Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language

ANONYMOUS AUTHOR(S)

Automated program verifiers are typically implemented using an intermediate verification language (IVL), such as Boogie or Why3. A verifier front-end translates the input program and specification into an IVL program, while the back-end generates proof obligations for the IVL program and employs an SMT solver to discharge them. Soundness of such verifiers therefore requires that the front-end translation faithfully captures the semantics of the input program and specification in the IVL program, and that the back-end reports success only if the IVL program is actually correct. For a verification tool to be trustworthy, these soundness conditions must be satisfied by its *actual implementation*, not just the program logic it uses.

In this paper, we present a novel validation methodology that provides formal soundness guarantees for front-end implementations. For each successful run of the verifier, we automatically generate a proof in Isabelle showing that the correctness of the produced IVL program implies the correctness of the input program. This proof can be checked independently from the verifier in Isabelle and can be combined with existing work on validating back-ends to obtain an end-to-end soundness guarantee. Our methodology based on forward simulation employs several modularisation strategies to handle the large semantic gap between the input language and the IVL, as well as the intricacies of practical, optimised translations. We present our methodology for the widely-used Viper and Boogie languages. Our evaluation demonstrates that it is effective in validating the translations performed by the existing Viper implementation.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Semantics.*

Additional Key Words and Phrases: Software Verification, Intermediate Verification Languages, Formal Semantics, Proof Certification

1 INTRODUCTION

Program verifiers are tools that try to automatically establish the correctness of an input program with respect to a specification. A standard approach to achieve automation is by reducing the input program and specification to a set of first-order formulas whose validity implies the correctness of the input program; this is automatically checked using an SMT solver. Instead of directly producing logical formulas, many program verifiers are *translational verifiers:* they translate an input program and specification into a program in an *intermediate verification language (IVL)*; we call this a *front-end translation*. An IVL comes with its own *back-end verifier* that ultimately reduces IVL programs to logical formulas. This translational approach via an IVL allows for the reuse of the IVL's back-end technology across multiple front-end verifiers, and makes for a more understandable target representation than direct mappings to logical formulas, simplifying the development of state-of-the-art program verifiers.

A very wide variety of practical program verifiers are translational verifiers; e.g. Corral [Lal and Qadeer 2014], Dafny [Leino 2010], SMACK [Carter et al. 2016], SYMDIFF [Lahiri et al. 2012], and Viper [Müller et al. 2016] target the imperative Boogie IVL [Leino 2008], while Creusot [Denis et al. 2022] and Frama-C [Kirchner et al. 2015] translate to the functional Why3 IVL [Filliâtre and Paskevich 2013]. Multiple layers of front-end translations and IVLs can also be *composed* (e.g. Prusti [Astrauskas et al. 2019] builds on Viper as an IVL).

To ensure that successful verification indeed implies that the input program satisfies its specification, any translational verifier must meet two *soundness conditions*: (1) *Front-end soundness*: the *translation* into the IVL is faithful, i.e. correctness of the produced IVL program implies correctness

of the input program, and (2) *IVL back-end soundness*: if the back-end IVL verifier reports success, the IVL program is correct. Trustworthiness of program verifiers requires formal guarantees for both soundness conditions. It is *not* sufficient to prove soundness of the program logics they employ in principle: automated verifiers are complex systems, and it is essential that formal guarantees also cover their *actual implementations*, where soundness bugs can and do arise.

Existing work on ensuring front-end soundness is based on idealised implementations that can be formalised on paper or in an interactive theorem prover. In practice, practical front-end translations are implemented in efficient mainstream programming languages, use diverse libraries and programming paradigms, and include subtle optimisations omitted from idealised implementations; there is a very large gap between the translations proved correct and the actual translations used in practice. In this paper, we bridge this gap for the first time, developing an approach to formally validate the front-end soundness of translations used in *existing, practical* verifier implementations. IVL back-end verifier soundness, which includes the soundness of the underlying SMT solver, is a better-studied and orthogonal concern; our results can be combined with work in that area to obtain end-to-end guarantees for an entire verification toolchain [Böhme and Weber 2010; Ekici et al. 2017; Fleury and Schurr 2019; Garchery 2021; Parthasarathy et al. 2021].

Proving front-end soundness once and for all for a realistic verifier implementation is practically infeasible, since such implementations are large (e.g. 17.2 KLOC and 8.5 KLOC for the Dafny-to-Boogie and Viper-to-Boogie front-ends, respectively) and are typically written in languages that lack a full formalisation (C# and Scala, in the example above). Instead, we develop a translation validation approach that *automatically* generates a formal proof on every successful run of the verifier via an instrumentation of the existing implementation. Our proofs are expressed in the Isabelle theorem prover [Nipkow et al. 2002], and thus can be checked independently, effectively removing the (substantial) front-end translation from the trusted code base of the verifier.

Challenges. Formally validating front-end translations is challenging for three main reasons:

1. Semantic gap: There is a large semantic gap between a front-end and an IVL language, which concerns the state model (e.g. neither Boogie nor Why3 have a built-in heap, whereas most front-end languages do), the execution model (e.g. Boogie and Why3 allow unguarded access to program state, while e.g. Viper heap accesses are partial operations and must be guarded by checks), and the program logics used to specify and verify programs (e.g. Boogie and Why3 use first-order predicate transformers, whereas front-end languages use complex logics, such as dynamic frames [Kassios 2006] in Dafny and a flavour of separation logic [Parkinson and Summers 2012; Smans et al. 2012] in Viper). To bridge the semantic gap, front-ends translate input programs into a complex combination of low-level operations and background logical axiomatisations of input language concepts; a validation technique needs to precisely account for the combination of these logical ingredients, while allowing the separation of translation aspects for the sake of modularity and maintainability.

2. Diverse translations: Practical front-end translations are *diverse* in the sense that they use multiple alternative translations for the same feature, e.g. more efficient translations that are sound only in certain cases. These translations also evolve frequently over time, as new techniques and features are developed or optimized; ideally a formal approach to validation should provide means of minimizing the impact of the exchange of one translation for another.

3. Non-locality: The soundness of the translation of a fragment of the input program may depend on several checks that are performed at different places in the IVL program. For instance, the translation of a procedure call might be sound only because well-formedness of the procedure specification has been checked elsewhere in the generated IVL code. Such non-local checks are commonly used to speed up verification, for instance, to check well-formedness conditions once

99 and for all rather than each time a specification is used. However, they complicate the soundness
100 argument, which needs to somehow track the dependencies on properties checked elsewhere.
101
102 *This paper.* We present the first approach for enabling automatic formal validation for existing
103 implementations of the front-end translations employed in many practical program verifiers.
104 This validation guarantees front-end soundness and, thus, makes automatic program verifiers
105 substantially more trustworthy.
106 The core of our approach is a general methodology for generating *forward simulations* [Lynch
107 and Vaandrager 1995] between the statements of the input and the IVL program in a modular way.
108 Our methodology provides solutions to the three challenges above. It (1) bridges the semantic gap
109 with a novel approach by which the simulation proof is split into smaller simulations, (2) supports
110 diverse translations by modularising the proof in terms of our smaller simulations, and (3) handles
111 non-locality by systematically and formally tracking dependencies during a simulation proof.
112 For concreteness, we present our methodology for the translation from a core fragment of Viper
113 to Boogie, as implemented in a pre-existing and actively-used verification tool. This translation is
114 significant both because it exhibits all of the challenges discussed above and because both Viper
115 and Boogie are widely used. For instance, Viper is used in Gobra (Go) [Wolf et al. 2021], Prusti
116 (Rust) [Astrauskas et al. 2019], Nagini (Python) [Eilers and Müller 2018], VerCors (Java) [Blom
117 et al. 2017], and Gradual C0 [DiVincenzo et al. 2022]. The soundness of each of these tools relies
118 on the Viper verifiers being sound. Note that these tools use Viper as an IVL, but for the purpose
119 of this paper, we will treat it solely as a front-end language that is translated to Boogie. While
120 our methodology is phrased in terms of Viper and Boogie, we have designed our approach, which
121 solves the three key challenges above, to generalise to other front-end translations, for example,
122 the Dafny-to-Boogie translation.
123
124 *Contributions.* We make the following technical contributions:

125 • We develop a general methodology for the automated validation of front-end translations
126 based on forward simulation proofs. We present this methodology for the translation from
127 Viper to Boogie. As a foundation for the generated proofs, we formalise a semantics for a
128 core subset of Viper in Isabelle and connect this with an existing Isabelle formalisation for
129 Boogie [Parthasarathy et al. 2021].
130 • We instrument the existing Viper-to-Boogie implementation such that, for a subset of Viper,
131 it automatically produces a proof in Isabelle justifying the soundness of the translation. These
132 proofs can be checked independently in Isabelle, which ensures front-end soundness of the
133 Viper verifier.
134 • Our evaluation on a diverse set of Viper programs demonstrates our approach's effectiveness:
135 we were able to generate proofs and check them in Isabelle fully automatically in all cases.
136 • As part of the consistency proof for the axiomatisations used in Boogie programs, we provide
137 the first approach to formally deal with a restricted version of Boogie's (impredicatively-
138)*polymorphic maps* [Leino and Rümmer 2010].
139
140 *Outline.* Sec. 2 provides the necessary background on Viper and Boogie. Sec. 3 introduces our
141 forward simulation methodology for relating Viper and Boogie statements. Sec. 4 presents how
142 we formally validate the existing implementation of the Viper-to-Boogie translation using our
143 forward simulation methodology. Sec. 5 evaluates the proofs generated by our instrumentation.
144 Sec. 6 presents related work and Sec. 7 concludes. **All our technical results (Sec. 3, Sec. 4) have**
145 **been proved in Isabelle; the mechanisation and an appendix that we refer to is included**
146 **in the supplementary material.**
147

$$VExpr \ni e ::= x \mid lit \mid e.f \mid e\ bop\ e \mid uop(e) \qquad VAssert \ni A ::= e \mid \textbf{acc}(e.f, e) \mid A * A \mid e \Rightarrow A \mid e\ ?\ A : A$$
$$VStmt \ni s ::= x := e \mid e.f := v \mid \vec{y} := m(\vec{x}) \mid m(\vec{x}) \mid \textbf{var}\ x : \tau \mid \textbf{inhale}\ A \mid \textbf{exhale}\ A \mid \textbf{assert}\ A \mid$$
$$s; s \mid \textbf{if}(e)\ \{s\}\ \textbf{else}\ \{s\}$$

$$BExpr \ni e_b ::= x \mid lit_b \mid e_b\ bop\ e_b \mid uop(e_b) \mid f[\vec{\tau_b}](\vec{e_b}) \mid \forall x : \tau_b.\ e_b \mid \exists x : \tau_b.\ e_b \mid \forall_{ty}\, t.\ e_b \mid \exists_{ty}\, t.\ e_b$$
$$BSimpleCmd \ni c_b ::= \textbf{assume}\ e_b \mid \textbf{assert}\ e_b \mid x := e_b \mid \textbf{havoc}\ x \qquad BStmtBlock \ni b_b ::= \vec{c_b}; if_b$$
$$BIfOpt \ni if_b ::= \textbf{if}(e_b)\ \{s_b\}\ \textbf{else}\ \{s_b\} \mid \textbf{if}(*)\ \{s_b\}\ \textbf{else}\ \{s_b\} \mid \epsilon \qquad BStmt \ni s_b ::= \vec{b_b}$$

Fig. 1. The syntax of our formalised Viper subset (top, blue keywords) and corresponding Boogie subset (bottom, with subscript b, orange keywords) without top-level declarations. τ (τ_b), *bop*, and *uop* denote types, binary and unary operations, respectively.

2 VIPER AND BOOGIE: BACKGROUND AND SEMANTICS

In this section, we present the necessary background on the Viper and Boogie languages We introduce and compare the language subsets targeted by the Viper-to-Boogie translation (Sec. 2.1), give an overview of the operational semantics of Boogie (Sec. 2.2) and Viper (Sec. 2.3), and finally show an example of the translation used by the pre-existing Viper verifier implementation (Sec. 2.4).

2.1 The Viper and Boogie languages

Viper programs in the subset considered here consist of a set of top-level declarations of fields (reference-field pairs are used to access the heap) and methods. Boogie programs consist of a set of top-level declarations of global variables, constants, uninterpreted (polymorphic) functions, type constructors, axioms (which constrain the constants and functions), and procedures. Both languages are imperative and separate *statements* from *expressions* (whose evaluation have no side-effects) and specification-only *assertions*. The body of each Viper method / Boogie procedure is a statement. Viper methods have pre- and post-conditions (assertions); method calls are verified modularly against these assertions.[1] In Viper, variables can be declared within statements; Boogie procedures declare all variables upfront. Our supported Viper and Boogie statements, assertions, and expressions are shown in Fig. 1. Both languages have the same control flow elements and have some built-in types in common (e.g. Booleans and integers). Viper additionally provides a single *reference* type, and supports reading from and writing to heap locations via a field access `e.f`, where `e` is a reference expression and `f` a field.

Our validation generates proofs that connect the abstract syntax tree (AST) of a Viper program with the AST of the corresponding Boogie program[2]. Proof generation is complicated by the fact that the Viper and Boogie AST are structured differently. As shown in Fig. 1, the Viper AST uses a standard *sequential composition* $s_1; s_2$, whereas a Boogie statement is given by a list of *statement blocks*. Each statement block $\vec{c_b}; if_b$ consists of a list of *simple commands* (i.e. no control flow), followed by either an if-statement or empty statement (ϵ).

As is typical for verifiers for higher-level languages, Viper's verification methodology employs a custom advanced program logic, in this case based on a flavour of separation logic called *implicit dynamic frames* (IDF) [Parkinson and Summers 2012; Smans et al. 2012] which reasons about the heap via *permissions*. Viper's assertions include the *accessibility predicate* **acc**`(e.f,p)`, which represents

[1] Boogie supports pre-/post-conditions and procedure calls, but they are not used by the Viper-to-Boogie implementation.
[2] The actual verifier implementation produces a Boogie AST by generating and then parsing a text file. Targeting the resulting AST directly avoids the need to trust a Boogie parser, and also generalises to verifier implementations that choose to directly target the Boogie AST such as Dafny's.

a *resource* (a logical notion which can be neither freely fabricated nor duplicated): the fractional (p) amount of *permission to access heap location* `e.f` [3]. Fractional permission amounts [Boyland 2003] range between 0 and 1; nonzero permission is required to *read* heap locations and full (1) permission is required to *write to* heap locations. $A * B$ expresses the *separating conjunction* from separation logic, which specifies that the permissions in A and B must *sum up to an amount currently held*. One difference between IDF and separation logic is that IDF (and thus, Viper) supports general heap-dependent expressions such as `x.val = 5` or `x.f.f`, whose evaluation is *partial* (only allowed with suitable permissions); this necessitates a notion of *well-definedness* checks on expressions (see Sec. 2.3). Boogie does not provide built-in heap reasoning, and uses a much simpler program logic: its assertions are (total) formulas in first-order logic.

The presence of a heap in Viper also results in a very different state model. A Viper state consists of a variable store, a heap (mapping heap locations to current values) and a *permission mask* (mapping heap locations to current permission amounts); a Boogie state is simply a variable store.

The main Viper features *not* included in our subset are loops, more-complex resource assertions (predicates, magic wands, iterated separating conjunctions), heap-dependent functions, and domains. Adding support for loops is straightforward: their semantics can be desugared via their invariant, in a pattern similar to method calls that we already support. For other features more work would be required, but we are confident that these extensions would fit within in our general methodology.

2.2 Boogie Semantics

We extend the operational Boogie semantics formalised in Isabelle by Parthasarathy et al. [2021] to support the statements in Fig. 1, and reuse many components including the state model and the semantics of simple commands. The semantics of Boogie statements is expressed via program executions. A finite program execution has one of three outcomes: (1) it *fails*, because an **assert** e command is reached in a state that does not satisfy the Boolean expression e, (2) it *stops*, because an **assume** e command is reached in a state that does not satisfy the Boolean expression e, or (3) it *succeeds*, because neither of the first two situations occur. The three outcomes are represented formally via: (1) a failure outcome F, (2) a *magic* outcome M for when the execution stops, and (3) a *normal* outcome $\mathsf{N}(\sigma_b)$ in all other cases, where σ_b is the resulting Boogie state, which is given by a mapping from variables to values. Assignments and havoc commands always succeed; **havoc** x nondeterministically assigns a value of x's declared type to x.

Formally, executions of Boogie statements are expressed via a small-step semantics. The judgement $\Gamma_b \vdash (\gamma, \mathsf{N}(\sigma_b)) \rightarrow_{\mathrm{b}}^{*} (\gamma', r_b)$ expresses a finite execution w.r.t. *Boogie context* Γ_b that takes 0 or more steps starting from the *program point* γ and Boogie state σ_b, and ending in the program point γ' and outcome r_b. A Boogie context includes the interpretation of uninterpreted types and functions, and the types of declared variables. A program point is given by a pair of the currently active statement block b and the continuation representing the statement to be executed after b. A continuation is either the empty continuation (i.e. there is nothing to execute) or a sequential continuation (i.e. a statement block followed by a continuation). A continuation-based small-step semantics avoids local search rules commonly required in a small-step semantics [Appel and Blazy 2007].

2.3 Viper Semantics

To our knowledge, there is no mechanised semantics for any fragment of the Viper language; we outline the main points of the one we have formalised here. We give a big-step operational semantics to Viper statements via program executions again with three possible outcomes for finite

[3] For readers familiar with separation logics, this is analogous to a fractional points-to assertion in a separation logic.

246 executions: *failure* F, *magic* M, and *normal outcomes* $\mathsf{N}(\sigma_v)$ where σ_v is a *Viper state*. A Viper state
247 σ_v comprises a local variable mapping $\mathrm{st}(\sigma_v)$, a heap $h(\sigma_v)$ (a total mapping from heap locations to
248 values), and a permission mask $\pi(\sigma_v)$ (a total mapping from heap locations to permission amounts).
249 The judgement $\Gamma_v \vdash \langle s, \sigma_v \rangle \rightarrow_{\mathsf{v}} r_v$ holds if, in the *Viper context* Γ_v (fixing the declarations of methods,
250 fields and local variables) for statement s in the state σ_v terminates with outcome r_v. Determining
251 the outcome of a Viper execution is much more complex than for Boogie; for example, our semantics
252 takes care that all Viper states have *consistent permission masks* (mapping each location to values
253 between 0 and 1); executions that would produce inconsistent states in this sense are pruned by
254 going to M.
255 Formalising expression evaluation requires care for Viper, since, in a given state, not even all
256 type-correct expressions are *well-defined*: in our subset this can be either because of (1) division
257 by zero, or (2) dereferencing a heap location for which no permission is held (subsuming null
258 dereferences). In our semantics, evaluating an ill-defined expression causes execution to fail (in
259 contrast to Boogie, where expression evaluation is *always* allowed and defined). Our judgement
260 $\langle e, \sigma_v \rangle \Downarrow \mathsf{V}(v)$ expresses that expression e evaluates to a value v in state σ_v (in particular, e is
261 well-defined in σ_v) and $\langle e, \sigma_v \rangle \Downarrow \lightning$ expresses that e is ill-defined in σ_v.
262 Viper uses two main statement primitives to encode separation logic reasoning: (1) **inhale** A
263 adds the permissions specified by assertion A to the state, and *stops* any execution where either a
264 logical constraint in A does not hold (these are *assumed*) or the added permissions would yield an
265 inconsistent permission mask. (2) **exhale** A *removes* the permissions specified by A, and *fails* if
266 either insufficient permissions are held or if a constraint in A does not hold; for any heap locations
267 to which *all permission was removed*, an **exhale** also non-deterministically assigns arbitrary values[4]
268 This non-deterministic assignment reflects the fact that, while our Viper states employ total heaps
269 (as is typical for IDF formalisms [Parkinson and Summers 2012]), the values stored in heap locations
270 without permission should be completely unconstrained.
271 **inhale** and **exhale** operations are typically used in Viper to encode external or more-complex
272 operations [Müller et al. 2016]. For instance, a Viper method calls can be desugared into exhaling the
273 precondition and then inhaling the postcondition of the callee; the nondeterministic assignments
274 made by the **exhale** model possible side effects of the call. We present here some of the key rules
275 for **exhale**, which will be used later in this paper. Additional rules for **inhale** are presented in
276 the appendix (App. A); the complete rules are included in our Isabelle formalisation.
277 An **exhale** A must cause the loss of heap information (via non-deterministic assignments) in
278 general, but also needs to check that logical constraints *were* true when the exhale started. Our
279 semantics for **exhale** A first removes the permissions and checks the constraints specified in A
280 *without changing the heap yet* via an intermediate operation **remcheck** A; only then, it applies
281 nondeterministic assignments. The inference rule EXH-SUCC in Fig. 2 formalises this behaviour for
282 the case when **exhale** A succeeds. The big-step judgement $\sigma_v \vdash \langle A, \sigma_v \rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v'')$ defines the
283 successful execution of an **remcheck** A operation from σ_v to σ_v''. nonDet specifies the nondeter-
284 ministic assignment for all heap locations for which **remcheck** A removed all permission. The
285 case when **remcheck** A (and thus **exhale** A) fails, is captured by the rule EXH-FAIL.
286 Our semantics for **remcheck** A decomposes the assertion A from left to right: That is, **remcheck** $A*$
287 B first executes **remcheck** A and then **remcheck** B (rule RC-SEP formalises the case when
288 **remcheck** A succeeds; if **remcheck** A fails, then **remcheck** $A * B$ also fails). However, we need to
289 also take care that the removal of permissions on-the-fly doesn't cause subexpressions to be consid-
290 ered ill-defined, e.g. for the subexpression `x.f==1` in `exhale(acc(x.f)*x.f==1)` which comes
291
292 [4]For separation-logic-versed readers, the Hoare triples $\{R\}$ **inhale** A $\{R * A\}$ and $\{R * A\}$ **exhale** A $\{R\}$ reflect this
293 behavior (assuming the expressions in A and R are well-defined).
294

295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311

$$\frac{\sigma_v \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathrm{rc}} \mathsf{N}(\sigma_v'') \quad \mathsf{nonDet}(\sigma_v, \sigma_v'', \sigma_v')}{\Gamma_v \vdash \langle \mathbf{exhale}\ A, \sigma_v\rangle \rightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v')}\ (\textsc{exh-succ}) \qquad \frac{\sigma_v \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathrm{rc}} \mathsf{F}}{\Gamma_v \vdash \langle \mathbf{exhale}\ A, \sigma_v\rangle \rightarrow_{\mathsf{v}} \mathsf{F}}\ (\textsc{exh-fail})$$

$$\frac{\sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathrm{rc}} \mathsf{N}(\sigma_v') \quad \sigma_v^0 \vdash \langle B, \sigma_v'\rangle \rightarrow_{\mathrm{rc}} r_v}{\sigma_v^0 \vdash \langle A * B, \sigma_v\rangle \rightarrow_{\mathrm{rc}} r_v}\ (\textsc{rc-sep}) \qquad \frac{\langle e, \sigma_v^0\rangle \Downarrow \mathsf{V}(r) \quad \langle e_p, \sigma_v^0\rangle \Downarrow \mathsf{V}(p) \quad r_v = \mathit{if}\ \mathsf{exhAccSucc}(r, p, \sigma_v)\ \mathit{then}\ \mathsf{N}(\sigma_v^R)\ \mathit{else}\ \mathsf{F}}{\sigma_v^0 \vdash \langle \mathbf{acc}(e.f, e_p), \sigma_v\rangle \rightarrow_{\mathrm{rc}} r_v}\ (\textsc{rc-acc})$$

$$\mathsf{nonDet}(\sigma_v, \sigma_v'', \sigma_v') \triangleq \begin{array}{l} \mathsf{st}(\sigma_v') = \mathsf{st}(\sigma_v'') \wedge \pi(\sigma_v') = \pi(\sigma_v'') \wedge \\ \forall l.\ (\pi(\sigma_v)(l) = 0 \vee \pi(\sigma_v'')(l) > 0) \Rightarrow h(\sigma_v')(l) = h(\sigma_v'')(l) \end{array}$$

$$\mathsf{exhAccSucc}(r, p, \sigma_v) \triangleq p \geq 0 \wedge (r = \mathbf{null}\ ?\ p = 0 : \pi(\sigma_v)(r.f) \geq p) \qquad \sigma_v^R \triangleq \mathsf{rem}(\sigma_v, r, f, p)$$

Fig. 2. A subset of the rules for the formal semantics of exhale. $\mathsf{rem}(\sigma_v, r, f, p)$ is the state σ_v where permission p is removed from $r.f$.

312 313 314 315 316 317

after the permission to `x.f` is removed. Thus, our judgement carries both an *expression evaluation state* (σ_v^0 in RC-SEP) in which expressions are evaluated and a *reduction state* (σ_v and σ_v' in RC-SEP) from which permissions are removed. Rule RC-ACC for **remcheck acc**$(e.f, e_p)$ models removing e_p permission for heap location $e.f$. The operation succeeds (expressed by $\mathsf{exhAccSucc}(r, p, \sigma_v)$) iff (1) the to-be-removed permission is nonnegative and, (2) there is sufficient permission.

318

2.4 Example Viper-to-Boogie Translation

319 320 321 322 323 324 325

To give a flavour of a translation of a Viper statement into a Boogie, consider Fig. 3, which shows a simplified translation used by the standard Viper-to-Boogie implementation. The Viper statement first adds permission to `x.f`, then updates `y.g`, and finally removes the added permission to `x.f` and checks that `y.g` is greater than `x.f`. This sequence of operations occurs, for instance, when verifying a method with the permission to `x.f` as precondition, the field update as method body, and the exhaled assertion as postcondition.

326 327 328 329 330 331 332 333 334

The corresponding Boogie program is significantly larger. The **inhale** is encoded on lines 1-4, the assignment is encoded on lines 5-7, and the **exhale** is encoded on lines 8-18. The Boogie program uses map-typed variables `H` and `M` to model the heap and permissions, respectively.[5] The uninterpreted function `GoodMask` expresses when a permission map is consistent; an axiom constrains the function correspondingly. The permission mask of the expression evaluation state during the **remcheck** operation is captured by the auxiliary variable `WM` (line 8). The corresponding nondeterministic assignment of heap values is performed on line 16. Even this tiny snippet of code illustrates the explosion in concerns, complexity and the inobvious mapping between concepts in one language and the other, all of which must be taken care of for formal translation validation.

335 336

3 A FORWARD SIMULATION METHODOLOGY FOR FRONT-END TRANSLATIONS

337 338 339 340

A translational verifier is *sound* iff the correctness any input program is implied by the correctness of the correspondingly-translated IVL program. In our setting: a Viper program (resp. a Boogie program) is *correct* if each of its methods (resp. procedures) is correct. At a high level (details in Sec. 4.5), a method (resp. procedure) is correct if its body has no failing executions. Our goal, for a

341 342 343

[5]The notation `m[a]` is syntactic sugar here. We describe in Sec. 5 how maps are represented using the subset from Fig. 1.

```
inhale acc(x.f, q)
y.g := x.f+1
exhale acc(x.f, q) * y.g > x.f
```

⇝

```
1  tmp := q; assert tmp >= 0
2  assume tmp > 0 ==> x != null;
3  M[x,f] += tmp
4  assume GoodMask(M)
5  assert M[x,f] > 0; assert M[y,g] == 1
6  H[y,g] := H[x,f]+1
7  assume GoodMask(M)
8  WM := M, tmp := q;
9  assert tmp >= 0
10 if(tmp != 0) {
11   assert M[x,f] >= tmp
12 }
13 M[x,f] -= tmp
14 assert WM[y,g] > 0; assert WM[x,f] > 0
15 assert H[y,g] > H[x,f]
16 havoc H'; assume idOnPositive(H,H',M)
17 H := H'
18 assume GoodMask(M)
```

Fig. 3. A Viper statement (on the left) and the corresponding (simplified) Boogie statement (on the right) that is emitted by the current Viper-to-Boogie implementation.

given run of the Viper-to-Boogie translation, is to generate *automatically* a formal proof that shows that *if* the Viper program has a failing execution, the translated Boogie program has one also.

We generate such proofs via a novel general methodology for proving *forward simulations* [Lynch and Vaandrager 1995] between source and IVL target statements. We observed early on that generating such simulation proofs directly based on global knowledge of the entire translation would require handling the entire semantic gap between the source and target languages monolithically in one result, which would be both unfeasible to automate effectively and highly-brittle to any changes in the translation.

Instead, our methodology employs a combination of key technical strategies that work together to achieve reliable and robust automation of our formal simulation results: (1) syntactic and semantic *decompositions* into smaller and more-focused simulation sub-results that are easier to automate, (2) *generic simulation judgements* which can be instantiated to obtain the diverse simulation notions we require, (3) *generic composition lemmas* which factor out common idioms arising in diverse facets of the overall translation, and (4) *contextual hypotheses* which can be injected into specific simulation proofs to handle non-locality of certain translation checks. We present these key ingredients of our methodology in the remainder of this section. We illustrate them concretely for Viper and Boogie, but they can be naturally ported to other front-end translations because they are designed to abstract over the specifics of states, relations and specific statements employed in a translation.

3.1 Focusing Forward Simulation Proofs by Decomposition

Intuitively, a forward simulation between a Viper and a Boogie statement shows that for any execution of the Viper statement, there exists a corresponding execution of the Boogie statement that *simulates* it. By defining the simulation such that a *failing* Viper execution is simulated only by *failing* Boogie executions, a forward simulation implies our desired result in particular.

393
394
395
396
397
398
399
400
401
402
403
404

$$\mathsf{sim}_{\Gamma_b}(R_{in}, R_{out}, Succ, Fail, \gamma_{in}, \gamma_{out}) \triangleq \forall \tau, \sigma_b.\ R_{in}(\tau, \sigma_b) \Longrightarrow$$
$$\big(\forall \tau'.\ Succ(\tau, \tau') \Longrightarrow \exists \sigma_b'.\ \Gamma_b \vdash (\gamma_{in}, \mathsf{N}(\sigma_b)) \rightarrow_{\mathsf{b}}^{*} (\gamma_{out}, \mathsf{N}(\sigma_b')) \wedge R_{out}(\tau', \sigma_b')\big) \wedge \quad \text{(Success case)}$$
$$\big(Fail(\tau) \Longrightarrow \exists \gamma'.\ \Gamma_b \vdash (\gamma_{in}, \sigma_b) \rightarrow_{\mathsf{b}}^{*} (\gamma', \mathsf{F})\big) \quad \text{(Failure case)}$$

$$\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', s, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}(R, R', \lambda\sigma_v\ \sigma_v'.\ \Gamma_v \vdash \langle s, \sigma_v\rangle \rightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v'), \lambda\sigma_v.\ \Gamma_v \vdash \langle s, \sigma_v\rangle \rightarrow_{\mathsf{v}} \mathsf{F}, \gamma, \gamma')$$

$$\mathsf{wfSim}_{\Gamma_b}(R, R', es, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}\left(\begin{array}{l} R, R', (\lambda\sigma_v\ \sigma_v'.\ \sigma_v = \sigma_v' \wedge \exists vs.\ \langle es, \sigma_v\rangle[\Downarrow]\mathsf{V}(vs)), \\ (\lambda\sigma_v.\ \langle es, \sigma_v\rangle[\Downarrow]\lightning), \gamma, \gamma' \end{array}\right)$$

$$\mathsf{rcSim}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}\left(\begin{array}{l} R, R', (\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \sigma_v^0 = \sigma_v^1 \wedge \sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')), \\ (\lambda(\sigma_v^0, \sigma_v)\ \sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{F}), \gamma, \gamma' \end{array}\right)$$

405 Fig. 4. The definition of the generic forward simulation judgement and three common instantiations. The
406 judgement $\langle es, \sigma_v\rangle[\Downarrow]r$ lifts the judgement for the evaluation of an expression (see Sec. 2.3) to a list of
407 expressions es.

408
409
410 To tackle the complexity of automatically (and reliably) generating simulation proofs in general
411 for the Viper-to-Boogie translation, we employ a variety of strategies for aggressively decomposing
412 the desired simulation result into smaller and simpler sub-goals that are themselves still simulation
413 results. These decompositions are sometimes intuitive based on the syntax: for example, in the case
414 of decomposing simulation of a Viper sequential composition into simulations for its constituent
415 statements. However, we go *further than the syntax*, decomposing across different *semantic concerns*
416 for the *same* Viper statement, into what we call Viper *effects*.
417 For example, we discussed in Sec. 2.3 that the semantics of **exhale** consists of two effects,
418 **remcheck** and a nondeterministic assignment. The atomic simulation proofs for each of these
419 Viper effects are made separately, and then composed for a simulation proof for the primitive
420 statement as a whole; this would in turn be composed with simulation proofs for other sequentially-
421 composed statements, and so on. Note in particular, that atomic simulation proofs may need to relate
422 only a *part of* the semantics of a Viper statement to some appropriate Boogie code, a technicality
423 which requires special care when tracking the *relations* between corresponding states in the two
424 programs.
425 Via our decompositions, each atomic simulation proof focuses on a different specific semantic
426 concern with respect to the translation in question; these proofs can be made simple enough to
427 discharge automatically, optionally with tailored tactics. However without care, our decomposition
428 approach could lead easily to an explosion of ad hoc simulation judgements with disparate forms
429 and parameters. Instead, our simulation methodology defines a *single, generic* simulation judgement
430 which can be instantiated appropriately to define each particular simulation judgement required.
431 We design our generic judgements to support instantiations which reflect not only the semantics
432 of the particular effect in isolation, but to optionally include additional contextual information to
433 be propagated to specialise and aid the simulation proof itself.
434
435 ### 3.2 One Simulation Judgement to Rule Them All
436 Our generic forward simulation judgement sim is defined in Fig. 4. All concrete forward simulations
437 (e.g. for statements, well-definedness checks, etc.) are instantiations of this judgement. As well as
438 aiding understanding, this approach enables both tactics which manipulate this generic judgement
439 directly, and *generic composition proof rules* which embody recurring proof idioms in a way which
440 is again parametric with the specific simulation in question (Sec. 3.3).
441

sim is defined in terms of multiple parameters: (1) the Boogie context Γ_b, (2) an *input relation* R_{in} and an *output relation* R_{out} on Viper and Boogie states, (3) a *success predicate* $Succ$ characterising the set of input and output Viper state pairs (τ, τ') for which there is a successful Viper execution from τ to τ', (4) a *failure predicate* $Fail$ characterising the set of input Viper states that result in a failing execution, (5) input and output Boogie program points γ and γ' where the Boogie executions are expected to start and end, respectively. The success and failure predicate together abstractly describe the set of Viper executions that must be shown to be simulated.

$\mathsf{sim}_{\Gamma_b}(R_{in}, R_{out}, Succ, Fail, \gamma_{in}, \gamma_{out})$ holds iff for any Viper and Boogie input states related by R_{in}, the following two conditions hold: (1) if the Viper execution from the input Viper state is successful for some output Viper state τ', then there must be a Boogie execution from program point γ_{in} and the input Boogie state to program point γ_{out} and some output Boogie state that is related to τ' by R_{out}, and (2) if the Viper execution fails in the input state, then there must be a failing Boogie execution from γ_{in} and the input Boogie state (the reached Boogie program point need not be γ_{out}). The second condition is the end goal that we need to show soundness of the Viper-to-Boogie translation. The first condition is needed in order to derive sim compositionally; it guarantees, for example, that not all Boogie executions for a successful Viper execution produce a magic outcome.

Three important instantiations of sim that we use are shown at the bottom of Fig. 4. stmSim is the forward simulation for Viper statements, where the success and failure predicates are instantiated to be a successful and a failing Viper statement reduction, respectively. Thus, the resulting failure case in sim directly gives us the key property to show the soundness of a Viper-to-Boogie translation. wfSim is the forward simulation for the well-definedness check of a list of Viper expressions. Here, the instantiation of the success predicate explicitly expresses that the Viper state does not change during the evaluation of expressions. rcSim is the forward simulation for `remcheck`. Here, the instantiation makes use of the fact that the generic simulation judgement sim is in fact also (implicitly, here) parametric with the notions of states employed: the "Viper state" is in fact instantiated to be a pair of standard Viper states in this case, where the first Viper state represents the expression evaluation state and the second Viper state represents the reduction state (see Sec. 2.3 for this distinction). The success predicate expresses that the expression evaluation state does not change during an `remcheck` operation. These three common instantiations are all expressed directly via the Viper reduction judgements introduced in Sec. 2.3. Like the generic simulation judgement, the three instantiations *are themselves generic*, abstracting away *how* the Viper and Boogie states are related by taking the input and output state relations as parameters. As we will show in Sec. 3.4, we also use instantiations that do not just use Viper reduction judgements (e.g. to express the non-deterministic assignment of heap values in `remcheck`).

3.3 Instantiation-Independent Rules

Many simulation idioms arise repeatedly (but differently) in a complex translation. Notions of sequential composition, conditional evaluation, stuttering steps are all good example idioms, which require a certain stylised formal justification to reason about. Our generic simulation judgement allows us to identify and formalise these idioms once and for all, providing, for example generic composition lemmas that can be proved once and instantiated for different purposes. In this subsection, we visualise these idioms as inference rules, but in our Isabelle formalisation they are expressed and proved as regular lemmas.

For example, we prove a single general composition rule from which we derive concrete rules to combine (1) simulations of s_1 and s_2 to a simulation of $s_1; s_2$, (2) simulations of `remcheck` A_1 and `remcheck` A_2 to `remcheck` $A_1 * A_2$, (3) simulations of `inhale` A_1 and `inhale` A_2 to `inhale` $A_1 * A_2$. The general composition rule COMP in Fig. 5 captures the composition of two, possibly different, instantiations of sim, where the output relation and Boogie program point of the first instantiation

491 492 493 494 495 496

$$\frac{\begin{array}{c}\mathsf{sim}_{\Gamma_b}(R, R'', S_1, F_1, \gamma, \gamma'') \\ \mathsf{sim}_{\Gamma_b}(R'', R', S_2, F_2, \gamma'', \gamma') \\ \forall \tau, \tau'.\ S(\tau, \tau') \Rightarrow \exists \tau''.\ S_1(\tau, \tau'') \wedge S_2(\tau'', \tau') \\ \forall \tau.\ F(\tau) \Rightarrow F_1(\tau) \vee \exists \tau''.\ S_1(\tau, \tau'') \wedge F_2(\tau'')\end{array}}{\mathsf{sim}_{\Gamma_b}(R, R', S, F, \gamma, \gamma')}\ (\textsc{comp}) \qquad \frac{\begin{array}{c}\mathsf{bSim}_{\Gamma_b}(R, R_1, \gamma, \gamma_1) \\ \mathsf{sim}_{\Gamma_b}(R_1, R_2, S, F, \gamma_1, \gamma_2) \\ \mathsf{bSim}_{\Gamma_b}(R_2, R', \gamma_2, \gamma')\end{array}}{\mathsf{sim}_{\Gamma_b}(R, R', S, F, \gamma, \gamma')}\ (\textsc{bprop})$$

497 498 499 500 501

$$\frac{\begin{array}{c}\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R'', s_1, \gamma, \gamma'') \\ \mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R'', R', s_2, \gamma'', \gamma')\end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', (s_1; s_2), \gamma, \gamma')}\ (\textsc{seq-sim}) \qquad \textit{where} \qquad \begin{array}{l}\mathsf{bSim}_{\Gamma_b}(R, R', \gamma, \gamma') \triangleq \\ \mathsf{sim}_{\Gamma_b}(R, R', \lambda\tau\ \tau'.\ \tau = \tau', \lambda_.\ \bot, \gamma, \gamma')\end{array}$$

502 Fig. 5. The instantiation-independent rules COMP and BPROP and the concrete rule for the simulation of $s_1; s_2$.

503 504 505 506 507 508

$$\frac{\begin{array}{ll}\mathsf{rcSim}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)], R_1, A, \gamma, \gamma_1) & (\text{sim. of } \mathbf{remcheck}\ A) \\ \mathsf{sim}_{\Gamma_b}(R_1, [\lambda(_, \sigma_v)\ \sigma_b.\ R'(\sigma_v, \sigma_b)], \mathit{Succ}_2, \lambda_.\ \bot, \gamma_1, \gamma') & (\text{non-det. selection})\end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', \mathbf{exhale}\ A, \gamma, \gamma')}\ (\textsc{exh-sim})$$

509

$$\mathit{Succ}_2 \triangleq \lambda(\sigma_v^0, \sigma_v)\ (_, \sigma_v').\ \mathsf{nonDet}(\sigma_v^0, \sigma_v, \sigma_v') \wedge \sigma_v^0 \vdash \langle A, \sigma_v^0 \rangle \rightarrow_{\mathsf{rc}} \sigma_v$$

510 511 Fig. 6. Rule for the simulation of **exhale** A. The definition of nonDet is given in Fig. 2.

512 513

514 match the input relation and program point of the second one. The two final premises constrain
515 the resulting success and failure predicates. In particular, the composed Viper execution should fail
516 only if either the first instantiation fails or if the second instantiation fails in a state successfully
517 reached by the first one. The rule SEQ-SIM in Fig. 5 shows the concrete composition rule for $s_1; s_2$,
518 which is derived from COMP. Note that SEQ-SIM does not impose any constraints on the Boogie
519 program points, which is crucial to handle Viper's and Boogie's disparate ASTs (see Sec. 2.1).
520 As a second example, the notion of simulation *stuttering steps* also arises in many diverse ways,
521 whenever some auxiliary Boogie code is generated as a translation feature without fully reflecting
522 a step in the Viper source. This includes initialisations of auxiliary variables, or Boogie **assume**
523 statements for properties from the current simulation state relation. This idiom is captured by the
524 *Boogie propagation rule* BPROP in Fig. 5, in which bSim expresses simulations in which only the
525 Boogie state may change (causing adjustment to the state relations).
526

527 ### 3.4 Examples: Generic Decomposition in Action

528 As outlined above, the general strategy for our simulation methodology is to decompose our
529 simulation goals as far as possible, while leaving as many parameters generic as we can to enable
530 maximal reuse of our results and composition lemmas. While decomposition handles the semantic
531 gap, our use of generic parameterisation provides the abstraction necessary to address the diverse
532 translations used in practical translational verifiers. In the following, we showcase our methodology
533 on one rule, but the same ideas apply to all our formal rules (see a second example in App. B).
534 Consider the rule EXH-SIM for the simulation of **exhale** A in Fig. 6. The first premise is expressed
535 as a simulation of the first effect, **remcheck**, which we can express via the rcSim instantiation
536 (see Fig. 4). The second premise models nondeterministic assignment, which is captured by the
537 first conjunct nonDet of the corresponding success predicate and by the failure predicate, which
538 reflects that the nondeterministic assignment cannot fail.
539

540
541
542
543
544

$$\frac{\mathsf{rcInvSim}^{Q}_{\Gamma_v}(R, R'', A_1, \gamma, \gamma'') \quad \mathsf{rcInvSim}^{Q}_{\Gamma_v}(R'', R', A_2, \gamma'', \gamma') \qquad \forall \sigma_v^0, \sigma_v.\ Q(A_1 * A_2, (\sigma_v^0, \sigma_v)) \Rightarrow \left(\begin{array}{l} Q(A_1, (\sigma_v^0, \sigma_v)) \wedge \\ \forall \sigma_v'.\ \sigma_v^0 \vdash \langle A_1, \sigma_v \rangle \rightarrow_{\mathrm{rc}} \mathsf{N}(\sigma_v') \Rightarrow Q(A_2, (\sigma_v', \sigma_v^0)) \end{array} \right)}{\mathsf{rcInvSim}^{Q}_{\Gamma_v}(R, R', (A_1 * A_2), \gamma, \gamma')} \text{(RSEP-SIM)}$$

545
546

$$\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \mathsf{rcSim}_{\Gamma_b}((\lambda \tau, \sigma_b.\ R(\tau, \sigma_b) \wedge Q(A, \tau)), R', A, \gamma, \gamma')$$

547
548
549

Fig. 7. The instantiation for simulating **remcheck** A with assertion predicate Q (bottom) and the corresponding rule for the separating conjunction (top).

550
551
552
553
554
555
556
557
558
559
560
561
562

By modularly abstracting over the details of these premises, as well as the precise definitions of the states and state relations (e.g. the intermediate relation R_1 in this rule), we obtain robustness to diverse translations: crucially, our rules do not constrain *which exact Boogie statements* correspond to a Viper effect. For example, the Viper-to-Boogie implementation establishes the nondeterministic heap assignment premise in EXH-SIM in two different ways depending on whether the assertion contains an accessibility predicate **acc**$(e.f, e_p)$ or not. In the latter case, the implementation does not emit any Boogie code for the nondeterministic assignment, which is sound, since no permission is removed. We are able to justify even this special case with the exact same rule because the success predicate in the premise includes the fact that the current Viper state was reached via **remcheck** A. This allows us, when proving the third premise, to conclude that the nondeterministic assignment would have no effect.

563
564
565
566
567
568

Note that this genericity does not prevent the rule from exploiting specific contextual information: for example, the input state relation of the first premise makes explicit that at the beginning of the **remcheck** A effect the expression evaluation state state and the reduction state are the same. This property does not hold in general for executions of **remcheck** (e.g. it might not hold when performing the effect on the second conjunct of a separating conjunction), but it does hold here, at the beginning of an **exhale**.

569
570

3.5 Injecting Non-Local Hypotheses into Simulation Proofs

571
572
573
574
575

Our rules are designed to be parametric in the state relation between the Viper and Boogie state and permit adjusting this state relation at different points in the simulation proof (e.g. via the Boogie propagation rule BPROP in Fig. 5). In principle, this allows the injection of arbitrary non-locally-justified hypotheses into all of our simulation judgements. However, automating the *usage* of general logical assumptions embedded into our state relations can become a challenge in itself.

576
577
578
579
580

For example, in cases that we will discuss in detail in Sec. 4.2, the Viper-to-Boogie implementation omits the well-definedness checks of expressions in the translation of **remcheck** A and **inhale** A. This is justified, because A is checked to be *well-formed* non-locally in those cases, but to *use* this additional hypothesis requires propagating and adjusting it through the cases of the definition of **remcheck** A.

581
582
583
584
585

As a last key ingredient of our methodology, to avoid these recurring adaptations and proof steps, we allow *specialised* instantiations of the generic forward simulation judgement sim that encapsulate these extra hypotheses as *additional* premises. Consequently, applications of the rule can work with a fixed state relation and replace recurring proof steps by the justification of an additional premise.

586
587

Fig. 7 shows (at the bottom) an instantiation of sim that expresses the simulation of **remcheck** A, parameterised with a predicate Q on assertions. Its definition in terms of rcSim requires $Q(A, \tau)$ to

588

589 hold as part of the input state relation. The specialised rule RSEP-SIM (top of Fig. 7) for **remcheck** $A_1 *$
590 A_2 decomposes the simulation into simulations for A_1 and A_2. Both sub-simulations use the *same*
591 predicate Q, such that applications of the rule do *not* need to adjust the state relations explicitly to
592 reflect that, for example, Q holds for A_1 and A_2 in the respective states. This property is ensured by
593 the third premise. In practice, for a specific Q, we prove the third premise once and for all for all
594 assertions A_1 and A_2, which avoids the recurring proof steps that would be necessary without the
595 specialised rule. Note that the same parameter can be instantiated many different ways to capture
596 different non-local hypotheses for different applications of the same rule.
597
598
599 In summary, our methodology solves all three challenges outlined in the introduction. The
600 *large semantic gap* between input language and IVL is handled by decomposing the statements of
601 the input language into smaller effects and defining for each of them instantiations of a generic
602 forward simulation relation. The parameterisation of this relation allows us, in particular, to capture
603 information about the context in which the effects are execute. This parameterisation also supports
604 *diverse translations* by abstracting from the details of the translation. Finally, *non-locality* is handled
605 by capturing properties checked elsewhere in the state relations, and by devising specialised
606 rules that simplify the proof generation. All of these ideas are needed to validate the existing
607 Viper-to-Boogie translation, but apply equally to other front-end translations.
608

4 PUTTING THE METHODOLOGY TO WORK

609
610 This section presents ideas for applying the methodology from Sec. 3 to concrete front-end transla-
611 tions. In particular, the section presents our instantiation of the state relation (Sec. 4.1), a concrete
612 instance of non-local reasoning (Sec. 4.2), and how our proof automation works (Sec. 4.3). Finally,
613 the section discusses the background theory for Boogie (Sec. 4.4), which includes polymorphic
614 maps, and shows how to use forward simulation proofs to generate the final theorem (Sec. 4.5).
615

4.1 State Relation

616
617 Our rules for deriving forward simulation judgements (Sec. 3) allow us to adjust state relations as
618 needed during a simulation proof. We use this flexibility in many ways, e.g. when (E1) a scoped Viper
619 variable is introduced, (E2) a new auxiliary Boogie variable is introduced, (E3) the Boogie variables
620 tracking the Viper state are changed.To facilitate proof automation for handling such adjustments,
621 we build in a stylised form for expressing state relations for this translation via two parameters: a
622 partial *auxiliary variable map* from auxiliary Boogie variables to *logical conditions they each satisfy*,
623 and a *translation record* specifying how key Viper components are represented in the Boogie state;
624 the scenarios above are all handled by adjusting one of these two parameters. Translation records
625 comprise: (1) a mapping $var(Tr)$ from Viper variables to their Boogie counterparts, (2) the pair
626 $HM(Tr)$ of Boogie variables representing the Viper heap and permission mask (and optionally
627 an additional pair $HM^0(Tr)$ whenever we use a separate expression evaluation state) and (3) a
628 mapping $field(Tr)$ from Viper fields to corresponding Boogie constants.
629 The following definition shows a simplified excerpt of our state relation instantiation for transla-
630 tion record Tr and auxiliary variable map AV, where σ_v and σ_b are the Viper and Boogie states,
631 and σ_v^0 is a distinguished Viper expression evaluation state (if there is none, then $\sigma_v = \sigma_v^0$):
632

$$
\begin{aligned}
&R_{\Gamma_b}^{Tr,AV}((\sigma_v^0, \sigma_v), \sigma_b) \triangleq \mathsf{consistent}(\sigma_v^0) \wedge \mathsf{consistent}(\sigma_v) \wedge \\
&\mathsf{fieldRel}_{\Gamma_b}(field(Tr), \sigma_b) \wedge (\forall x, P.\ AV(x) = P \Rightarrow P(\sigma_b(x))) \wedge \\
&\mathsf{stRel}_{\Gamma_b}(var(Tr), \sigma_v, \sigma_b) \wedge \mathsf{hmRel}_{\Gamma_b}(HM(Tr), \sigma_v, \sigma_b) \wedge \mathsf{hmRel}_{\Gamma_b}(HM^0(Tr), \sigma_v^0, \sigma_b) \wedge \cdots
\end{aligned}
$$

633
634
635
636
637

The first line ensures that the Viper states are consistent. The second ensures that the Viper fields are represented in the Boogie state (fieldRel) and that for each (x, P) in the auxiliary variable map, P holds for the value of x. The third line ensures that the Boogie state correctly captures the Viper state: both in terms of its variable store (stRel) and heap and permission mask (hmRel).

4.2 Non-Locality

For most occurrences of **remcheck** A the translation to Boogie generates well-definedness checks corresponding to expressions evaluated in A. However, specifically in the case of exhaling a method call's precondition, the translation completely omits these well-definedness checks. This is justified by a *non-local check*: the method precondition is checked once-and-for-all to be always well-defined as part of translating the method *declaration*[6].

Given Viper's semantics, our standard simulation proof for **remcheck** A would fail if we did not reflect the consequences of this non-local guarantee *in a way that is used automatically during the proof*. We instantiate the general strategy outlined in Sec. 3.5 for this purpose, which allows us to choose a predicate Q_{pre} on assertions A that will be applied throughout the simulation proof for **remcheck** A. Our strategy requires us to find Q_{pre} such that (a) it is implied by the non-local check elsewhere, and (b) it can be propagated identically to sub-expressions of A during the proof (e.g. satisfying the third premise of RSEP-SIM in Fig. 7, and similarly for other connectives).

In this case, we instantiate the predicate in our strategy with the following definition:

$$Q_{pre}(A, \sigma_v^0, \sigma_v) \triangleq \mathsf{consistent}(\sigma_v^0) \land \exists \sigma_v^i.\ \sigma_v \oplus \sigma_v^i \leq \sigma_v^0 \land \neg\langle A, \sigma_v^i\rangle \rightarrow_{\mathsf{inh}} \mathsf{F}$$

Here, $\oplus$ and $\leq$ (and later, $\ominus$) have standard pointwise meanings on permission masks, leaving heaps and stores identical. This predicate expresses that possibly after restoring some permissions (in σ_v^i) that we *had* at the start of this exhale, at least an *inhale* of A would not fail (in particular, guaranteeing that all expressions within it are well-defined). This matches the non-local check of the method precondition (which effectively checks that an inhale would not fail starting from an *empty* σ_v^i). Showing formally that it can be propagated over connectives occurring in A requires in particular a technical lemma stating a partial *inversion* property between exhale and inhale:

LEMMA 4.1. *Let A be an assertion and σ_v^0, σ_v', σ_v^i, σ_v^s be Viper states, where $\sigma_v^s = \sigma_v^i \oplus (\sigma_v \ominus \sigma_v')$ and σ_v^s is consistent. If $\sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} N(\sigma_v')$ and $\neg\langle A, \sigma_v^i\rangle \rightarrow_{\mathsf{inh}} F$ holds, then $\langle A, \sigma_v^i\rangle \rightarrow_{\mathsf{inh}} N(\sigma_v^s)$.*

We prove this result by induction on the reduction of **remcheck**. The lemma essentially states that the permissions that get removed by **remcheck** A (expressed by $\sigma_v \ominus \sigma_v'$) are exactly those that will be added by a corresponding (non-failing) **inhale** A operation.

4.3 Proof Automation

We developed an Isabelle tactic to automatically generate proofs of our forward simulation judgements for the Viper-to-Boogie translation using hints provided by our lightweight instrumentation of its implementation. Our tactic applies the rules provided by our methodology (Sec. 3.3) to decompose simulations into smaller ones; for atomic simulations we explain our approach below.

A general challenge when applying the rules from Sec. 3 is that the Viper and Boogie ASTs are structured differently (see Sec. 2.1). As a result, the automatic selection of Boogie program points in the premises of rules is not immediate. For example, in the case of rule SEQ-SIM for $s_1; s_2$, we cannot easily choose the intermediate program point γ'' by inspecting the initial program point γ. Instead, we start proving the first premise with an *existentially quantified* γ''. Once the proof reaches a primitive construct such as a Viper assignment, then it becomes clear how to advance the Boogie program γ and by the end of the proof of the first premise, the choice of γ'' becomes

[6]There is an analogous non-local check for m's postcondition that we do not discuss here for simplicity of presentation.

687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735

$$\mathsf{Correct}_b^G(p) \triangleq \forall \mathcal{T}, \mathcal{F}, \sigma_b.\ [\mathsf{DeclsWf}_{G,p}(\mathcal{T}, \mathcal{F}) \wedge \mathsf{AxiomSat}_G(\mathcal{T}, \mathcal{F}, \sigma_b)] \Longrightarrow$$
$$\forall \gamma', r_b'.\ \mathsf{initCtxt}_b^G(p, \mathcal{T}, \mathcal{F}) \vdash (\mathsf{init}_b(p), \mathsf{N}(\sigma_b)) \rightarrow_{\mathsf{b}}^* (\gamma', r_b') \Rightarrow r_b' \neq \mathsf{F}$$

$$\mathsf{Correct}_v^{F,M}(m) \triangleq$$
$$\forall \sigma_v.\ (\forall l. \pi(\sigma_v)(l) = 0) \Longrightarrow$$
$$\forall r_v.\ \mathsf{initCtxt}_v^{M,F}(m) \vdash \langle \textbf{inhale}\ \mathsf{pre}(m); \mathsf{body}(m); \textbf{exhale}\ \mathsf{post}(m), \sigma_v \rangle \rightarrow_{\mathsf{v}} r_v \Rightarrow r_v \neq \mathsf{F}$$

Fig. 8. The correctness definitions for a Boogie procedure p (top) and Viper method m (bottom).

possible. This proof strategy is enabled by the our routine use of *schematic variables* in Isabelle (*evars* in other tools), for postponing the choice of witnesses for existentially-quantified values.

Our instrumentation generates hints for various cases including: (1) to aid some cases of diverse translations (e.g. a hint for when the translation of the usual nondeterministic assignments for **remcheck** A is omitted (when A contains no accessibility predicates), (2) to instantiate the parameters for our strategies for handling non-local checks (cf. the previous subsection), and (3) to suggest when/how to apply the Boogie propagation rule (BPROP in Fig. 5), e.g. when replacing the Boogie variables representing the Viper state elements.

For proving atomic simulations, we use two main automation approaches. Firstly, we prove (once and for all) simple lemmas about the behaviours of small sequences of simple Boogie commands; these are applied (and their hypotheses discharged) automatically when needed. These are used for only small parts of the overall translation; their applicability is robust to most (but not all) changes to the translation over time. Secondly, we employ more-general tactics which abstract over parts of the statements involved (e.g. grouping the effect of a sequence of Boogie **assert** statements).

We apply both approaches in the translation example given by Fig. 3 in Sec. 2. For example, we apply the second approach for the justification of the nonfailure check for **remcheck acc**$(e.f, p)$ shown on lines 9-12. We apply the first approach for the justification of the translation of the nondeterministic heap assignment that is a part of the **exhale** A operation shown on lines 16-17. Note that this Boogie encoding overapproximates the nondeterministic assignment specified by the Viper semantics: assigning new values to *all* locations without permission, rather than only those newly without permission. We still prove the necessary forward simulation automatically; this requires just one Boogie execution that simulates the Viper-required effect precisely.

4.4 Background Theory and Polymorphic Maps

Boogie does not have any notion of a heap location or a Viper state. Such Viper (and other front-end) constructs are translated using particular global declarations in Boogie. A subset of the Boogie declarations always emitted by the Viper-to-Boogie translation is given by:

- Uninterpreted types `bref` and `bfield` to model references and fields. `bfield` takes one type argument indicating the type of the corresponding Viper field.
- An uninterpreted function `goodMask` that maps a permission map to a Boolean and an axiom restricting this function to return true only if the permission map models a consistent Viper permission mask.
- Global variables `H` and `M` to model the heap and permission mask, respectively. `H[x,f]` stores the heap value for heap location `x.f` and `M[x,f]` stores the permission value for `x.f`. The types of both variables are represented via Boogie's *impredicatively-polymorphic maps* [Leino and Rümmer 2010], which we explain below.

736 The correctness of a Boogie procedure guarantees no failing executions of the procedure's body for *any* interpretation of the uninterpreted types and functions for which (1) the function interpretation respects the declared function signatures, and (2) all the Boogie axioms in the Boogie program are satisfied. The formal correctness definition for a Boogie procedure p reflects this directly (a simplified version is shown at the top of Fig. 8). $\mathcal{T}$ and $\mathcal{F}$ are the type and function interpretation, respectively. G denotes the global declarations in the Boogie program. $\mathsf{init}_b(p)$ is the initial Boogie program point in the procedure p. $\mathsf{initCtxt}_b^G(p, \mathcal{T}, \mathcal{F})$ constructs a Boogie context from the provided parameters. Thus, to use the correctness of a Boogie procedure, we must choose a type and function interpretation that satisfy the required conditions. The main challenge here is formally expressing instantiations which deal with polymorphic Boogie maps, as we discuss next.

Polymorphic maps. The heap and permission maps are represented (via the Viper-to-Boogie translation) using Boogie's polymorphic maps; this choice is not unusual (e.g. the Dafny-to-Boogie implementation also currently uses polymorphic maps with similar polymorphic map types as the ones used by Viper-to-Boogie implementation). The Boogie maps used to model Viper heaps have the polymorphic map type `<T>[bref, bfield T]T`: a total map storing, for *any* type `T`, values of type `T` given (as key) a reference and field with type argument `T`.

To the best of our knowledge, there exists no formal model for Boogie's polymorphic maps. Providing a general model is challenging: in particular, Boogie's polymorphic types are *impredicative*: a map type such as `<T>[T]T'`, which permit *any* type of value as a key, including the map itself! Instead of providing a formal model for such polymorphic maps in general, we provide one tailored to the polymorphic maps that the Viper-to-Boogie implementation uses. To aid the incorporation of our model, we adjust the implementation to represent its polymorphic maps via uninterpreted types (`HType`), polymorphic functions `upd` and `read`, and two axioms expressing their expected meanings. The only change in the translation itself is to simply rewrite heap and mask lookups and updates into calls to these functions; everything else remains identical. Then, we provide instantiations of the types and functions, and prove that the axioms hold for these instantiations for any state; the same approach could be used for e.g. the Dafny-to-Boogie translation.

What remains for our simulation proofs is to *instantiate* these new features `HType`, `upd`, and `read` such that the axioms are fulfilled. The challenge here is avoiding circularities: e.g. if the field provided to `read` has type parameter `HType`, then the instantiation of `read` must itself return a heap; to construct an initial heap, we already need a heap of the same type. To break this circularity, we instantiate `HType` as a *partial* mapping from reference and fields to values, and allow the empty map to be of type `HType`, which provides us with a concrete heap. `read` is defined to return a default value for reference and field pairs that are not in the domain of the partial map; for heaps the default value is the empty map. This is sufficient to prove the axioms, since in practice the axioms only require `read` returning specific values when those values were previously inserted by `update`.

4.5 Generating A Proof of the Final Theorem

We will now discuss, given a Viper program and its Boogie translation, how forward simulation proofs can be used to generate a proof of the final theorem justifying the soundness of the translation: The correctness of the Boogie program (i.e. the correctness of all contained Boogie procedures) implies the correctness of the Viper program (i.e. the correctness of all contained Viper methods).

We decompose the proof of the final theorem into smaller parts. At a high level, the Viper-to-Boogie translation works as follows. Let F and M be the set of Viper fields and methods in the Viper program, respectively. The Viper-to-Boogie translation (1) emits global Boogie declarations G (see Sec. 4.4) and (2) generates a separate Boogie procedure $p(m)$ for every Viper method m

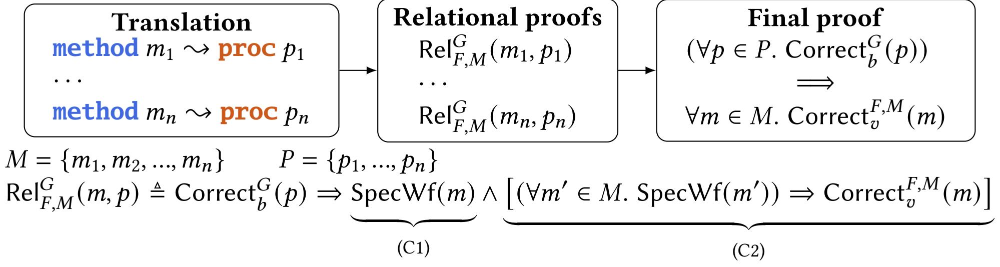

Fig. 9. Two-step proof strategy for the Viper-to-Boogie translation. First, a proof is generated relating each Viper method with the corresponding Boogie procedure. Second, the final proof is deduced. F denotes the Viper fields, G denotes the global constants, variables, Boogie axioms, and functions emitted by the translation.

in M. The intended relation between m and $p(m)$ is given by $\mathsf{Rel}^G_{F,M}(m, p(m))$ in Fig. 9, which states that the correctness of $p(m)$ w.r.t. G guarantees two things: (C1) the well-formedness of m's specification, and (C2) the correctness of m w.r.t. F and M *if* the specifications of all methods in the Viper program well-formed. The reason that the correctness of m is not implied *directly* is due to the optimised translation of method calls (as explained in Sec. 4.2).

Fig. 9 shows how we generate the proof of the desired theorem in two steps. First, for each Viper method m and its translated Boogie procedure $p(m)$, we generate a proof for $\mathsf{Rel}^G_{F,M}(m, p(m))$, explained next. Second, we obtain the desired theorem directly from these per-method relational proofs, since the correctness of all Boogie procedures implies that all Viper method specifications are well-formed using (C1), which implies that each Viper method is correct using (C2).

Next, we turn the focus to our strategy for proving $\mathsf{Rel}^G_{F,M}(m, p(m))$. For the sake of presentation, we focus on the proof of (C2) (correctness of m), and omit the proof of (C1) (well-formedness of m's specification). Intuitively, to prove that m is correct, we have to show that for any state that satisfies m's precondition, executing m's body in σ_v results in a state that satisfies m's postcondition. The correctness definition for a Viper method[7] (shown at the bottom of Fig. 8) expresses this by requiring that any execution starting in a state with no permissions that inhales the precondition, then executes the body, and finally exhales the postcondition, cannot fail. As planned, we obtain this result by contradiction, via a forward simulation proof between the executed Viper statement and $p(m)$'s procedure body using our presented methodology. Formally, we show:

$$\exists R', \gamma'.\ \mathsf{stmSim}_{\Gamma^0_v, \Gamma^0_b}(R_0, R', s^0_v, \mathsf{init}_b(p(m)), \gamma')$$
$$\textit{where } s^0_v \triangleq \textbf{inhale}\ \mathsf{pre}(m); \mathsf{body}(m); \textbf{exhale}\ \mathsf{post}(m)$$

In the statement above, $\Gamma^0_v \triangleq \mathsf{initCtxt}^{M,F}_v(m)$ is the initial Viper context. Γ^0_b is a Boogie context that is defined in terms of our chosen type and function interpretation (see Sec. 4.4). R_0 is an instantiation of the state relation shown in Sec. 4.1. $\mathsf{init}_b(p(m))$ is the initial Boogie program point in $p(m)$. The output state relation and output Boogie program point are irrelevant, since we care only about the simulation of failing Viper executions here. To complete the proof, we choose an initial Boogie state σ_b such that $R_0(\sigma_v, \sigma_b)$. As a result, using the failing Viper execution E_v of statement s^0_v in σ_v, the forward simulation provides us with a failing Boogie execution E_b of $p(m)$. Using the correctness of $p(m)$, we conclude that E_b cannot fail, and thus obtain a contradiction, which concludes the proof of $\mathsf{Rel}^G_{F,M}(m, p(m))$.

[7]We ignore typing related aspects here, but they are included in the formalisation.

834
835
836
837
838
839
840
841
842
843
844
845
846
847
848

Test suite	Files	Methods	Viper	Boogie	Isabelle	Proof Check	
	Nr.	Nr.	Mean LoC	Mean LoC	Mean LoC	Mean sec	Median sec
Viper	34	105	33	297	1759	42.3	27.2
Gobra	17	65	60	287	1976	38.8	30.1
VerCors	14	96	59	302	3090	50.5	49.4
MPP	3	13	206	1060	5220	122.7	54.3
Total	**68**	**279**	**53**	**329**	**2240**	**46.6**	**32.6**

Table 1. Overview of benchmarks and results. For each test suite, we report the number of Viper files, the total number of Viper methods contained in those files, as well as the *mean* number of non-empty lines of code for the Viper files, Boogie files, and produced Isabelle proofs. We measured the mean and median time it took to check the Isabelle proofs in seconds.

5 IMPLEMENTATION AND EVALUATION

849 We instrumented the existing Viper verifier implementation to automatically produce an Isabelle
850 proof justifying the soundness of its translation to Boogie, and evaluated this validation on a diverse
851 set of Viper benchmarks.
852
853 *Implementation.* Even though Viper passes the generated Boogie program to Boogie as a text file,
854 our soundness proof directly connects the input Viper AST to the internal AST representation of
855 the Boogie verifier. Therefore, we do not have to trust the Boogie parser.
856 We make the following four adjustments to the Viper verifier implementation. First, we desugar
857 the uses of polymorphic maps as described in Sec. 4.4, since there is no formal model for polymorphic
858 maps. Second, we adjust the implementation to not emit Boogie declarations or commands that
859 are used only for features outside of our subset (the implementation always emits those without
860 checking whether the corresponding features are actually used). Third, we switch off simple
861 syntactic transformations that the Viper verifier applies to the produced Boogie program (e.g.
862 constant folding, elimination of if-statements with no branches), since we do not support them
863 yet; justifying those transformations should be straightforward and is orthogonal to our work.
864 Fourth, we introduce a **havoc** statement in the Boogie program at the point when a scoped Viper
865 variable is introduced, which faithfully models the semantics of such a variable. The original Viper
866 implementation instead just introduces a fresh Boogie variable at the beginning of the Boogie
867 procedure. Proving the equivalence of both translations is straightforward.
868 *Benchmark Selection.* To evaluate our implementation on representative examples, we considered
869 the Viper test suite as well as the test suites of three tools that produce Viper code: Gobra [Wolf et al.
870 2021] (for Go), VerCors [Blom et al. 2017] (for Java), and MPP [Eilers et al. 2018] (a tool performing
871 a modular product transformation on Viper programs).
872 To eliminate trivial translations, we focused on Viper programs that use the heap, as indicated by
873 the occurrence of at least one accessibility predicate. Out of those, we included all Viper programs
874 that fall into our supported Viper subset. We followed different strategies to systematically obtain
875 additional examples from the different test suites. For Viper and MPP, we additionally included all
876 files that have an *old*-expression (by manually removing the corresponding assertion, i.e. verifying
877 weaker postconditions) or a *new* statement (by manually desugaring the allocation primitive into
878 our subset). For Gobra and VerCors, we removed boilerplate code that is emitted for each file and
879 then followed the same process as for Viper and MPP. Moreover, we additionally included files
880 generated by Gobra that had at most two occurrences of features outside of our subset if those
881 could be desugared into our subset (e.g. by eliminating a function by inlining its body).
882

Test suite	File	Methods Nr.	Viper Total LoC	Boogie Total LoC	Isabelle Total LoC	Proof Check Total sec
Viper	testHistoryProcesses.vpr	13	204	1709	7085	156.3
Gobra	defer-simple-02	9	211	853	4755	69.4
VerCors	SwapIntegerPass	8	81	469	3732	63.7
MPP	banerjee.vpr	8	414	2014	9601	266.8
MPP	darvas.vpr	2	91	582	2856	46.9
MPP	kusters.vpr	3	112	583	3202	54.3

Table 2. Detailed results of our evaluation for a selection of files showing the number of methods, the nonempty lines of code for the Viper program, Boogie program, and produced Isabelle proof, and the time it took to check the proof in seconds.

As summarised in Tab. 1, we collected a total of 68 Viper files (containing 279 methods), with a mean of 53 non-empty lines of code. The generated Boogie translations are on average 6.2x larger (329 non-empty LoC on average), illustrating the semantic gap between Viper and Boogie.

Results. Our implementation successfully generated Isabelle proofs for all of the Viper files, including the Viper programs automatically generated by other tools. This shows that our approach is effective for practical verifiers. The resulting Isabelle proofs have on average over 2000 lines and are checked in less than a minute. The measurements were run on a Lenovo T480 with 32GB, i7-8550U 1.8GhZ, Ubuntu 18.04 on the Windows Subsystem for Linux.

Tab. 2 shows the results for a selection of examples (the detailed results for each test suite are shown in App. C): All three examples from MPP, as well as the largest (in terms of lines of Viper code) example from each of the other test suites. The three MPP examples are drawn from different research papers and show that our tool can certify challenging programs.

For this selection, the times to check the proofs range from 47 seconds to 4.4 minutes, which is acceptable since we expect the validation to be performed occasionally (in particular, before the verified program is released), but not on every run of the verifier. Moreover, most of our proof strategies are not yet optimised to make proof checking faster. For example, field and variable accesses currently result in an overhead in the proof that is proportional to the number of fields and active variables, respectively. This could be improved by constructing and updating lookup tables efficiently.

6 RELATED WORK

Various works prove the soundness of front-end translations *once and for all*. For instance, Lehner and Müller [2007] prove a simplified translation from Java Bytecode to Boogie, and Vogels et al. [2009] target a translation from a toy object-oriented programming language to Boogie. Both proofs are done on paper and do not consider an actual implementation of the translation. Backes et al. [2011] prove a translation sound from the Dminor data processing language to the Bemol IVL in Coq. They do not provide a proof connecting the formalised translation to their F# implementation. Herms [2013] proves a translation from C to the WhyCert IVL (inspired by the Why3 IVL) sound in Coq, which they then turn into an executable tool via Coq's extraction to OCaml. The resulting tool has similarities to the Jessie Frama-C implementation [Marché and Moy 2018], which translates C programs to Why3; Herms [2013] discusses mismatches between their mechanisation and the Jessie implementation. In contrast, our certification applies to existing front-end implementations, which are typically implemented in efficient mainstream programming languages, use diverse libraries, and include subtle optimisations omitted from idealised implementations. Smans et al. [2012] prove

932 soundness of a verification condition generator for a language with implicit dynamic frames (IDF)
933 assertions once and for all on paper. They also implement a prototype, but do not formally connect
934 the proof to the implementation. We also applied our methodology to a verifier based on IDF, but
935 validate an actual implementation.

936 Validation has been used to obtain formal guarantees for implementations of verifiers, but none
937 of the existing works target front-end translations and the challenges they entail. Parthasarathy
938 et al. [2021] validate the verification condition generation of Boogie programs, including various
939 Boogie-to-Boogie transformations. Consequently, they neither face the semantic gap we handle,
940 nor did they have to support diverse translations and non-local checks. Their work can in principle
941 be combined with ours to enable end-to-end soundness guarantees for Viper, but first requires
942 extending their validation to all the Boogie-to-Boogie transformations applied by the Boogie verifier.
943 Lin et al. [2023] and [Wils and Jacobs 2023] validate verifiers obtained via the K framework and
944 VeriFast, respectively. These verifiers use symbolic execution, which requires a fundamentally
945 different validation approach. Garchery [2021] validate certain logical transformations in Why3,
946 but not the actual verification condition generation.

947 There are multiple works that also embed programs in an ITP and then automate forward
948 simulation proofs involving those programs. Rizkallah et al. [2016] define a refinement calculus
949 for the Cogent compiler to automatically produce a forward simulation proof in Isabelle for a
950 Cogent expression and its C translation. Their calculus includes syntax-directed rules for deriving
951 a concrete forward simulation judgement, but these rules do not provide the abstraction we needed
952 to handle diverse translations. The Cogent compiler was developed with formal validation in
953 mind, which simplifies, for instance, the treatment of optimisations. In contrast, our goal was to
954 validate existing verifier implementations with all their intricacies. The verification of the seL4
955 kernel includes two large forward simulation proofs (involving the C kernel implementation), for
956 which automation techniques were developed [Cock et al. 2008; Klein et al. 2010; Winwood et al.
957 2009]. This automation reduces the manual proof overhead, but still require user interaction. In
958 contrast, our validation proofs are generated and checked completely automatically. They prove
959 rules to decompose the forward simulation for composite statements but contrary to us, they do not
960 decompose non-composite statements further via rules. Instead, they develop a symbolic execution
961 technique to deal with forward simulation judgements by turning them into Hoare triples.

962 *Formal translation validation* approaches for compilers express a per-run validator in an ITP [Gour-
963 din et al. 2023; Tristan and Leroy 2008, 2009], prove it correct once and for all, and then extract
964 executable code (the extraction must be trusted). For many of these validators, the source and target
965 languages are similar. It would be interesting to test the feasibility of such approaches for front-end
966 translations, where the semantic gap between between the languages is large.

967 Zimmerman et al. [2023] define a formal Viper semantics for a Viper subset in order to prove
968 formal results for the gradual verifier Gradual C0 that uses Viper. However, in contrast to ours,
969 their formalisation is not mechanised.
970
971
7 CONCLUSION

972 We presented a methodology for the validation of the front-end translations implemented in practical
973 automated program verifiers. We demonstrated that it handles the complexity and intricacies of
974 the Viper-to-Boogie translation as implemented in the Viper tool. To the best of our knowledge,
975 this is the first formal soundness guarantee for a practical front-end translation. Together with
976 existing work on back-end (and SMT) validation, our work provides a path towards trustworthy
977 automated verifiers. As future work, we plan to extend the supported Viper subset and to apply
978 our methodology to verifiers that target Viper as an IVL and that verify, for instance, concurrent or
979 object-oriented programs.
980

981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029

REFERENCES

Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science)*, Klaus Schneider and Jens Brandt (Eds.), Vol. 4732. Springer, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147, 30 pages. https://doi.org/10.1145/3360573

Michael Backes, Cătălin HriŢcu, and Thorsten Tarrach. 2011. Automatically Verifying Typing Constraints for a Data Processing Language. In *Certified Programs and Proofs (CPP)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). https://doi.org/10.1007/978-3-642-25379-9_22

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods (IFM)*, Nadia Polikarpova and Steve Schneider (Eds.). https://doi.org/10.1007/978-3-319-66845-1_7

Sascha Böhme and Tjark Weber. 2010. Fast LCF-Style Proof Reconstruction for Z3. In *Interactive Theorem Proving (ITP)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). https://doi.org/10.1007/978-3-642-14052-5_14

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS)*, Radhia Cousot (Ed.). 55–72. https://doi.org/10.1007/3-540-44898-5_4

Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamaric, and Michael Emmi. 2016. SMACK software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 589–592. https://doi.org/10.1145/2889160.2889163

David A. Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Theorem Proving in Higher Order Logics (TPHOLS)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). https://doi.org/10.1007/978-3-540-71067-7_16

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *International Conference on Formal Engineering Methods (ICFEM)*, Adrián Riesco and Min Zhang (Eds.), Vol. 13478. 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. arXiv:cs.LO/2210.02428

Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification (CAV)*, Hana Chockler and Georg Weissenbacher (Eds.). https://doi.org/10.1007/978-3-319-96145-3_33

Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *European Symposium on Programming (ESOP)*, Amal Ahmed (Ed.). https://doi.org/10.1007/978-3-319-89884-1_18

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification (CAV)*, Rupak Majumdar and Viktor Kuncak (Eds.). https://doi.org/10.1007/978-3-319-63390-9_7

J.-C. Filliâtre and A. Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symposium on Programming (ESOP)*, Matthias Felleisen and Philippa Gardner (Eds.). https://doi.org/10.1007/978-3-642-37036-6_8

Mathias Fleury and Hans-Jörg Schurr. 2019. Reconstructing veriT Proofs in Isabelle/HOL. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, Giselle Reis and Haniel Barbosa (Eds.). https://doi.org/10.4204/EPTCS.301.6

Quentin Garchery. 2021. A Framework for Proof-carrying Logical Transformations. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, Chantal Keller and Mathias Fleury (Eds.). https://doi.org/10.4204/EPTCS.336.2

Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 224 (oct 2023), 30 pages. https://doi.org/10.1145/3622799

Paolo Herms. 2013. *Certification of a Tool Chain for Deductive Program Verification. (Certification d'une chaine de vérification déductive de programmes).* Ph.D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00789543

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Formal Methods (FM)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). https://doi.org/10.1007/11813040_19

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer, 323–339. https://doi.org/10.1007/978-1-4419-1539-9_11

1030 Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

Akash Lal and Shaz Qadeer. 2014. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 202–212. https://doi.org/10.1145/2635868.2635894

Hermann Lehner and Peter Müller. 2007. Formal Translation of Bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science* 190, 1 (2007), 35–50. https://doi.org/10.1016/j.entcs.2007.02.059 Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007).

K. Rustan M. Leino. 2008. This is Boogie 2. (2008). Available from http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Edmund M. Clarke and Andrei Voronkov (Eds.). https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino and Philipp Rümmer. 2010. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Javier Esparza and Rupak Majumdar (Eds.). https://doi.org/10.1007/978-3-642-12002-2_26

Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Rosu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 56–84. https://doi.org/10.1145/3586029

Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

Claude Marché and Yannick Moy. 2018. The Jessie plugin for Deductive Verification in Frama-C. http://krakatoa.lri.fr/jessie.pdf

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). https://doi.org/10.1007/978-3-662-49122-5_2

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3:01 (2012), 1–54. https://doi.org/10.2168/LMCS-8(3:1)2012

G. Parthasarathy, P. Müller, and A. J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification (CAV) (LNCS)*, Alexandra Silva and K. Rustan M. Leino (Eds.), Vol. 12760. 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby C. Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C.. In *Interactive Theorem Proving (ITP)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). https://doi.org/10.1007/978-3-319-43144-4_20

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *Transactions on Programming Languages and Systems (TOPLAS)* 34, 1, Article 2 (May 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of Programming Languages (POPL)*, George C. Necula and Philip Wadler (Eds.). https://doi.org/10.1145/1328438.1328444

Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified validation of lazy code motion. In *Programming Language Design and Implementation (PLDI)*, Michael Hind and Amer Diwan (Eds.). https://doi.org/10.1145/1542476.1542512

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2009. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *Theory and Practice of Computer Science, Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) (Lecture Notes in Computer Science)*, Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia (Eds.), Vol. 5404. Springer, 570–581. https://doi.org/10.1007/978-3-540-95891-8_51

Stefan Wils and Bart Jacobs. 2023. Certifying C program correctness with respect to CH2O with VeriFast. *CoRR* abs/2308.15567 (2023). https://doi.org/10.48550/ARXIV.2308.15567 arXiv:2308.15567

Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David A. Cock, and Michael Norrish. 2009. Mind the Gap. In *Theorem Proving in Higher Order Logics (TPHOLS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). https://doi.org/10.1007/978-3-642-03359-9_34
1078

1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127

Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification (CAV)*, Alexandra Silva and K. Rustan M. Leino (Eds.). https://doi.org/10.1007/978-3-030-81685-8_17

Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2023. Sound Gradual Verification with Symbolic Execution. *CoRR* abs/2311.07559 (2023). https://doi.org/10.48550/ARXIV.2311.07559 arXiv:2311.07559

$$\frac{\langle A, \sigma_v\rangle \rightarrow_{\mathsf{inh}} r_v}{\Gamma_v \vdash \langle \textbf{inhale}\ A, \sigma_v\rangle \rightarrow_{\mathsf{v}} r_v}\ (\textsc{inh}) \qquad \frac{\begin{array}{c}\langle e, \sigma_v\rangle \Downarrow \mathsf{V}(r) \quad \langle e_p, \sigma_v\rangle \Downarrow \mathsf{V}(p) \\ p < 0 \Rightarrow r_v = \mathsf{F} \\ p \geq 0 \Rightarrow r_v = \mathit{if}\ \mathit{inhSucc}(r,p)\ \mathit{then}\ \mathsf{N}(\sigma_v')\ \mathit{else}\ \mathsf{M} \\ \sigma_v' = \mathsf{addperm}(\sigma_v, r, f, p)\end{array}}{\langle \textbf{acc}(e.f, e_p), \sigma_v\rangle \rightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-acc})$$

$$\frac{\begin{array}{c}\langle A, \sigma_v\rangle \rightarrow_{\mathsf{inh}} \mathsf{N}(\sigma_v') \\ \langle B, \sigma_v'\rangle \rightarrow_{\mathsf{inh}} r_v\end{array}}{\langle A * B, \sigma_v\rangle \rightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-sep-s}) \qquad \frac{\langle A, \sigma_v\rangle \rightarrow_{\mathsf{inh}} \mathsf{F}}{\langle A * B, \sigma_v\rangle \rightarrow_{\mathsf{inh}} \mathsf{F}}\ (\textsc{inh-sep-f})$$

$$\mathit{inhSucc}(r,p) \triangleq (p > 0 \Rightarrow r \neq \textbf{null}) \wedge (r \neq \textbf{null} \Rightarrow p + \pi(\sigma_v)(r,f) \leq 1)$$

Fig. 10. A subset of the rules for the formal semantics of inhale. $\mathsf{addperm}(\sigma_v, r, f, p)$ denotes the state σ_v where permission p has been added to (r, f).

$$\frac{\begin{array}{c}\forall \sigma_v.\ \mathsf{wfSim}_{\Gamma_b}(\hat{R}(\sigma_v), \hat{R_A}(\sigma_v), [e, e_p], \gamma, \gamma_1) \quad \text{(subexpression well-definedness)} \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R_A, R_B(r,p), \mathit{Succ}_A(r,p), \mathit{Fail}_A(r,p), \gamma_1, \gamma_2) \quad \text{(non-failure check)} \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R_B(r,p), R', \mathit{Succ}_B(r,p), (\lambda_.\ \bot), \gamma_2, \gamma') \quad \text{(state update)}\end{array}}{\mathsf{rcSim}_{\Gamma_b}(R, R', \textbf{acc}(e.f, e_p), \gamma, \gamma')}\ (\textsc{racc-sim})$$

$$\hat{R}(\sigma_v) \triangleq \lambda \sigma_v^0\ \sigma_b.\ R((\sigma_v^0, \sigma_v), \sigma_b) \qquad \hat{R_A}(\sigma_v) \triangleq \lambda \sigma_v^0\ \sigma_b.\ R_A((\sigma_v^0, \sigma_v), \sigma_b)$$

$$\mathit{Succ}_A(r,p) \triangleq \left(\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \begin{array}{l}\mathsf{exhAccSucc}(r, p, \sigma_v) \wedge (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \\ \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0)\end{array}\right)$$

$$\mathit{Fail}_A(r,p) \triangleq \lambda(\sigma_v^0, \sigma_v).\ \neg\mathsf{exhAccSucc}(r, p, \sigma_v) \wedge \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0)$$

$$\mathit{Succ}_B(r,p) \triangleq \left(\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \begin{array}{l}\sigma_v' = \mathsf{rem}(\sigma_v, r, f, p) \wedge \sigma_v^0 = \sigma_v^1 \wedge \\ \mathsf{exhAccSucc}(r, p, \sigma_v) \wedge \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0)\end{array}\right)$$

$$\mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0) \triangleq \langle e, \sigma_v^0\rangle \Downarrow \mathsf{V}(r) \wedge \langle e_p, \sigma_v^0\rangle \Downarrow \mathsf{V}(p)$$

Fig. 11. Rule for the simulation of **remcheck acc**$(e.f, e_p)$. The definition of exhAccSucc is given in Fig. 2.

A INHALE SEMANTICS

The reduction of **inhale** A for an assertion A from σ_v to σ_v' is expressed via the judgement $\langle A, \sigma_v\rangle \rightarrow_{\mathsf{inh}} \sigma_v'$. The rules for the separating conjunction and the accessibility predicate (when the receiver and permission are well-defined) are shown in 10. In the case of the accessibility predicate, there is an additional rule where the **inhale** fails if e or e_p are not well-defined.

The accessibility rule shown in 10 expresses that if the added the permission is negative then the operation fails. If the permission is nonnegative, then the operation succeeds if (1) the receiver is non-null if $p > 0$, (2) the added permission does not yield an inconsistent state (i.e. does not result in more than 1 permission for (r, f)). Otherwise, the operation stops (denoted by outcome M). If the operation succeeds, then the new state additionally contains the added permission p at location (r, f).

1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195

File	#M	Viper	Boogie	Isabelle.	Mean [s]
examples/tutorial-examples/concurrency.gobra.vpr	2	24	164	1191	26.6
features/defer/defer-simple-01.gobra.vpr	6	142	639	3382	55.6
features/defer/defer-simple-02.gobra.vpr	9	211	853	4755	69.4
features/fractional_permissions/perm-fail1.gobra.vpr	15	165	661	6430	82.3
features/fractional_permissions/perm-simple1.gobra.vpr	9	131	622	4259	64.1
...s/fractional_permissions/predicates/fail1.gobra.vpr	3	44	283	1612	37.7
...s/fractional_permissions/predicates/fail3.gobra.vpr	2	19	116	1082	26.5
...fractional_permissions/predicates/simple1.gobra.vpr	2	30	237	1248	30.6
...fractional_permissions/predicates/simple2.gobra.vpr	1	10	90	710	23.0
...fractional_permissions/predicates/simple3.gobra.vpr	1	17	186	839	27.9
features/global_consts/global-const-8.gobra.vpr	6	49	206	2548	42.1
features/no_semicolons/pointer-identity.gobra.vpr	1	30	158	775	28.9
features/pointer-identity.gobra.vpr	1	30	158	775	28.6
issues/000008.gobra.vpr	1	10	85	710	28.2
issues/000009.gobra.vpr	1	16	98	717	24.9
issues/000039.gobra.vpr	3	49	178	1448	30.1
issues/000155.gobra.vpr	2	39	152	1113	32.3

Table 3. Detailed results of our evaluation for the files from the test suite of Gobra.

1196
1197
1198

B ANOTHER SIMULATION RULE EXAMPLE

1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221

Consider the rule RACC-SIM in Fig. 11 decomposes the simulation of **remcheck acc**$(e.f, e_p)$ (ignore the universal quantifiers for now) into the simulation of three separate Viper effects: (1) the check of well-definedness of the receiver e and permission expression e_p (via wfSim instantiation from Fig. 4), (2) a check exhAccSucc ensuring that the operation will not fail (from the semantics; see Fig. 2), and (3) the actual update of the Viper state, which removes the permission.

The second premise includes contextual information, namely the conjunct wfAccSucc expressing that e and e_p are well-defined (which is ensured by the first premise) and evaluate to the reference value r and permission value p. The third premise modelling the removal of the permission includes the same conjunct and that the operation will succeed (exhAccSucc). Without the latter, we could in general not prove that the resulting Boogie state satisfies crucial invariants, for instance, that none of the permissions stored in the Boogie state are negative. Again, we are agnostic as to syntactically *how* this is achieved by this check: our rule does *not* require the Boogie program to emit an explicit Boogie **assert** command checking that the permission is nonnegative. This is important, since the implementation omits such a command, for example, if the permission is the literal 1.

The first universal quantifier is technically motivated: it expresses that the simulation must hold for *any* reduction state. The other quantifiers over reference values r and permission values p make the rule more powerful and reusable. They permit the relation R_2 to directly talk about the values that e and e_p evaluate to as specified by the success and failure predicates. This is particularly useful for justifying cases where the simulation of the non-failure check establishes a property on r or p, which is then used in the simulation of the state update. For example, the Viper-to-Boogie translation stores p into an auxiliary variable that is used for both the non-failure check and the state update.

1222

C DETAILED RESULTS OF THE EVALUATION

1223
1224
1225

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

File	#M	Viper	Boogie	Isabelle.	Mean [s]
/concepts/basic/BasicAssert-e1.java.vpr-0.vpr	6	41	197	2627	32.9
/concepts/basic/BasicAssert.java.vpr-0.vpr	6	41	193	2627	34.6
/concepts/basic/DafnyIncr.java.vpr-0.vpr	8	60	265	3457	48.3
/concepts/basic/DafnyIncrE1.java.vpr-0.vpr	8	57	220	3378	47.6
/concepts/permissions/frame_error_1.pvl.vpr-0.vpr	5	35	173	2229	33.5
/concepts/permissions/SwapIntegerFail.java.vpr-0.vpr	8	79	429	3689	58.1
/concepts/permissions/SwapIntegerPass.java.vpr-0.vpr	8	81	469	3732	63.7
/concepts/permissions/SwapLong.java.vpr-0.vpr	6	57	277	2769	41.9
/concepts/permissions/SwapLongTwice.java.vpr-0.vpr	8	81	469	3732	64.0
/concepts/permissions/SwapLongWrong.java.vpr-0.vpr	8	79	429	3689	64.2
/concepts/refute/refute3.java.vpr-0.vpr	6	49	246	2700	59.3
/concepts/refute/refute4.java.vpr-0.vpr	6	54	258	2714	49.4
/concepts/refute/refute5.java.vpr-0.vpr	6	50	253	2700	49.3
/demo/demo1.pvl.vpr-0.vpr	7	60	347	3223	60.4

Table 4. Detailed results of our evaluation for the files from the test suite of VerCors.

1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323

File	#M	Viper	Boogie	Isabelle.	Mean [s]
0004.vpr	1	6	100	767	24.3
0004_CPG1.vpr	1	6	95	742	23.7
0005.vpr	1	4	78	703	23.0
0008.vpr	2	12	241	1434	31.8
0011.vpr	5	64	902	3340	64.4
0015.vpr	1	6	92	753	26.6
0052.vpr	1	7	100	757	23.5
0063.vpr	6	36	180	2633	42.3
0072.vpr	1	8	112	814	32.1
0073.vpr	1	10	132	781	24.6
0088-1.vpr	1	9	115	789	23.9
0094.vpr	1	7	91	717	23.0
0152.vpr	2	14	139	1175	27.3
0157.vpr	8	48	354	3564	52.2
0159.vpr	2	13	120	1121	27.0
0170.vpr	1	8	84	703	22.6
0177-1.vpr	1	10	102	703	23.4
0222.vpr	2	13	118	1092	26.1
0227.vpr	1	5	85	721	23.2
0324.vpr	1	7	104	742	23.9
0345.vpr	3	21	165	1501	30.3
0384.vpr	1	11	127	747	23.8
assert.vpr	1	7	92	731	23.6
negative_amounts.vpr	3	21	155	1561	31.4
old.vpr	6	38	318	2843	43.1
swap.vpr	2	16	177	1283	28.8
test.vpr	1	6	81	701	22.9
testHistoryProcesses.vpr	13	204	1709	7085	156.3
testHistoryProcessesPVL.vpr	13	204	1711	7085	176.5
testHistoryProcessesPVL_CPG1.vpr	4	56	490	2342	62.9
testHistoryThreadsProcessesPVL.vpr	4	56	490	2342	52.1
test_example1.vpr	4	57	374	2190	41.7
test_example3.vpr	5	74	430	2672	86.4
test_example4.vpr	5	71	451	2683	68.9

Table 5. Detailed results of our evaluation for the files from the test suite of Viper.

File	#M	Viper	Boogie	Isabelle.	Mean [s]
banerjee.vpr	8	414	2014	9601	266.8
darvas.vpr	2	91	582	2856	46.9
kusters.vpr	3	112	583	3202	54.3

Table 6. Detailed results of our evaluation for the files from the test suite of MPP.