ioprog\ (IPC\ DONE\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
  $(\sigma \models (P\ []))\ \Longrightarrow Q$

 shows *Q*
proof (*cases caller* ∈ *dom* (*act-info* (*th-flag* σ)))
 case *True*
 then show *?thesis*
 using *valid-exec*
  apply (*subst* (*asm*) *abort-done-recv-obvious12*, *simp*)
  apply (*erule disjE*)
  apply (*erule conjE*)+
  apply (*simp add*: *in-err-state1*)
  apply (*erule conjE*)+
  apply (*simp add*: *in-err-state2*)
  done
next
 case *False*
 assume *hyp1*: *caller* ∉ *dom* (*act-info* (*th-flag* σ))
 then show *?thesis*
 proof (*cases ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None*)
  case *True*
  then show *?thesis*
  using *assms*
  by (*subst* (*asm*) *abort-done-recv-obvious11*, *simp only*: *False*)
 next
  case *False*
  then show *?thesis*
  using *valid-exec False hyp1*
   apply (*subst* (*asm*) *abort-done-recv-obvious11*)
   apply (*simp only*: *if-False split*: *bool.split-asm*)
   apply (*elim not-in-err-state-None*)
   apply (*erule contrapos-np*)

apply (*simp-all*)
done
qed
qed

lemma *abort-done-recv-HOL-elim1'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

and *in-err-state1*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow caller \neq partner \Longrightarrow$
$((act\text{-}info\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner =$
$(act\text{-}info\ (th\text{-}flag\ \sigma))\ partner \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$

and *in-err-state2*:
$caller \in dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow\ \ caller = partner \Longrightarrow$
$((act\text{-}info\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner = None \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$

and *not-in-err-exec1*:
$caller \notin dom\ (act\text{-}info\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

shows Q
using *valid-exec*
by (*rule abort-done-recv-mbindFSave-E'*,
simp-all add: $exec\text{-}action_{id}$*-Mon-def in-err-state1 in-err-state2 not-in-err-exec1*)

end

theory *IPC-system-calls*

imports *IPC-symbolic-exec-intros IPC-symbolic-exec-elims*

begin

4.24 HOL representation of PikeOS IPC system calls

We define a system call by a set of operations. PikeOS IPC API contain 7 system calls, each system call can do a set of operations. In this section we will just present the most general one called $p4\_ipc$:

type-synonym $behaviour_{ipc} = trace_{ipc}\ set$
type-synonym $behaviour_{ipc}' = trace_{ipc}\ list$

4.24.1 System calls with thread ID as argument

type-synonym $behaviour_{id} = trace_{ipc}\ list$

definition $P4\text{-}IPC\text{-}BUF_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow nat\ list \Rightarrow behaviour_{id}$
where

P4-IPC-BUF$_{id}$ *caller partner msg* $\equiv$
[*caller* $\rhd_{id}$ *msg* $\rhd_{id}$ *partner*, *caller* $\lhd_{id}$ *msg* $\lhd_{id}$ *partner*,
caller $\unrhd_{id}$ *msg* $\unrhd_{id}$ *partner*, *caller* $\unlhd_{id}$ *msg* $\unlhd_{id}$ *partner*]

definition *P4-IPC-BUF-SEND*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *nat list* $\Rightarrow$ *behaviour*$_{id}$
where
P4-IPC-BUF-SEND$_{id}$ *caller partner msg* $\equiv$ [*caller* $\rhd_{id}$ *msg* $\rhd_{id}$ *partner*, *caller* $\unrhd_{id}$ *msg* $\unrhd_{id}$ *partner*]

definition *P4-IPC-BUF-RECV*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *nat list* $\Rightarrow$ *behaviour*$_{id}$
where
P4-IPC-BUF-RECV$_{id}$ *caller partner msg* $\equiv$ [*caller* $\lhd_{id}$ *msg* $\lhd_{id}$ *partner*, *caller* $\unlhd_{id}$ *msg* $\unlhd_{id}$ *partner*]

definition *P4-IPC-SEND*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *nat list* $\Rightarrow$ *behaviour*$_{id}$
where
P4-IPC-SEND$_{id}$ *caller partner msg* $\equiv$ [*caller* $\rhd_{id}$ *msg* $\rhd_{id}$ *partner*, *caller* $\unrhd_{id}$ *msg* $\unrhd_{id}$ *partner*]

definition *P4-IPC-RECV*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *nat list* $\Rightarrow$ *behaviour*$_{id}$
where
P4-IPC-RECV$_{id}$ *caller partner msg* $\equiv$ [*caller* $\lhd_{id}$ *msg* $\lhd_{id}$ *partner*, *caller* $\unlhd_{id}$ *msg* $\unlhd_{id}$ *partner*]

definition *P4-IPC*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *nat list* $\Rightarrow$ *behaviour*$_{id}$
where
P4-IPC$_{id}$ *caller partner msg* $\equiv$
[*caller* $\rhd_{id}$ *msg* $\rhd_{id}$ *partner*, *caller* $\lhd_{id}$ *msg* $\lhd_{id}$ *partner*,
caller $\unrhd_{id}$ *msg* $\unrhd_{id}$ *partner*, *caller* $\unlhd_{id}$ *msg* $\unlhd_{id}$ *partner*]

4.24.2 System calls based on datatype

datatype ('*thread-id*, '*msg*) *P4-IPC-call* =
P4-IPC-call '*thread-id* '*thread-id* '*msg*
| *P4-IPC-SEND-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-RECV-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-BUF-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-BUF-SEND-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-BUF-RECV-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-MAP-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-MAP-SEND-call* '*thread-id* '*thread-id* '*msg*
| *P4-IPC-MAP-RECV-call* '*thread-id* '*thread-id* '*msg*

value *int*(*card*(*interleave* ([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC MAP (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*)]))

([*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC MAP (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])))

fun *IPC-call-sem*::('*thread-id*, '*msg*) *P4-IPC-call* ⇒
((*p4-stage*$_{ipc}$, ('*thread-id* , '*msg*) *p4-direct*$_{ipc}$)*action*$_{ipc}$ *list*)
where
IPC-call-sem (*P4-IPC-call caller partner msg*) =
([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC MAP (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*),
IPC PREP (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC MAP (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])|
IPC-call-sem (*P4-IPC-SEND-call caller partner msg*) =
([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC MAP (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*)])
|
IPC-call-sem (*P4-IPC-RECV-call caller partner msg*) =
([*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC MAP (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])|
IPC-call-sem (*P4-IPC-BUF-call caller partner msg*) =
([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*),
IPC PREP (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])|
IPC-call-sem (*P4-IPC-BUF-SEND-call caller partner msg*) =
([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*)])
|
IPC-call-sem (*P4-IPC-BUF-RECV-call caller partner msg*) =
([*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])|
IPC-call-sem (*P4-IPC-MAP-call caller partner msg*) =
([*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC MAP (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*),

$IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$IPC\ MAP\ \ (RECV\ caller\ partner\ msg),$
$IPC\ DONE\ (RECV\ caller\ partner\ msg)])|$
$IPC\text{-}call\text{-}sem\ (P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ caller\ partner\ msg) =$
$([IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$IPC\ WAIT\ (SEND\ caller\ partner\ msg),$
$IPC\ MAP\ \ (SEND\ caller\ partner\ msg),$
$IPC\ DONE\ (SEND\ caller\ partner\ msg)])|$
$IPC\text{-}call\text{-}sem\ (P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ caller\ partner\ msg) =$
$([IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$IPC\ MAP\ \ (RECV\ caller\ partner\ msg),$
$IPC\ DONE\ (RECV\ caller\ partner\ msg)])$

4.24.3 Predicates on system calls

definition $is\text{-}ipc\text{-}system\text{-}call_{id}$
where $is\text{-}ipc\text{-}system\text{-}call_{id}\ sc = (\exists\, caller\ partner\ msg.\ sc = P4\text{-}IPC_{id}\ caller\ partner\ msg)$

lemmas $system\text{-}calls\text{-}normalizer =$
$is\text{-}ipc\text{-}system\text{-}call_{id}\text{-}def\ P4\text{-}IPC_{id}\text{-}def$

end

theory *IPC-coverage*

imports *IPC-system-calls*

begin

fun *sync-communication*
$::\ 'a\ list \Rightarrow 'a\ list \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list\ set\ ((\text{-}\ /\lfloor\text{-}\rfloor/\ \text{-})\ [201, 0, 201]\ 200)$
where
$[]\ \lfloor[]\rfloor\ [] \quad = \{[]\}|$
$A\ \lfloor[]\rfloor\ B \quad = interleave\ A\ B|$
$[]\ \lfloor N\rfloor\ [] \quad = \{[]\}|$
$A\ \lfloor[n1, n2]\rfloor\ [] \quad = (if\ n1 \in set\ A \vee n2 \in set\ A\ then\ \{\}\ else\ \{A\})|$
$[]\ \lfloor[n1,n2]\rfloor\ (B) \quad = \ (if\ n1 \in set\ B \vee n2 \in set\ B\ then\ \{\}\ else\ \{B\})|$
$(a\#A)\ \lfloor[n1,n2]\rfloor\ (b\#B)\ = (if\ (a = n1\ \ \wedge b = n2)$
$then\ \ image\ (\lambda\, x.\ n1\ \#n2\#\ x)\ (A\ \lfloor[n1,n2]\rfloor\ B)$
$else$
$if\ a \neq n1\ \ \wedge b = n2$
$then\ \ image\ (\lambda\, x.\ a\ \#\ x)\ (A\ \lfloor[n1,n2]\rfloor\ (b\#B))$
$else$
$if\ a = n1 \wedge b \neq n2$
$then\ image\ (\lambda\, x.\ b\ \#\ x)\ ((a\#A)\ \lfloor[n1,n2]\rfloor\ B)$
$else\ (image(\lambda\, x.\ a\ \#\ x)\ (A\ \lfloor[n1,n2]\rfloor\ (b\#B)) \cup$
$(\ image\ (\lambda\, x.\ b\ \#\ x)\ ((a\#A)\ \lfloor[n1,n2]\rfloor\ \ B))))|$
$A\ \lfloor N\rfloor\ B \quad = \ A\ \lfloor[]\rfloor\ B$

datatype $('th\text{-}id,\ 'sclist)criterion =$
$interleave\text{-}all\ ('th\text{-}id \times 'sclist)\ list$
$|TPAIR\ 'th\text{-}id\ 'th\text{-}id\ 'th\text{-}id \rightharpoonup 'sclist$
$|COMM\ \ 'th\text{-}id\ 'th\text{-}id\ 'th\text{-}id \rightharpoonup 'sclist$

4.24.4 Derivation of communication from system calls

— Definition that let us to derive PikeOS ipc communication from the different system calls

definition

[*simp*]:

$sc\text{-}cases\text{-}IPC\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$ $(*check\ if\ th'\ is\ caller\ of\ sc'\ and$
$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ ((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \unlhd msg \unlhd th) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \unrhd msg \unrhd th') \cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$(th \unlhd msg \unlhd th') \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$(th' \unrhd msg \unrhd th)))$
$|P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$(th \unlhd msg \unlhd th') \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$(th' \unrhd msg \unrhd th)))$
$|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \unlhd msg \unlhd th) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \unrhd msg \unrhd th'))$
$|P4\text{-}IPC\text{-}BUF\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$

$$
\begin{array}{l}
\qquad (th \lhd msg \lhd th') \\
\qquad \lfloor [IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)] \rfloor \\
\qquad ([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)])))) \\
|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow \\
\ (if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg') \\
\ \ then\ \{\} \\
\ \ else\ (th' \rhd msg \rhd th) \\
\qquad \lfloor [IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ] \rfloor \\
\qquad ([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg), \\
\qquad\ \ IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup \\
\qquad (th \lhd msg \lhd th') \\
\qquad \lfloor [IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)] \rfloor \\
\qquad ([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)])) \\
|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow \\
\ (if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg') \\
\ \ then\ \{\} \\
\ \ else\ (th \rhd msg \rhd th') \\
\qquad \lfloor [IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ] \rfloor \\
\qquad ([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg), \\
\qquad\ \ IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup \\
\qquad (th' \lhd msg \lhd th) \\
\qquad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor \\
\qquad ([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)])) \\
|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow \\
\ (if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg') \\
\ \ then\ \{\} \\
\ \ else\ ((th \rhd msg \rhd th') \\
\qquad \lfloor [IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ] \rfloor \\
\qquad ([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg), \\
\qquad\ \ IPC\ MAP\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup \\
\qquad (th' \lhd msg \lhd th) \\
\qquad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor \\
\qquad ([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ MAP\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup \\
\qquad (th' \rhd msg \rhd th) \\
\qquad \lfloor [IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ] \rfloor \\
\qquad ([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg), \\
\qquad\ \ IPC\ MAP\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup \\
\qquad (th \lhd msg \lhd th') \\
\qquad \lfloor [IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)] \rfloor \\
\qquad ([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ MAP\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg), \\
\qquad\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)]))) \\
|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow \\
\ (if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg') \\
\ \ then\ \{\} \\
\ \ else\ (th' \rhd msg \rhd th)
\end{array}
$$

$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)])\ \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])\ \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)])))$

definition
$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}SEND\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$ $(*check\ if\ th'\ is\ caller\ of\ sc'\ and$
$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ ((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \unlhd msg \unlhd th)\ \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \unrhd msg \unrhd th')))$
$|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \unlhd msg \unlhd th)\ \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \unrhd msg \unrhd th'))$
$|P4\text{-}IPC\text{-}BUF\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])\ \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))))$

$|\mathit{P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call}\ th1'\ th2'\ msg' \Rightarrow$
$(\mathit{if}\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\mathit{then}\ \{\}$
$\mathit{else}\ (th \rhd msg \rhd th')$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ BUF}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})]) \cup$
$\quad(th' \lhd msg \lhd th)$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ BUF}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})]))$
$|\mathit{P4\text{-}IPC\text{-}MAP\text{-}call}\ th1'\ th2'\ msg' \Rightarrow$
$(\mathit{if}\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\mathit{then}\ \{\}$
$\mathit{else}\ ((th \rhd msg \rhd th')$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})]) \cup$
$\quad(th' \lhd msg \lhd th)$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})])))$
$|\mathit{P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call}\ th1'\ th2'\ msg' \Rightarrow$
$(\mathit{if}\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\mathit{then}\ \{\}$
$\mathit{else}\ (th \rhd msg \rhd th')$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})]) \cup$
$\quad(th' \lhd msg \lhd th)$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor$
$\quad([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}),$
$\quad\ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})]))$
$|\text{-} \Rightarrow \{\})$

definition
$[\mathit{simp}]$:
$\mathit{sc\text{-}cases\text{-}IPC\text{-}RECV\text{-}call}\ th\ msg\ th'\ sc' =$
$(\mathit{case}\ sc'\ \mathit{of}\ \ \mathit{P4\text{-}IPC\text{-}call}\ th1'\ th2'\ msg' \Rightarrow$
$(\mathit{if}\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\ (*\mathit{check\ if}\ th'\ \mathit{is\ caller\ of}\ sc'\ \mathit{and}$
$\qquad\qquad\qquad\qquad \mathit{th\ is\ his\ partner\ and\ msg}\ msg'\ \mathit{are\ equal}*)$
$\mathit{then}\ \{\}$
$\mathit{else}\ ((th' \rhd msg \rhd th)$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ]\rfloor$
$\quad(th \unlhd msg \unlhd th') \cup$
$\quad(th \lhd msg \lhd th')$
$\quad\lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})]\rfloor$
$\quad(th' \unrhd msg \unrhd th)))$
$|\mathit{P4\text{-}IPC\text{-}SEND\text{-}call}\ th1'\ th2'\ msg' \Rightarrow$
$(\mathit{if}\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\mathit{then}\ \{\}$
$\mathit{else}\ ((th' \rhd msg \rhd th)$

$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$(th \trianglelefteq msg \trianglelefteq th') \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$(th' \trianglerighteq msg \trianglerighteq th)))$
$|P4\text{-}IPC\text{-}BUF\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (($
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \lor (th1' \neq th') \lor\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$

```
|- ⇒ {})
```

definition

```
[simp]:
 sc-cases-IPC-BUF-call th msg th' sc' =
 (case sc' of  P4-IPC-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg') (*check if th' is caller of sc' and
                                                th is his partner and msg msg' are equal*)
   then {}
   else (((th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
         IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
         IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
         IPC BUF  (SEND th th' msg), IPC DONE (SEND th th' msg),
         IPC DONE (RECV th th' msg)]) ∪
        (th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
         IPC BUF  (RECV th th' msg), IPC DONE (SEND th th' msg),
         IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
         IPC BUF  (SEND th' th msg), IPC DONE (SEND th' th msg),
         IPC DONE (RECV th' th msg)]))))
 |P4-IPC-SEND-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else ((th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
         IPC BUF  (RECV th th' msg), IPC DONE (SEND th th' msg),
         IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
         IPC BUF  (SEND th' th msg), IPC DONE (SEND th' th msg),
         IPC DONE (RECV th' th msg)])))
 |P4-IPC-RECV-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else (th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
         IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
         IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
         IPC BUF  (SEND th th' msg), IPC DONE (SEND th th' msg),
         IPC DONE (RECV th th' msg)]))
 |P4-IPC-BUF-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
```

$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))))$
$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|\text{-} \Rightarrow \{\})$

definition
$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\ (*check\ if\ th'\ is\ caller\ of\ sc'\ and$
$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$

$\quad IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$\quad IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\quad IPC\ DONE\ (RECV\ th\ th'\ msg)])))$
$|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ , IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$\quad IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ , IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$\quad IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\quad IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|P4\text{-}IPC\text{-}BUF\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ , IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$\quad IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ , IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$\quad IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\quad IPC\ DONE\ (RECV\ th\ th'\ msg)])))))$
$|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ , IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$\quad IPC\ BUF\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ , IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$\quad IPC\ BUF\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\quad IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|\text{-} \Rightarrow \{\})$

definition
$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$ (*check if th' is caller of sc' and
th is his partner and msg msg' are equal*)
$then\ \{\}$
$else\ (($
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ , IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$

$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ , IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ , IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ , IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))))$
$|P4\text{-}IPC\text{-}BUF\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (($
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ , IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ , IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ , IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ , IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|\text{-} \Rightarrow \{\})$

definition
$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$ (*check if th' is caller of sc' and
th is his partner and msg msg' are equal*)
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$

$$
\begin{array}{l}
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})])\cup \\
\quad\quad (\mathit{th'} \lhd \mathit{msg} \lhd \mathit{th}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})])\cup \\
\quad\quad (\mathit{th'} \rhd \mathit{msg} \rhd \mathit{th}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{RECV\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})])\cup \\
\quad\quad (\mathit{th} \lhd \mathit{msg} \lhd \mathit{th'}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{SEND\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})]))))\\
|\mathit{P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg'} \Rightarrow \\
\ (\mathit{if}\ \ (\mathit{th2'} \neq \mathit{th}) \vee (\mathit{th1'} \neq \mathit{th'}) \vee\ \ (\mathit{msg} \neq \mathit{msg'}) \\
\ \ \mathit{then}\ \{\} \\
\ \ \mathit{else}\ ((\mathit{th'} \rhd \mathit{msg} \rhd \mathit{th}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{RECV\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})])\cup \\
\quad\quad (\mathit{th} \lhd \mathit{msg} \lhd \mathit{th'}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th\ th'\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg})]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{SEND\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})])))\\
|\mathit{P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg'} \Rightarrow \\
\ (\mathit{if}\ \ (\mathit{th2'} \neq \mathit{th}) \vee (\mathit{th1'} \neq \mathit{th'}) \vee\ \ (\mathit{msg} \neq \mathit{msg'}) \\
\ \ \mathit{then}\ \{\} \\
\ \ \mathit{else}\ (\mathit{th} \rhd \mathit{msg} \rhd \mathit{th'}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})])\cup \\
\quad\quad (\mathit{th'} \lhd \mathit{msg} \lhd \mathit{th}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})]))\\
|\mathit{P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg'} \Rightarrow \\
\ (\mathit{if}\ \ (\mathit{th2'} \neq \mathit{th}) \vee (\mathit{th1'} \neq \mathit{th'}) \vee\ \ (\mathit{msg} \neq \mathit{msg'}) \\
\ \ \mathit{then}\ \{\} \\
\ \ \mathit{else}\ (((\mathit{th} \rhd \mathit{msg} \rhd \mathit{th'}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{RECV\ th'\ th\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th'\ th\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th'\ th\ msg})])\cup \\
\quad\quad (\mathit{th'} \lhd \mathit{msg} \lhd \mathit{th}) \\
\quad\quad \lfloor[\mathit{IPC\ WAIT}\ (\mathit{RECV\ th'\ th\ msg})\ ,\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg})]\rfloor \\
\quad\quad ([\mathit{IPC\ PREP}\ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ WAIT}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ MAP}\ \ (\mathit{SEND\ th\ th'\ msg}),\ \mathit{IPC\ DONE}\ (\mathit{SEND\ th\ th'\ msg}), \\
\quad\quad\ \ \mathit{IPC\ DONE}\ (\mathit{RECV\ th\ th'\ msg})])\cup
\end{array}
$$

$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg), IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|\text{-} \Rightarrow \{\})$

definition
$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$ $(*check\ if\ th'\ is\ caller\ of\ sc'\ and$
$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ \ (RECV\ th'\ th\ msg), IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg), IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ \ (SEND\ th\ th'\ msg), IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))))$
$|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$

else (th $\triangleright$ msg $\triangleright$ th')
$\lfloor$[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg)]$\rfloor$
([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
IPC MAP (RECV th' th msg), IPC DONE (SEND th' th msg),
IPC DONE (RECV th' th msg)]) $\cup$
(th' $\triangleleft$ msg $\triangleleft$ th)
$\lfloor$[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]$\rfloor$
([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
IPC MAP (SEND th th' msg), IPC DONE (SEND th th' msg),
IPC DONE (RECV th th' msg)]))
|P4-IPC-MAP-call th1' th2' msg' $\Rightarrow$
(*if* (th2' $\neq$ th) $\vee$ (th1' $\neq$ th') $\vee$ (msg $\neq$ msg')
then {}
else (((th $\triangleright$ msg $\triangleright$ th')
$\lfloor$[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg)]$\rfloor$
([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
IPC MAP (RECV th' th msg), IPC DONE (SEND th' th msg),
IPC DONE (RECV th' th msg)]) $\cup$
(th' $\triangleleft$ msg $\triangleleft$ th)
$\lfloor$[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]$\rfloor$
([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
IPC MAP (SEND th th' msg), IPC DONE (SEND th th' msg),
IPC DONE (RECV th th' msg)]))))
|P4-IPC-MAP-RECV-call th1' th2' msg' $\Rightarrow$
(*if* (th2' $\neq$ th) $\vee$ (th1' $\neq$ th') $\vee$ (msg $\neq$ msg')
then {}
else (th $\triangleright$ msg $\triangleright$ th')
$\lfloor$[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg)]$\rfloor$
([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
IPC MAP (RECV th' th msg), IPC DONE (SEND th' th msg),
IPC DONE (RECV th' th msg)]) $\cup$
(th' $\triangleleft$ msg $\triangleleft$ th)
$\lfloor$[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]$\rfloor$
([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
IPC MAP (SEND th th' msg), IPC DONE (SEND th th' msg),
IPC DONE (RECV th th' msg)]))
|- $\Rightarrow$ {})

definition
[*simp*]:
sc-cases-IPC-MAP-RECV-call th msg th' sc' =
(case sc' of P4-IPC-call th1' th2' msg' $\Rightarrow$
(*if* (th2' $\neq$ th) $\vee$ (th1' $\neq$ th') $\vee$ (msg $\neq$ msg') (*check if th' is caller of sc' and
th is his partner and msg msg' are equal*)
then {}
else ((
(th' $\triangleright$ msg $\triangleright$ th)
$\lfloor$[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg)]$\rfloor$
([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
IPC MAP (RECV th th' msg), IPC DONE (SEND th th' msg),
IPC DONE (RECV th th' msg)]) $\cup$
(th $\triangleleft$ msg $\triangleleft$ th')
$\lfloor$[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]$\rfloor$
([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
IPC MAP (SEND th' th msg), IPC DONE (SEND th' th msg),
IPC DONE (RECV th' th msg)]))))
|P4-IPC-SEND-call th1' th2' msg' $\Rightarrow$
(*if* (th2' $\neq$ th) $\vee$ (th1' $\neq$ th') $\vee$ (msg $\neq$ msg')

then {}
else ((th′ ▷ msg ▷ th)
⌊[IPC WAIT (SEND th′ th msg) , IPC WAIT (RECV th th′ msg)]⌋
([IPC PREP (RECV th th′ msg), IPC WAIT (RECV th th′ msg),
IPC MAP (RECV th th′ msg), IPC DONE (SEND th th′ msg),
IPC DONE (RECV th th′ msg)]) ∪
(th ◁ msg ◁ th′)
⌊[IPC WAIT (RECV th th′ msg) , IPC WAIT (SEND th′ th msg)]⌋
([IPC PREP (SEND th′ th msg), IPC WAIT (SEND th′ th msg),
IPC MAP (SEND th′ th msg), IPC DONE (SEND th′ th msg),
IPC DONE (RECV th′ th msg)])))
|P4-IPC-MAP-call th1′ th2′ msg′ ⇒
(*if* (th2′ ≠ th) ∨ (th1′ ≠ th′) ∨ (msg ≠ msg′)
then {}
else ((
(th′ ▷ msg ▷ th)
⌊[IPC WAIT (SEND th′ th msg) , IPC WAIT (RECV th th′ msg)]⌋
([IPC PREP (RECV th th′ msg), IPC WAIT (RECV th th′ msg),
IPC MAP (RECV th th′ msg), IPC DONE (SEND th th′ msg),
IPC DONE (RECV th th′ msg)]) ∪
(th ◁ msg ◁ th′)
⌊[IPC WAIT (RECV th th′ msg) , IPC WAIT (SEND th′ th msg)]⌋
([IPC PREP (SEND th′ th msg), IPC WAIT (SEND th′ th msg),
IPC MAP (SEND th′ th msg), IPC DONE (SEND th′ th msg),
IPC DONE (RECV th′ th msg)]))))
|P4-IPC-MAP-SEND-call th1′ th2′ msg′ ⇒
(*if* (th2′ ≠ th) ∨ (th1′ ≠ th′) ∨ (msg ≠ msg′)
then {}
else (th′ ▷ msg ▷ th)
⌊[IPC WAIT (SEND th′ th msg) , IPC WAIT (RECV th th′ msg)]⌋
([IPC PREP (RECV th th′ msg), IPC WAIT (RECV th th′ msg),
IPC MAP (RECV th th′ msg), IPC DONE (SEND th th′ msg),
IPC DONE (RECV th th′ msg)]) ∪
(th ◁ msg ◁ th′)
⌊[IPC WAIT (RECV th th′ msg) , IPC WAIT (SEND th′ th msg)]⌋
([IPC PREP (SEND th′ th msg), IPC WAIT (SEND th′ th msg),
IPC MAP (SEND th′ th msg), IPC DONE (SEND th′ th msg),
IPC DONE (RECV th′ th msg)]))
|- ⇒ {})

definition
[simp]:
comm-cases th th′ sc sc′ =
(*case* sc *of* P4-IPC-call th1 th2 msg ⇒
(*if* (th2 ≠ th′) ∨ (th1 ≠ th) ∨ (th = th′) (∗check if th is caller of sc and th′ is his partner∗)
then {}
else sc-cases-IPC-call th msg th′ sc′)
|P4-IPC-SEND-call th1 th2 msg ⇒
(*if* (th2 ≠ th′) ∨ (th1 ≠ th) ∨ (th = th′)
then {}
else sc-cases-IPC-SEND-call th msg th′ sc′)
|P4-IPC-RECV-call th1 th2 msg ⇒
(*if* (th2 ≠ th′) ∨ (th1 ≠ th) ∨ (th = th′)
then {}
else sc-cases-IPC-RECV-call th msg th′ sc′)
|P4-IPC-BUF-call th1 th2 msg ⇒
(*if* (th2 ≠ th′) ∨ (th1 ≠ th) ∨ (th = th′)
then {}
else sc-cases-IPC-BUF-call th msg th′ sc′)

$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1\ th2\ msg \Rightarrow$
$(if\ (th2 \neq th') \vee\ (th1 \neq th) \vee (th = th')$
$then\ \{\}$
$else\ sc\text{-}cases\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th\ msg\ th'\ sc')$
$|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1\ th2\ msg \Rightarrow$
$(if\ (th2 \neq th') \vee\ (th1 \neq th) \vee (th = th')$
$then\ \{\}$
$else\ sc\text{-}cases\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th\ msg\ th'\ sc')$
$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1\ th2\ msg \Rightarrow$
$(if\ (th2 \neq th') \vee\ (th1 \neq th) \vee (th = th')$
$then\ \{\}$
$else\ sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}call\ th\ msg\ th'\ sc')$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1\ th2\ msg \Rightarrow$
$(if\ (th2 \neq th') \vee\ (th1 \neq th) \vee (th = th')$
$then\ \{\}$
$else\ sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th\ msg\ th'\ sc')$
$|P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th1\ th2\ msg \Rightarrow$
$(if\ (th2 \neq th') \vee\ (th1 \neq th) \vee (th = th')$
$then\ \{\}$
$else\ sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th\ msg\ th'\ sc')\ )$

fun $criteria :: ('th\text{-}id, ('th\text{-}id, 'msg)\ P4\text{-}IPC\text{-}call)criterion \Rightarrow$
$((p4\text{-}stage_{ipc}, ('th\text{-}id, 'msg\ )\ p4\text{-}direct_{ipc})action_{ipc}\ list)\ set$
where
$criteria\ (interleave\text{-}all\ S) = undefined$

$|criteria\ (COMM\ th\ th'\ scTab) =$
$(case\ scTab\ th\ of\ None \Rightarrow \{\}$
$|\ Some\ sc \Rightarrow$
$(case\ scTab\ th'\ of\ None \Rightarrow \{\}$
$|\ Some\ sc' \Rightarrow comm\text{-}cases\ th\ th'\ sc\ sc'))$

$|criteria\ (TPAIR\ th\ th'\ scTab) =$
$(case\ scTab\ th\ of\ None \Rightarrow$
$(case\ scTab\ th'\ of\ None \Rightarrow \{\}$
$|\ Some\ sc \Rightarrow$
$\{IPC\text{-}call\text{-}sem\ sc\})$
$|\ Some\ sc \Rightarrow$
$(case\ scTab\ th'\ of\ None \Rightarrow \{IPC\text{-}call\text{-}sem\ sc\}$
$|\ Some\ sc' \Rightarrow interleave\ (IPC\text{-}call\text{-}sem\ sc)\ (IPC\text{-}call\text{-}sem\ sc')))$

4.24.5 Partial order theorem

lemma *partial-order-ipc-instance-resource*:
assumes 1: $th \neq th'$
shows
$image\ (\lambda\ is.\ mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ; MAP\text{-}RECV_{MON}\ a))\ \sigma)$
$(criteria\ (COMM\ th\ th'\ [th \mapsto P4\text{-}IPC\text{-}call\ th\ th'\ msg\ ,$
$th' \mapsto P4\text{-}IPC\text{-}call\ th'\ th\ msg])) =$
$image\ (\lambda\ is.\ mbind\ is\ (\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ; MAP\text{-}RECV_{MON}\ a))\ \sigma)$
$(interleave\ (th \lhd msg \lhd th')\ (th' \unrhd msg \unrhd th))$
oops

lemma $(int\ o\ card)\ (criteria\ (COMM\ th\ th'\ [th \mapsto P4\text{-}IPC\text{-}call\ th\ th'\ msg\ ,$
$th' \mapsto P4\text{-}IPC\text{-}call\ th'\ th\ msg])) <$
$(int\ o\ card)\ ((interleave\ (th \lhd msg \lhd th')\ (th' \unrhd msg \unrhd th)))$

by *simp*

4.24.6 ipc communications derivations

4.24.7 Lemmas on ipc communications

lemma *comm-with-P4-IPC-call-Some*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}call\ th'\ th\ msg)$
and 2: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$
shows $criteria\ (COMM\ th\ th'\ scTab) \neq \{\}$
proof (*cases scTab th*)
fix *scTab th*
case *None*
from *this*
show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
from *this*
show *?thesis*
using *assms*
by *auto*
qed

lemma *comm-with-P4-IPC-BUF-call-Some*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}BUF\text{-}call\ th'\ th\ msg)$
and 2: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$
shows $criteria\ (COMM\ th\ th'\ scTab) \neq \{\}$
proof (*cases scTab th*)
case *None*
assume *1*: $scTab\ th = None$
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: $scTab\ th = Some\ a$
then show *?thesis*
using *assms*
by (*auto simp*: *split:option.split*)
qed

lemma *comm-with-P4-IPC-BUF-SEND-call-Some*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th'\ th\ msg)$
and 2: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$
shows $criteria\ (COMM\ th\ th'\ scTab) \neq \{\}$
proof (*cases scTab th*)
case *None*
assume *1*: $scTab\ th = None$
then show *?thesis*
using *assms*

by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-BUF-RECV-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-BUF-RECV-call th′ th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and *3*: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-MAP-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-MAP-call th′ th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and *3*: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-MAP-SEND-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-MAP-SEND-call th′ th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and *3*: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*

then show *?thesis*
using *assms*
by *auto*
next
case $(Some\ a)$
assume *1*: $scTab\ th = Some\ a$
then show *?thesis*
using *assms*
by $(auto\ simp: split{:}option.split\ )$
qed

lemma *comm-with-P4-IPC-MAP-RECV-call-Some*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th'\ th\ msg)$
and *2*: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$
shows $criteria\ (COMM\ th\ th'\ scTab) \neq \{\}$
proof $(cases\ scTab\ th)$
case *None*
assume *1*: $scTab\ th = None$
then show *?thesis*
using *assms*
by *auto*
next
case $(Some\ a)$
assume *1*: $scTab\ th = Some\ a$
then show *?thesis*
using *assms*
by $(auto\ simp: split{:}option.split\ )$
qed

4.24.8 No communications

lemma *not-comm-SEND-SEND*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}SEND\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}SEND\text{-}call\ th'\ th\ msg)$
and *2*: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$
shows $criteria\ (COMM\ th\ th'\ scTab) = \{\}$
proof $(cases\ scTab\ th)$
case *None*
assume *1*: $scTab\ th = None$
then show *?thesis*
using *assms*
by *auto*
next
case $(Some\ a)$
assume *1*: $scTab\ th = Some\ a$
then show *?thesis*
using *assms*
by $(auto\ simp: split{:}option.split\ )$
qed

lemma *not-comm-SEND-SEND-BUF*:
assumes *1*:$(the\ o\ scTab)\ th = (P4\text{-}IPC\text{-}SEND\text{-}call\ th\ th'\ msg) \wedge$
$(the\ o\ scTab)\ th' = (P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th'\ th\ msg)$
and *2*: $th \in dom\ scTab \wedge th' \in dom\ scTab$
and *3*: $th \neq th'$

shows *criteria* (*COMM th th′ scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-SEND-SEND-MAP*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-SEND-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-MAP-SEND-call th′ th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and *3*: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-RECV-call th′ th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and *3*: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV-BUF*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-BUF-RECV-call th′ th msg*)

and 2: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and 3: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV-MAP*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-MAP-RECV-call th′ th msg*)
and 2: *th* ∈ *dom scTab* ∧ *th′* ∈ *dom scTab*
and 3: *th* ≠ *th′*
shows *criteria* (*COMM th th′ scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

end

Bibliography

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*. Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2002.

[BBW15] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Formal firewall testing: An application of test and proof techniques. *Softw. Test., Verif. Reliab.*, 25(1):34–71, 2015.

[BFNW13] Achim D. Brucker, Abderrahmane Feliachi, Yakoub Nemouchi, and Burkhart Wolff. Test program generation for a microprocessor. *Lecture Notes in Computer Science*, 7942:76–95, 2013.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[BHNW15] Achim D. Brucker, Oto Havle, Yakoub Nemouchi, and Burkhart Wolff. Testing the ipc protocol for a real-time operating system. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Working Conference on Verified Software: Theories, Tools, and Experiments*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2015.

[BN04] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM)*, pages 230–239, 2004.

[BTV09] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

[BW07] Achim D. Brucker and Burkhart Wolff. Test-sequence generation with hol-testgen with an application to firewall testing. In *Tests and Proofs*, pages 149–168, 2007.

[BW13] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. *Formal Asp. Comput. (FAOC)*, 25(5):683–721, 2013.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY USA, 2000. ACM Press.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, June 1940.

[dag96] Handbook of tableau methods. 1996.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. 670:268–284, April 1993.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[FGW10] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. *Unifying Theories in Isabelle/HOL*, volume 6445 of *0302-9743*. Springer Berlin Heidelberg, November 2010.

[FGW12] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/Circus: A process specification and verification environment. In *VSTTE*, volume LNCS 7152 of *Lecture Notes in Computer Science*, pages 243–260, 2012.

[FGW15] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Symbolic test-generation in HOL-TestGen/Cirta: A case study. *Int. J. Software Informatics*, 9(2):177–203, 2015.

[FGWW13] Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel, and Burkhart Wolff. The circus testing theory revisited in Isabelle/HOL. In *Formal Methods and Software Engineering*, pages 131–147, 2013.

[FTW04] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In *FATES*, pages 1–15, 2004.

[FWG12] Abderrahmane Feliachi, Burkhart Wolff, and Marie-Claude Gaudel. Isabelle/circus. *Archive of Formal Proofs*, June 2012. http://afp.sourceforge.net/entries/Circus.shtml, Formal proof development.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In PeterD. Mosses, Mogens Nielsen, and MichaelI. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer Berlin Heidelberg, 1995.

[GB91] M.-C Gaudel G.Bernot and B.Marre. Software testing based on formal specification: A theory and a tool. *Software Engeneering Journal*, 6(6):387–405, November 1991.

[Gil62] Arthur Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, 1962.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. In *Papers Presented at the IEEE Symposium on Logic in Computer Science*, number 22, pages 305–326, Orlando, FL, USA, 1994. Academic Press, Inc.

[haf15] Florian haftmann. Code generation from isabelle/hol theories, May 2015.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Symp. on Operating System Principles (SOSP)*, 1997.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[MQB07] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nip12] Tobias Nipkow. *Programing and Proving in Isabelle/HOL*, May 2012.

[Nip13] Tobias Nipkow. Hol/list.thy, 2013.

[NP00] Tobias Nipkow and Lawrence C Paulson. Hol/nat.thy, 2000.

[NPW14] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. Theory set, 2014.

[NWP13] Tobias Nipkow, Markus Wenzel, and Larry Paulson, Dec 2013.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, pages 409–423, 1993.

[PP10] Leaf Petersen and Enrico Pontelli, editors. *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*. ACM, 2010.

[SLZ07] Weihang Jiang Shan Lu and Yuanyuan Zhou. A study of interleaving coverage criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 533–536, September 2007.

[SYS] SYSGO. Pikeos.

[SYS13a] SYSGO. *PikeOS Fundamentals*. SYSGO, 2013.

[SYS13b] SYSGO. *PikeOS Kernel*. SYSGO, 2013.

[Urb13] Christian Urban. The isabelle cookbook, July 2013.

[Wad92] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, New York, NY, USA, 1992.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB '02*, pages 184–203, London, UK, UK, 2002. Springer-Verlag.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[Wen02] Markus M Wenzel. *Isabelle/Isar—a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.