Formal development demonstrator prototype, final release

Project acronym:	4SECURail
Starting date:	01/12/2019
Duration (in months):	24
Call (part) identifier:	H2020-S2R-OC-IP2-2019-01
Grant agreement no:	881775
Due date of TIN:	Month 20 (31 July 2021)
Actual submission date:	31-07-2021
Responsible/Author:	CNR / Franco Mazzanti
Dissemination level:	PU
Status:	Draft/Issued

Reviewed: yes

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 881775.

Document history		
Revision	*Date*	*Description*
0.1	25-06-2021	first structure of content
0.2	20-07-2021	version for internal review
0.3	27-07-2021	version with consolidated review comments
1.0	31-07-2021	final version

Report contributors		
Name	Beneficiary Short Name	Details of contribution
Franco Mazzanti	CNR	Overall Structure and Content
Dimitri Belli	CNR	Overall Structure and Content
Alessio Ferrari	CNR	Comments and suggestions
Alessandro Fantechi	CNR	Comments and suggestions
Davide Basile	CNR	Comments and suggestions
Stefania Gnesi	CNR	Comments and suggestions
Laura Masullo	SIRTI/MERMEC-STE[1]	Comments and suggestions
Daniele Trentini	SIRTI/MERMEC-STE	Comments and suggestions
Lisa Quadrini	SIRTI/MERMEC-STE	Comments and suggestions
Lambert Grange	ARDANUY	Final Internal Review
Carlo Vaghi	FIT	Final Internal Review

Disclaimer

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author's view – the Joint Undertaking is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability. The content of this deliverable does not reflect the official opinion of the Shift2Rail Joint Undertaking (S2R JU). Responsibility for the information and views expressed in the deliverable lies entirely with the author(s).

[1] Sirti S.p.A. was awarded the Open Call and signed the Grant Agreement for 4SECURAIL In December 2020, the BU Transportation of Sirti S.p.A. became a new company called Sirti Transportation srl, 100% owned by Sirti SpA. On February 9th 2021 Sirti Transportation srl was totally acquired by MER MEC SpA, changing the name into MER MEC STE srl. At the time this deliverable is written, the procedure for the replacement of Sirti with MER MEC STE in this project is still ongoing. Therefore, the name SIRTI appears in this deliverable when documents, facts, events from up to February 2021 are reported. The name MER MEC STE appears when referring to documents, facts, events from February 2021 on.

Table of Contents

1 Executive Summary 1
2 Abbreviations and acronyms 3
3 Background 4
4 Objective/Aim 5
5 The Exercising of the Formal Development Demonstrator 6
5.1 An Overview of the Signalling Case Study 7
5.2 From Natural Language to Semi-Formal, to Formal 8
5.3 From Formal, to Semi-Formal, to Natural Language 16
5.4 Verification Architectures and Scenarios 19
5.5 Advanced Formal Analysis 22
6 Conclusions 29
7 References.... 33
Appendix A: Formal notations and transformations 35
Appendix B: Graphical semiformal UML state machine designs.... 40
Appendix C: Structured Natural Language Requirement Specifications.... 42
Appendix D: Difformities with respect to the D2.3 Requirements 60
Appendix E: Model reduction techniques 70
Appendix F: Analysing the behavior at the interfaces.... 73

1 Executive Summary

This Deliverable describes the final results of Task 2.3 of 4SECURail project. The goal of Task 2.3 is to apply the formal development demonstrator process defined in Task 2.1 to the signalling case study defined in Task 2.2 and to describe the *observed* impact of the selected tools and methodologies for improving the *quality of the system specifications* under analysis.

The activity performed in Task 2.3 focusses on three main issues:

1. A revision of the modelling and analysis *process* adopted for the initial fragment of the case study. In particular, the main revisions involve:
    a. The choice to *complement* the initially selected formal method with a second one.
    b. The choice to *mechanically* generate formal models from their semi-formal description.
    c. The definition of a *structured logical framework* within which to experiment with formal analysis based on the UMC, ProB, and CADP frameworks.
2. The extension of selected portion of *the case study* already considered in Task 2.1, to progress towards its complete modelling and analysis.
3. The *experimentation* of formal verification approaches based upon the definition of selected scenarios for the stimulation of the subsystems (or group of subsystems) of interest.

In this deliverable, we also describe the *observations* resulting from the demonstrator process activity. In particular, from our perspective, the most important takeaways concern:

- The presentation of an easy way in which UML/SysML artefacts can be effectively used as a complement to the specification of system requirements.
- The importance of *multiple* (formal methods diversity)*, mechanical* generation of formal models of different types.
- The observation of the practical impact of semi-formal modelling and formal analysis techniques on identifying weaknesses in the initial natural language system requirements definition.
- The observation of difficulties and limits incurring in the exploitation of formal methods for the requirement specification phase.
- The importance of a *consistent and integrated* set of rigorous natural language descriptions, UML-based semi-formal artefacts, and formal models to consolidate the overall quality of the system requirements specification.

In defining the structure of this document, we have tried to keep separate, as far as possible, the formal technical details of the points raised by the demonstrator process activity from the conceptual issues to which they are related.
After presenting in Section 5.1 an overview on the case study to which 4SECURail demonstrator

has been applied, in Section 5.2, 5.4 and 5.5 the model construction and analysis process which has been undergone is described. In Section 5.3 the kind of output which can be expected from the application of our formal analysis process is described, and in Appendix B, C, and D specific demonstrator outputs related to the selected case study are reported.
Appendix A, E, and F provide a more detailed description of the undergone process, but can be skipped if the reader is not particularly interested in more technical details.
All the generated models, developed architectures, and analysed scenarios are available from a public data repository [ZenodoWP2] containing all the WP2 generated artifacts.

2 Abbreviations and acronyms

Abbreviation / Acronyms	Description
CADP	Construction and Analysis of Distributed Processes
CFM	Communication Functional Module
CSL	Communication Supervision Layer
EA	Enterprise Architect
ECS	Execution Cycle Start
ER-SL	EuroRadio Safety Layer
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
FIFO	First-In-First-Out
IC	Innovation Capability
IM	Infrastructure Manager
LNT	LOTOS New Technology
LTS	Labelled Transition System
MAAP	Multi-Annual Action Plan
MBSD	Model-Based Software/System Development
OMG	Object Management Group
RBC	Radio Block Centre
SAI	Safe Application Intermediate sub-Layer
SFM	Safe Functional Module
SysML	Systems Modeling Language
TD	Technology Demonstrator
TTS	Triple Time Stamp
UMC	UML Model Checker
UML	Unified Modelling Language
UNISIG	Union industry of signalling
WP	Work Package

3 Background

The present document is the Deliverable 2.5 (D2.5) "*Formal development Demonstrator prototype*" of Task 2.3 of Work Package 2 (WP2) "*Demonstrator Development for the use of Formal Methods in Railway Environment*" of the project 4SECURail (GA 881775) in the context of the open call S2R-OC-IP2-01-2019, part of the "Annual Work Plan and Budget 2019", of the programme H2020-S2RJU-2019.

The challenge to which 4SECURail is deemed to deal, and its relation with the Shift2Rail Technology Demonstrator 2.7 (TD2.7) "Formal methods and standardisation for smart signalling systems" is well described in the call S2R-OC-IP2-01-2019, as shown below:

"Shift2Rail has identified the use of **formal methods** and standard interfaces as two key concepts to enable reducing the time it takes to develop and deliver railway signalling systems, and to reduce costs for procurement, development, and maintenance. Formal methods are needed to ensure correct behavior, interoperability and safety, and standard interfaces are needed to increase market competition and standardization, reducing long-term life cycle costs."

According to [MAAP2019], the Shift2Rail Innovation Programme 2 (IP2) focusses on innovative technologies, systems, and applications in the fields of telecommunication, train separation, supervision, engineering, automation, and security to enhance the overall performance of all railway market segments.
The TD2.7 aims at contributing to enable two Innovation Capabilities (ICs) of the Shift2Rail IP2,

- IC7 "Low-Cost Railway"
- IC12 "Rapid and Reliable R&D Delivery"

through the Building Block achievement BB2.7_1 "Formal and semi-formal methods for requirement capture, design, verification, and validation, proposing open standards".

4SECURail contributes to the above Building Block achievement with the demonstration and evaluation of techniques based on formal methods to reduce life-cycle costs and to improve the global availability of railway systems.

For our purposes, the project scenario considers the Infrastructure Managers (IMs) applying formal and semi-formal methods to build robust and verifiable system requirements specifications, which makes the procurement of systems and equipment - compliant with legal requirements and needs of operators - possible and suitable for their easy integration in the existing railway subsystems. Such an effort contributes to the progress towards an open market for maintenance (availability of spare parts) and future enhancements (implementation of new functions and/or performance, exploiting open and standardized interfaces).
The idea of IMs is to have modular systems and to define standardized interfaces to integrate these modules. In this context of modular systems, the use of formal methods is a solid support for the definition of more effective, efficient, and satisfactory standard interfaces.

4 Objective/Aim

One of the objectives of the 4SECURail project is to perform a cost-benefit analysis for the adoption of formal methods in the railway environment by prototyping a formal method Demonstrator to be exercised with a selected case study. The use of formal methods in the railway context covers many distinct aspects, from the definition of verifiable requirements to the construction of a more affordable and efficient development process. A recent detailed study on the subject is presented in [FMRMAP].

The objective of Task 2.3 is to exercise a system requirements analysis process that exploits the use of semi-formal and formal methods to improve the quality of the specifications written by the railway IMs. The definition, rationale, and overall structure of this process have been described in detail in Deliverable 2.1 (D2.1) and Deliverable 2.2 (D2.2). The purpose of this deliverable is to describe the experience gained in the application of the defined formal methods demonstrator process to the signalling system case study explained in D2.3, by putting in evidence the advantages gained in terms of better understanding and possibly better specification and presentation of the system requirements.

This activity is aligned with the objective of TD2.7 [MAAP2019] *Formal Methods and standardisation for smart signalling,* which focusses on applying Formal Methods and Standard Interfaces in application Demonstrators and the business case study for using them.

5 The Exercising of the Formal Development Demonstrator

The goal of our formal methods demonstrator is to illustrate a *possible* impact of the introduction of formal methods inside the *system requirements definition process of the IMs.*
This is done by observing, in our specific case, the effects of applying the *specific tools and methodologies* used by the demonstrator process to our *specific case study*. We take the point of view of an IM that intends to define the system requirements specification document to be used in tenders or in standard interface definitions. Formal methods are exploited in this requirements definition process for *improving the confidence* that:

- the requirements document *clearly* and *unambiguously* reflects the intentions of the designers (aka the IMs);
- the implementations eventually deriving from the requirements document will correctly *interoperate* with other environment components with which the system is expected to interact.

We are, in fact, talking of two very different kinds of goals. The first one is related to the *precision* (i.e., the clarity, the completeness, and the consistency) of a determined subsystem specification, targeted to become an attractive tender for the providers.
The second one is related to the improvement of the confidence that what specified is *precisely what is needed*, i.e., it is something that really corresponds to the ideas of the designer and to the expectation of a proper interoperability of the system with the other components of the railway framework, which is essentially a system of systems.

It is worth pointing out that our experimentation does not cover the possible adoption of formal methods during the system development phase carried out by the system providers. However, it is clear that a potential application of formal methods in the development phase is useful only in presence of a clear, rigorous, complete, and consistent system requirements specification document.

The activity described in this document is strictly related to three previous deliverables:

- D2.1 [D2.1] describes the planned structure of our formal development demonstrator process and the rationale behind it.
- D2.2 [D2.2] presents a first attempt to apply the demonstrator process to an initial fragment of the case study. Such an attempt allowed us to gain some early experience and led us to the improvement of the process itself.
- Deliverable 2.3 (D2.3) [D2.3] describes the planned case study for testing the application of the formal development demonstrator process.

Parts of the above deliverables might be repeated here to give a more self-contained view of the demonstrator process activity.

The rest of the Section is split as follows: In Subsection 5.1 we give an overview of the case study in use to exercise the demonstrator; in Subsection 5.2 we describe the building process of formal models, starting from the initial Natural Language requirements provided and the basic verification steps that can be performed; in Subsection 5.3 we describe the set of artifacts generated as a result of the demonstrator process activity, which represents the feedback towards the IMs of the formal analysis; in Subsections 5.4 and 5.5 we describe in detail the ways in which we conducted the formal analysis.

5.1 An Overview of the Signalling Case Study

In the European Rail Traffic Management System/ European Train Control System (ERTMS/ETCS), a Radio Block Centre (RBC) is responsible for managing trains under its area of supervision. A handover procedure is needed to manage the interchange of train control supervision between two neighbour RBCs. When a train is approaching the end of the area supervised by one handing over RBC, an exchange of information with the accepting RBC takes place to manage the transaction of responsibilities. Since the two neighbouring RBCs may have been manufactured by different providers, the RBC/RBC interface is a typical product where the products (RBCs) of different suppliers must be interoperable.
D2.3 [D2.3] integrates the ETCS specifications contained in SUBSET-039 – "FIS for the RBC/RBC Handover"[SUB-039] and SUBSET-098 – "RBC/RBC Safe Communication Interface [SUB-098] with additional requirements". The adapted definition of the subsystem is limited to higher application levels and safety levels (SAI sub-level of SUBSET-098). Thus, the case study isolates and identifies two layers:

- A Communication Supervision Layer (CSL): It is responsible for commanding the opening/closing of the communication line between RBCs and for keeping the connection alive through Life Signs. Its functional requirements are covered by UNISIG SUBSET-039.
- A Safety Application Intermediate sub-Layer (SAI): it is logically located below the CSL, and relies on the introduction of time-related data (e.g., execution cycle counters), message sequence numbers to implement the protection mechanisms against threats (like package deletion, replication, resequencing, and delay) as identified by CENELEC EN50159 [EN50159]. In this case, its functional requirements are covered by UNISIG SUBSET-098. Most requirements related to the safety of the communication are allocated in the SAI sub-layer.

Above the CSL, the RBC User layer includes all application functions (e.g., evaluation of Movement Authorities, communication with on-board units, actual management of RBC/RBC handover transactions) and the generation/reception of information to communicate. While protocol layers are dedicated to formatting and exchanging such information with communication partners. The specification of RBC User functions is not included in the requirements of the case study. Moreover, also the lower levels below SAI, that is, the EuroRadio Safety Layer (ER-SL) [SUB-037]

and the Communication Functional Module (CFM) of SUBSET-098, are not part of the requirements of the case study.
Thus, the Demonstrator will be applied to the CSL and SAI levels, whilst RBC User, ER-SL, and CFM are treated as components of the external environment.
Figure 1 shows the overall structure of the system. Notice that of the two communicating sides, one side is configured as *initiator* of safe connections while the other is configured as *called* side.

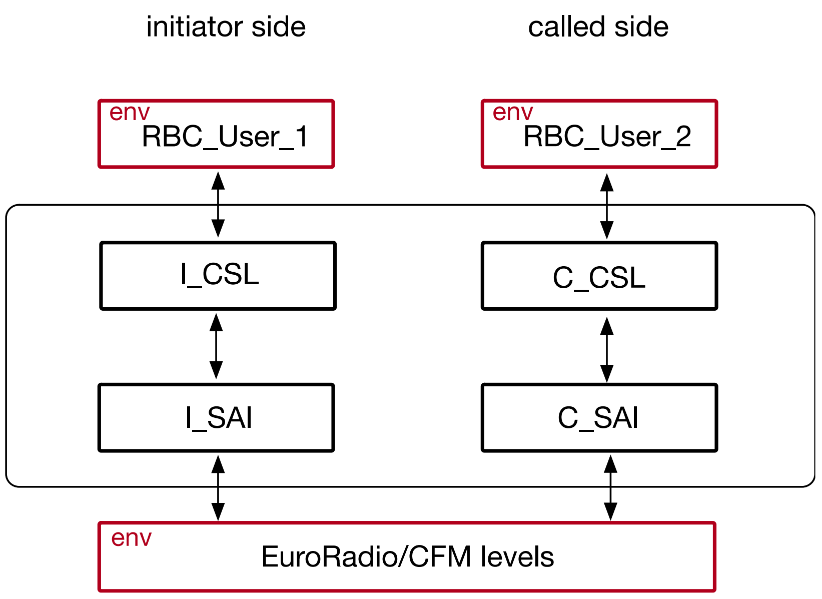


Figure 1 The signaling case study structure

Summarizing, in our case study we have seven logical components:

- Two sides of *RBC* responsible for the RBC/RBC handover transactions;
- Two sides of *CSL* responsible for the creation and supervision of RBC communications.
- Two sides of *SAI* in charge of handling the creation and maintenance of the safe connection;
- The underlying ER-SL abstracting the physical communication line between the two RBC sides.

Only the CSL and the SAI components are the object of the requirements specification, for which we have an initial natural language description in [D2.3], which is also the target of the analysis. The RBC and ER-SL/CFM components act as elements of the execution environment. They stimulate and receive data from our system components. More than one version of these environmental elements can be imagined to model different scenarios and to analyze the responses to the various stimuli of our key system components.

5.2 From Natural Language to Semi-Formal, to Formal

In line with the current trend in fact of signalling system interfaces standardisation in the railway sector (see, e.g., [EULYNX]), the first step towards the formalization of our initial natural language requirements is based on the construction of a SysML operational model of the system, in which the various components are described in terms of UML state machines.

As deeply discussed in D2.1, this choice can be rather risky due to the numerous ambiguities still present in the (still in natural language) definition of SysML/UML, and due to the great level of implementation freedom that is left to the UML-like supporting tools.
To mitigate this risk, it is essential that all the semantic details of the inter-state-machine communications are rigorously specified, and that a clear and unambiguous subset of the UML state machines features is used so that their behavior appears to be uniquely specified. However, it is definitely not a goal of the project to define a maximal UML subset. The subset used in our Demonstrator is based just on the set of clear and simple features required for the modelling of our case study. Such restrictions ease the translation from the UML state machines to the target formal notations.

With respect to the precise semantics of our inter-state-machine communications, the assumptions we made are as follows:

1. The sending of an event from one side corresponds to the receiving of the event in the event pool of the other side during inter-machine communications. Therefore, communication events are not delayed, lost, or reordered.
2. The event pool associated with each state machine is an (unbounded) FIFO queue.

The first assumption reflects the case in which communications between system components occur via shared memory (e.g., by writing into a buffer). If this is not the case, we must explicitly model the existence of a *communication component* introducing delay, loss, or reordering of messages[2].
The second assumption (i.e., the *FIFO* event queues) directly reflects the default policy suggested by the UML standard [OMG-UML, OMG-PSSM], which is in agreement with the needs of our case study.

The system requirements introduced in D2.3 describe a generic system with many configuration parameters, customizable according to several configuration options. Most of the parameters can be modelled as parameters of the state machines and take a definitive value when a specific system configuration is defined. Other aspects, like the protocols describing the SAI connection initialization phases (i.e., the TTS or ECS option), have been fixed before the beginning of the actual modelling. In particular, we chose the ECS option for our implementations. Therefore, the analysis performed refers to a specific configuration.
While the actual system is a real-time system, the planned modelling techniques do not support real-time features. Therefore, time-related aspects are reflected in the actual models only in an approximate way.

[2] This is what has been done with the introductions of the EuroRadio component.

The first step towards the definition of formal models of our system is the design of the UML state machines constituting the system itself.

A possible way, already discussed in D2.1 and D2.2, is to use a commercial Model-Based Software/System Development (MBSD) environment, like the Sparx Enterprise Architect (EA) framework, for the UML design phase. For our purposes, the Sparx-EA framework has been selected during the activity of Task 2.1. This choice has the advantage of introducing in the process a robust, commercially supported MBSD framework well integrated in the software development life cycle. The disadvantages are related to the limited utility of the framework during the requirements analysis phase. In particular, the deterministic execution model supported by the tool does not allow to observe all the possible evolutions of the SysML system, and the translation from the UML code to our target formal notations should be done by hand since it is not available, nor easy to produce, automatic translators.
In D2.2 we describe the experience of designing the CSL layers with the support of the Sparx-EA [SPARX], but then the generated UML designs have been subsequently encoded in the state machine textual format accepted by the *UML Model Checker* (UMC) tool before translation into the other target formal notations. This intermediate UMC step allowed us to perform system animations and many consistency checks, as well as to increase the confidence about the correctness of the design before affording the complex manual tasks of translating the SysML design into other formal notations. In Task 2.3 these translations are performed mechanically and consistency checks can be performed, as the design proceeds incrementally, in the used formal frameworks.

The experience gained in the second part of Task 2.1 has clearly shown the importance of being able to mechanically generate formal models directly from the SysML design. This advantage has been obtained by the development of reasonably simple translators capable of producing from the UMC textual encoding the desired formal notations (ProB [ProB] and LNT [LNT]). Therefore, the activity of designing and modelling the complete version of the system in Task 2.3 has been achieved bypassing the Sparx-EA step and directly using UMC for the design, animation, and mechanical translation of the system model into the various formal notations.

Many possible target specification languages can be selected, and even once the target notation has been chosen, many different translation schemes can be adopted.
As already mentioned in previous deliverables, there is no single formal notation, method, or tool that can act as a silver bullet for satisfying the verification needs about all the desirable properties. The world of formal verification is extremely variegated, based on very different mathematical concepts, and supported by different - often not much cooperating - communities.
In the case of requirements designs, the starting point is likely not to be a precise specification but a more abstract, parametric, often generic, natural language description, sometimes enriched with graphical artefacts as exemplified by our initial D2.3 requirements. In this case, formal methods based on model construction and model checking may be easier and more effective to apply than formal methods based, e.g., on theorem proving, that fits well the case of an already precise and correct specification to be refined and implemented.

A novelty introduced in Task 2.3 is the experimentation of a second approach (beyond the initial one based on "B" state machine notation) for the formalization of our UML design. This second approach is based on the LNT [LNT] specification language of the CADP [CADP] toolset. One interesting aspect of this new approach is that the mathematical representation used for the model is based on process algebras and can exploit the rich theory around Labelled Transition Systems (LTS) for supporting the verification process. The goal of this second experimentation is to observe if and how a compositional approach can be helpful in reducing the risk of state combinatorial explosion, so to improve the overall scalability in the analysis of the system properties.
Another interesting aspect of the CADP framework is that the structure of models in use is event-based, and in particular of communication actions. The logic used to reason on these models is a very powerful, action-based branching-time logic. This creates another point of view from the one supported by ProB, which is more state-oriented.

This "application of formal methods diversity" (in the style of [FMDR]) allowed us to solve also the issue of improving the confidence in the *correctness* of the performed translations because, starting from an initial UML model, we are able to generate two other models using different notations, and *formally* verify that the three formal frameworks actually describe the same behavior[3].
More details about UMC, ProB, and LNT notations, as well as about their translations, are given in Appendix A. The models and the source code of the translators are publicly accessible through Zenodo [ZenodoWP2].

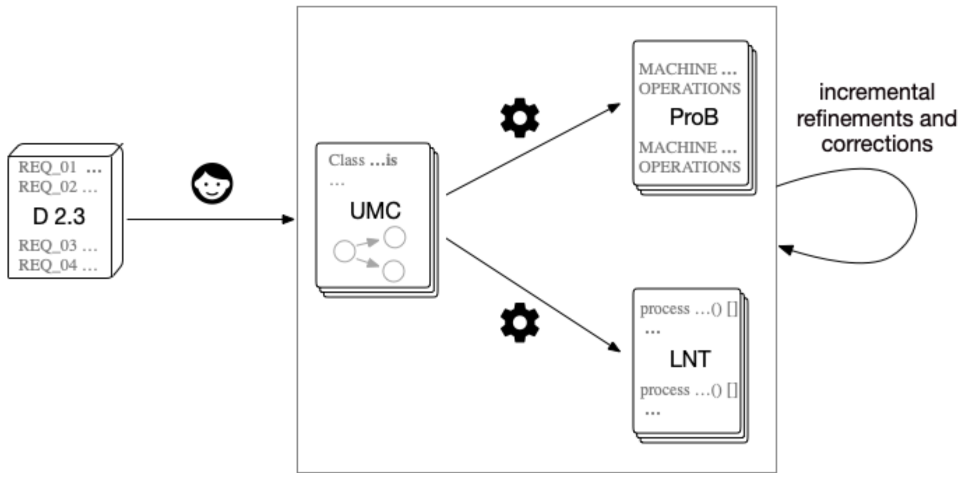


Figure 2: From NL Requirements to Formal Notations

Figure 2 summarizes the flow of information occurring in the Demonstrator when passing from the D2.3 natural language requirements to the encoding of the formal models.
The biggest and most troublesome step is the first one, where the initial requirements must be interpreted, bypassing all the possibly existing ambiguities and inconsistencies, and where all

[3] i.e., The labelled transition system which can be generated from them are strongly bisimilar.

those aspects that are intentionally left unspecified must be in some way implemented to generate an actually executable (and possibly nondeterministic) model. The second step consists of the mechanical translation from the UMC model to the ProB and the LNT one, so to perform static and other lightweight formal analysis activity on all of them. Clearly, the first and second steps are iteratively repeated each time the prototypical UMC model is corrected and/or refined with the progress of the modelling.

In this first step, many weaknesses of the initial document can already be spotted, especially ambiguities, inconsistencies, duplications, and missing points.
Appendix D reports the complete list of the weaknesses in the initial requirements identified or further specified in the Demonstrator.

Let us consider, for example, REQ_064 of D2.3:

REQ_063	If N (configurable) consecutive messages are missing in the sequence of the received messages, i.e. if a message whose sequence number is greater that the sequence number of the last correctly received message + N, the message shall be ignored and the SAI shall send an order to terminate the safe connection to ER sublayer and report its state to CSL.

The underlying idea is rather clear: all the messages are sequentially numbered. Looking at those numbers, it should be possible to detect if the incoming message is either a duplication, or a previously expected message that has been overtaken by a subsequent one, or a new message showing that several other messages have been lost (or further delayed) during the communication. In other words, we can compute the distance (in terms of sequence position) between the incoming message and the previous one. Based on the above, four cases may occur:

1. if the distance is equal to *1* then all is OK;
2. if the distance is lower than 1, the message is old and therefore discarded;
3. if the distance is between *2* and *N* (limits included) the message is still accepted (an error is notified);
4. if the distance is greater than *N,* the message is discarded and the connection is closed.

The problem is that sequence numbers have an upper bound range defined by the number of bits allocated for the field, and after reaching the maximum value, a new sequence number series restarts from 0. Therefore, the computation of the *distance* cannot be just the value *current_received_seqnum* minus *last_received_seqnum* but, more properly, it must take into account the range overflow. For small distances this computation is rather intuitive: the distance between *max_seq_num* and *zero* is *1, and* the distance between *2* and *max_seq_num - 1* is *4*. But if the distance is very great (e.g., *max_seq_num / 2*), it might become not obvious to decide if the last received message is a very old one (to be discarded) or a new one notifying the loss of a long sequence of messages. This aspect is neither properly covered in the original SUBSET-98 standard which describes the operations to be performed without taking into account the possibility of range overflow.
For computing such distance, in our implementation[4], we opted to perform the following:

[4] see *SEQ_NUM5* in Appendix C for more details.

```
distance := seqnum - last_received_sequence_number;
if  (distance <  -M/2) then {distance :=  distance + M };
else if  (distance  >  M/2) then {distance := distance - M };
```

Where *M* is, in our case, a model parameter that specifies the upper limit of sequence numbers. However, the above calculus remains an implementation dependent aspect that might actually affect the system behavior and creates interoperability problems if not clearly specified.
A similar issue exists in the management of execution cycle numbers, used to evaluate the delay occurred for the transmission of the message.

Another example of ambiguity/implementation freedom is related to D2.3 REQ_008. Again, the rationale is clear: While in state *COMMS* (Connected), the *CSL* should not let pass a certain period of time (send timeout) without sending a *DATA.request* or a life sign to the *SAI*.
A potential implementation freedom is left, when the *CSL* moves to *Connected* state, about the delay to be awaited before sending a first life sign (if no user data request is pending). In fact, it might be allowed (and useful) to immediately send a first life sign without delay; if the ExecutionCycle option is active for the *SAI* initialization process, the called *SAI* should wait for a first *DATA.indication* message before moving to the *Connected state*. If we send the life sign only when the send-timeout delay is expired, the successful creation of the connection becomes more difficult.

It is also in this first step that, unfortunately, coding errors can easily be introduced while encoding implementation dependent aspects. Fortunately, there are some lightweight formal analysis techniques that can be exploited with very little (or null) effort to repeatedly perform, as the design progresses incrementally, quick but very useful checks.

Static Analysis

All tools automatically perform some degree of static analysis on the models to which they are applied. Static analysis of the resulting SysML executable design is particularly useful for detecting early errors introduced in the UML encoding. Since UMC supports only a limited form of static analysis, it is useful to generate the ProB and LNT versions in order to exploit the more advanced static analysis features of these environments.

For example, a LNT warning about the existence of a declared variable not later used in the body of a process statically revealed the presence of some anomaly in the code of the SysML model. On the other side, ProB revealed statically a type error on the expression:

(currentEC - OFFSET) mod Mec

pointing out the issue that the *modulus* operation is not unambiguously defined in presence of negative values. For instance:

- in C the expression "(-2) % 7" evaluates "-2";
- in Ada the expression "(-2) mod 7 " evaluates "5"[5].

[5] In our model we modified the code so that modulus operations are always performed on positive values,

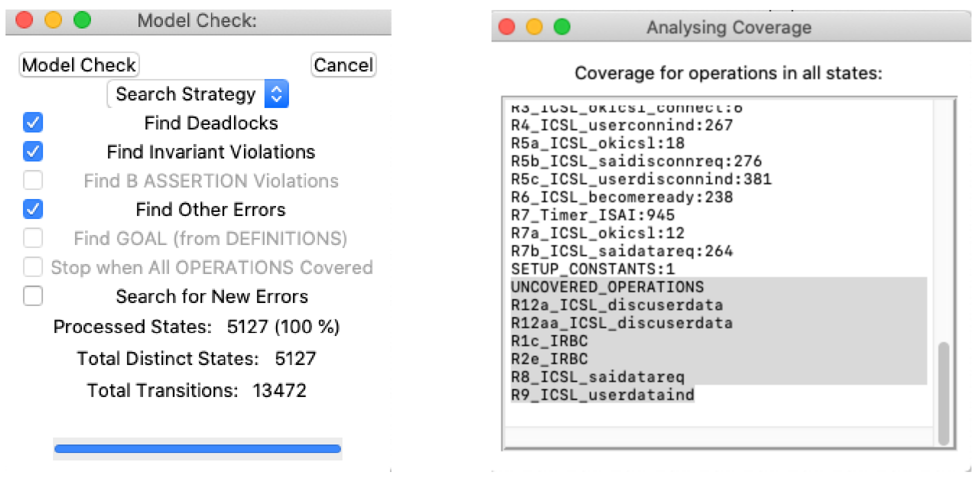
Interactive simulation

Interactive simulation of the whole system, controlling all the system nondeterministic aspects, is another useful method for the early discovery of coding errors that do not require advanced competencies in formal methods. All the three formal frameworks allow an interactive exploration of *all the possible* system evolutions, and interactive simulations over them have actually allowed to quickly identify simple coding errors (e.g., duplications of rules, copy & paste mistakes, etc.).

Full statespace exploration

Full statespace exploration can be easily triggered in all the three environments in a simple push button way. Full statespace exploration reveals *runtime errors* (e.g., violation of invariants), deadlocks, or missing requirements (i.e., situations for which no rules specify how to system evolution should progress). This a very powerful method that has been extensively used during our incremental design and analysis process for detecting mistakes. Clearly, when the model statespace becomes rather big, this kind of analysis may require (too) much time for being routinely used during the incremental design.

In UMC, the absence of a necessary transition rule can be detected through the observation of a "lost event" event. In ProB and LNT, with the current encoding, the absence of a transition rule causes an observable deadlock (the state machine is not able to remove the top element from the queue of events).
In ProB ,full statespace exploration is activated, as shown by the left side of Figure 3, by just selecting the default “Verify -> Model_Check" command.
In CADP, the statespace exploration can be requested with the command "bcginfo", or by evaluating on-the-fly the formula "[true*] <true> true".
In UMC the statespace exploration can be requested the command "umcstats" or evaluating the formulas "EF FINAL" and "EF {lostevent}".
In case of failure of the above tests it is possible to observe the execution trace that leads to the failure (and in the case of Prob and UMC also visualize the execution trace as a message sequence chart).



- ✓ Find Deadlocks
- ✓ Find Invariant Violations
- ☐ Find B ASSERTION Violations
- ✓ Find Other Errors
- ☐ Find GOAL (from DEFINITIONS)
- ☐ Stop when All OPERATIONS Covered
- ☐ Search for New Errors

Processed States: 5127 (100 %)
Total Distinct States: 5127
Total Transitions: 13472

Coverage for operations in all states:

```
R3_ICSL_okicsl_connect:6
R4_ICSL_userconnind:267
R5a_ICSL_okicsl:18
R5b_ICSL_saidisconnreq:276
R5c_ICSL_userdisconnind:381
R6_ICSL_becomeready:238
R7_Timer_ISAI:945
R7a_ICSL_okicsl:12
R7b_ICSL_saidatareq:264
SETUP_CONSTANTS:1
UNCOVERED_OPERATIONS
R12a_ICSL_discuserdata
R12aa_ICSL_discuserdata
R1c_IRBC
R2e_IRBC
R8_ICSL_saidatareq
R9_ICSL_userdataind
```

Figure 3: a result of standard ProB checks.

Another rather standard check is the analysis of the coverage of the state machine transitions. With ProB this can be obtained directly, once performed the previous model checking, with the command *Analyse -> Coverage -> Operation Coverage*.
Activating this check on the *ICSLtesting_V27_nodata we* obtain the result shown in Figure 3 (right side), from which we can see that there are several transitions that are never triggered, but this is precisely what we would expect given the no data request or data indication messages are ever generated.

Reachability analysis

Finally, reachability analysis requires a little more effort in writing simple logical formulas but allows to observe specific executions traces (In the case of ProB and UMC also in the form of message sequence diagrams) that lead to a given situation or event. For example, a simple reachability property like,

"Eventually, in at last one execution, the initiator CSL receives the notification of the establishment of a safe connection"

can be encoded:

- **in UMC** as *EF {ISAI_Connect_confirm}*[6]*;*
- **in ProB** as *not G not [R4_ICSL_userconnind]*[7]*;*
- **in LNT** as *<true*.ISAI_Connect_confirm>true.*

The verification of reachability properties like the above one also allows (in the ProB and UMC cases) to display a requested execution paths in terms of a user-friendly sequence diagram usable for documentation purposes.

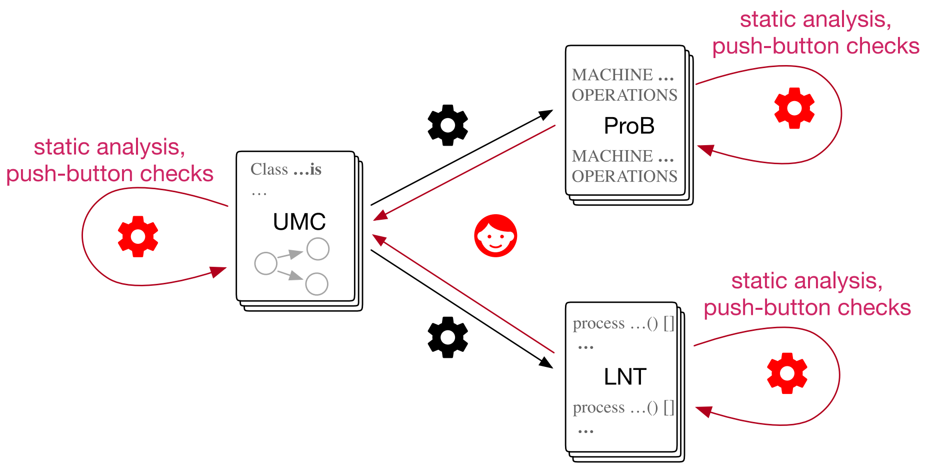


Figure 3: "lightweight" use of Formal Methods (static analysis, statespace exploration)

[6] ISAI_Connect_confirm is the event correspond the delivering of the notification
[7] R4_ICSL_userconnind is the label of the CSL transition (operation name in prob) accepting the notification

The above verification techniques already allow to increase the confidence that the created models of the various system components reflect a clear and complete design, not an inconsistent one. Figure 3 summarizes such lightweight use of formal methods.
Apart from that, we want to see if it is possible to go beyond these properties and try to provide some evidence to reply to the following question:

Are all the components and the whole system doing what it is actually desired?

This question encloses within it two crucial ones:

- *Have we correctly modelled the initial natural language requirements?*
- *Actually, are those requirements correct?*

We do not know for sure *what it is actually desired*, and *which were the designer intentions behind the initial natural language requirements*, but we can provide some feedback on the overall behavior of the system (or some components), hoping that the feedback confirms the designer's expectations.
We should also remember that we are in the phase of *constructing/analysing* the requirements, not in the phase of developing a system starting from some presumably rigorous and correct requirements.
In the following section, we attempt to provide appropriate answers to the above questions so to clarify the doubts related to this problem.

5.3 From Formal, to Semi-Formal, to Natural Language

Abstracting from the introduced implementation details

During the UMC encoding of the system, several design/implementation choices have been made that appear not to be explicitly specified by the initial D2.3 requirements. This is normal, because system requirements are usually at a higher level of abstraction than a directly and fully executable model.

> As a trivial example, while in the requirements we have a rule stating that "*if something does not happen within a given timeout, something else should be done*" in our encoding, we have a *timer* object sending *tick* events, a *counter* variable initialized with 0 and incremented at each tick event, and a *check* on the value of the counter that triggers the timeout-related activities.
> Similarly, while in the requirements we have the rule stating that "*periodically we should set an ack-request flag in the next outgoing message unless the previous request is still waiting for a response*", in the actual encoding we might have an *ack_request* counter, appropriately initialized and incremented at each *tick* event, a *variable* recording the fact that the flag should be set in the next outgoing message, and another *variable* recording whether there are still pending requests.
> Moreover, to reduce the state-explosion effect, it is advisable to reset all the variables to some

default static values as soon as their current value is no longer needed.

However, it is definitely useful to present a graphical view of the encoded UML state machine that abstracts away again from all these details, allowing a reader to understand the overall structure of the actually modelled design without being overwhelmed by all these implementation details. These abstract graphical views of the UML state machines actually composing the formal model of the system may provide a first kind of evidence towards a reply to the main question raised in the previous section, that is:

- *Have we correctly modelled the initial natural language requirements?*

In Appendix B are shown the 4 (graphical, semiformal) state machine diagrams corresponding to our relevant system components (i.e., initiator and called *CSL*, initiator and called *SAI*).

Together with our new abstract, semiformal UML state machine models of the system components, it would make sense to associate them with a new system specification in the form of structured, rigorous natural language requirements[8]. These new requirements should now overcome all the potential weaknesses present in the informal natural language requirements that have been taken as input for our formal analysis process.
We believe that it is an important point to have strictly connected natural language, semi-formal, and formal artefacts as an output of our formal analysis process. Figure 4 illustrates the resulting information flow generated by our demonstrator.

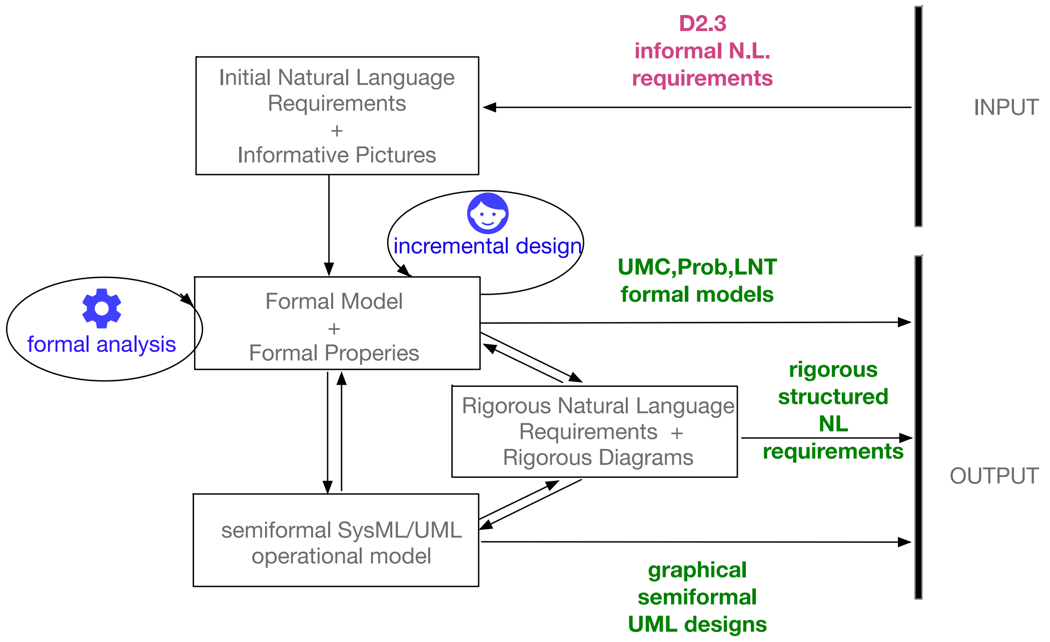

D2.3
informal N.L.
requirements
INPUT
Initial Natural Language Requirements + Informative Pictures
incremental design
UMC,Prob,LNT
formal models
formal analysis
Formal Model + Formal Properies
Rigorous Natural Language Requirements + Rigorous Diagrams
rigorous
structured
NL
requirements
OUTPUT
semiformal SysML/UML operational model
graphical
semiformal
UML designs

Figure 4 Input and Outputs of Formal Analysis Process

[8] The process might also be reversed, i.e., starting from the formal models we might produce the rigorous, structured, NL requirements, and from these than generate the abstract semiformal UML designs.

The output of the analysis process becomes usable when the complete system is finally designed in its entirety, the generated formal models[9] are correctly described by the rigorous natural language description, their overview correctly presented in terms of graphically semi-formal SysML/UML models, and sufficient confidence is gained on the fact that all the properties of interest are satisfied.
Notice that there is not a precise "ending point" for the process. More "properties of interest" can be identified at a later time, and a "greater level of confidence" might be desired, triggering the creation of further architectures and scenarios to be analysed.

<u>Making explicit assumptions and guarantees</u>

This natural language description, for each system component under design, should describe in a precise way:

- the *parametric aspects of the component;*
- the *interface* towards the outside of the component (i.e., the messages sent and received);
- the *assumptions* on the external environment which underlies the component definition (whose violation might compromise the correct component behavior);
- the *requirements* on the internal functional behavior of the component;
- the *guarantees* that the component should ensure towards the external environment.

Most of this information, and in particular the assumptions and guarantee related aspects, are somewhat already present also in our initial D2.3 requirements, but often in a not-well-structured, explicit and clear form, and sometimes only in the form of external references to other standards. The presence or the absence of external assumptions may play a relevant role in the design of a component, and inconsistencies on these aspects may lead to interoperability problems.

> For example, in the D2.3 requirements, the assumptions on the expected behavior of the *ER-SL* are completely missing. Looking at the UNISIG-SUBSET-037 [SUB-037] standard, we have observed that the *SAI* can assume that the *ER-SL* always responds (eventually) with either a *Sa_CONNECT.confirm* or a *Sa_DISCONNECT.indication* to a *Sa_CONNECT.request*. Such information led us to a design in which as the component is in the *Connecting* state, further *SAI_CONNECT.request* orders (triggered by a connection timeout) from the *CSL* are discarded. If the above assumption would not hold, the *SAI* design would become inconsistent and at risk of deadlocks.

In Appendix C, we present the Rigorous Natural Language rewriting of the requirements that, applying our demonstrator process, have been associated with our formal and semi-formal models. These new rigorously structured natural language requirements play two very important roles:

- They constitute a clear *human-oriented* documentation artifact of the system specification;

[9] It is in this step that the the system properties of interest are indentified.

- They appear to state the properties that are really *expected to be satisfied by the system*, still using the natural language, but in a form more amenable to confirmation by formal analysis.

5.4 Verification Architectures and Scenarios

Prologue

"All models are wrong, but some are useful"

The above is a famous quote [BOX] from the statistician George E. P. Box. The meaning of the quote is that all models are, necessarily, an abstraction and an approximation that fails to represent reality in all its details. This means that from a rigorous point of view, models are all wrong. This does not exclude, however, that in their abstraction they allow reasoning in a simple way on specific aspects of the system, getting useful insights and confirmations or counterexamples about the expected behaviors of the system. However, we should be careful not to consider them as a *gold standard*, forgetting the implicit assumptions and abstractions which are at their base.
In our demonstrator, starting from the initial natural language requirements, we progress by designing operational UML models of the system. In doing that, it is important to state explicitly all the assumptions and abstractions that underlie the model design. Moreover, we should not forget that the resulted model is just one of the possible models that could be designed, as the natural language requirements are usually and intentionally at a higher level of abstraction (and ambiguity) than the specific operational design that if being modelled.
The operational UML models of the system constitute the base to derive our verifiable formal specifications.
The correct question we should ask about these specifications is therefore: *Is our formal model good enough for reasoning on the properties of the real system in which we are interested?*
The answer to the question partly depends on the available verification functionalities provided by the selected formal framework and partly depends on the various steps of abstraction and approximations performed from the initial system requirements. But it also depends on the *correctness* of the translation and encoding of the model into the notation of the formal specification.

Architectures

When reasoning on our *CSL* and *SAI* components, we have used two ways to build verification architectures for our analysis. As shown in Figures 5 and Figure 6, the first one is to build an architecture in which a single system component interacts with abstract models of the environment that satisfy only the set of required assumptions. Such environmental components must be consistent with the system component to be analysed, and able to stimulate all the possible interactions with it. This kind of architecture remembers the "single component stress testing" of a module, with the difference that through model checking are analysed *all* the possible component behaviors.

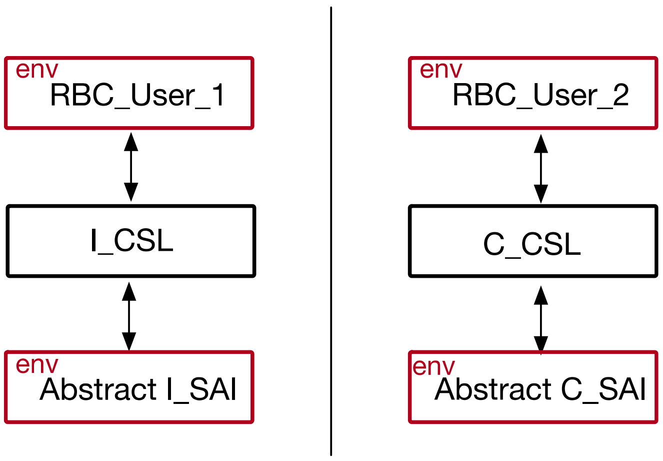


Figure 5 Testing CSL components in isolation

This kind of architecture has the advantage of being simple, and it is useful to check the consistency, safety, and robustness of the design. The environment, in this case, might also behave in ways that in practice might not occur when replaced by the actual software and hardware components. This kind of verification may also show undesirable behaviors of the system component that are not necessarily caused by mistakes in its design, but, more properly, they are due to the absence of further assumptions on the environment beyond those already stated.

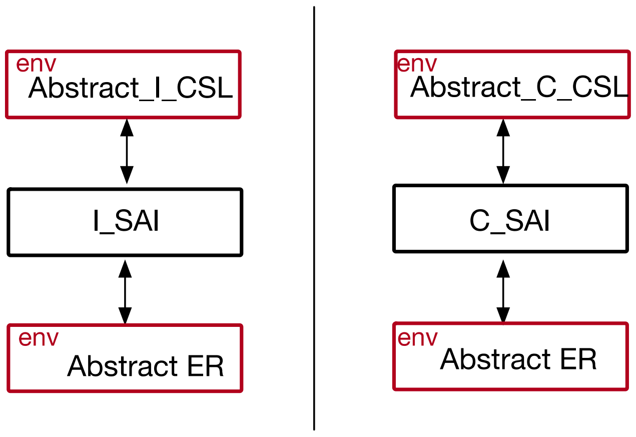


Figure 6 Testing SAI components in isolation

In order to analyse the interacting behavior of the components at the two sides of the same layer (i.e., initiator and called CSL), we need more complex architectures that integrate all the needed components as shown, for example, in Figures 7, 8, and 9.

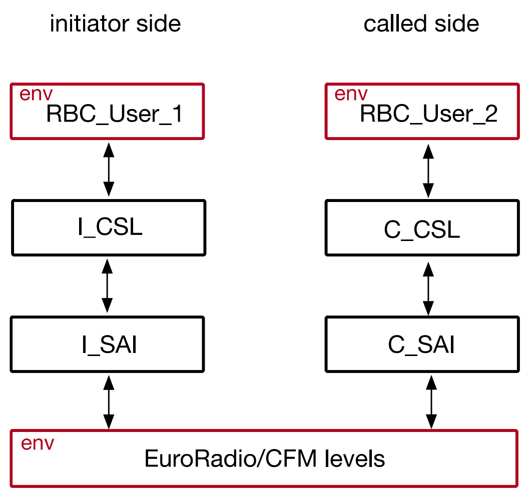


Figure 7: the architecture of the complete system

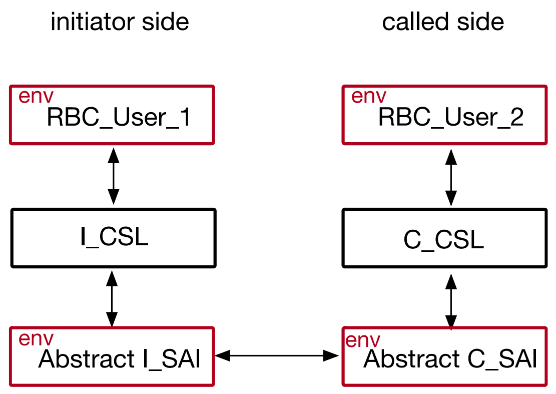


Figure 8: CSL layer testing architecture used in Task 2.1

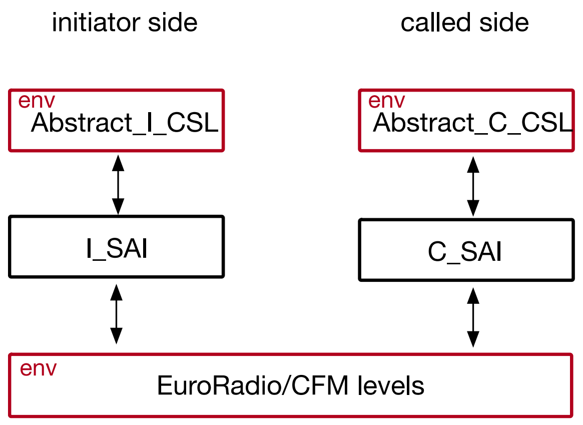


Figure 9: SAI layer testing architecture

If we model all the needed components as UML state machines (therefore instantiating in some way of the parametric aspects of the components), we can exploit the mechanical transformation of the architecture into a verifiable formal scenario without any further effort (apart from that of removing coding errors in the design of the new environment components).

Scenarios

Once fixed the overall architecture, we have that our system components may actually depend on several system parameters (e.g., timeouts, limits), and each instantiation for these parameters gives rise to a particular scenario used for the verification process.
Moreover, the SAI layer definition given in D2.3 is a generic system definition, whose instantiation can be configured according to a predefined set of options: Safe connection initialization through *Triple Time Stamp* (TTS), *Execution Cycle Start* (ECS). Each option gives rise to a different system and must be specifically instantiated before formally reasoning on it. In particular, our demonstrator models the SAI ECS option.

With respect to the other *environment* related components, like the RBC Users, the ER-SL component, or abstract versions of the CSL and SAI used to stimulate the other component under analysis, we have that several versions can be generated depending on the kind of properties that we are interested to observe and analyse. For example, when analysing the Safe Connection creation and abort events in the system, we are not interested in the values actually exchanged between the RBC User components, and we might ignore aspects related to the duplication or reordering of messages, therefore testing the system under restricted conditions that make the analysis easier to be performed.

Finally, in our architecture, we must introduce *timer* components that still allow the various system components to asynchronously proceed in parallel, but still preserving a certain degree of comparable speed. Several variations are possible in this case as well, and such variations give rise to several different verification scenarios.

5.5 Advanced Formal Analysis

The source of properties to be analysed

As previously mentioned, the initial D2.3 requirements cannot be considered as a correct base for the specification of the system properties to be formally verified. In Appendix D the original D2.3 requirements, annotated with the main interpretation problems faced during the analysis, are reported.

Let us consider, as a simple example, REQ_070 of D2.3:

REQ_070	At start-up, and when loss of safe connection is detected (Sa_DISCONN.indication), the SAI, if configured as initiator, shall wait for order from CSL.

- The event signaling the loss of safe connection is *Sa_DISCONNECT.indication,* not *Sa_DISCONN.indication*
- The meaning of "order" is not clear. It can be imagined that a *SAI_CONNECT.request* is meant.

- The meaning of "wait" is not clear. What is the component allowed to do while "waiting"? If the normal handling of all the incoming events (as described by all the other requirements) is what is actually intended, then this requirement actually does not describe anything.

Any attempt to directly "formalize" this requirement in terms of logical formulas would make no sense.

Let us now see how the behavior of the initiator SAI, with respect to this situation, is instead described by our new requirements[10] (the "order from CSL" is the *SAI_CONNECT.request*):

- **R1**: At startup, the SAI is in Disconnected state.
- **R2**: When in *Disconnected* state is received a *SAI_CONNECT.request* from the *CSL* component, the *SAI* sends a *Sa_CONNECT.request* to the *ER-SL* and moves to *Connecting* state.
- **R4**: When in *Connecting* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.
- **R9:** When in *Initializing* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.
- **R15:** When in *Connected* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* sends a *SAI_DISCONNECT.indication* to the *CSL* component and moves to *Disconnected* state.

The above requirements precisely correspond to the abstract structure of SAI state machine[11], and there is no formal encoding of logical formulas to be done and to be verified.

In fact, most of the *requirements on the internal functional behavior* of the CSL component have the form:

< When in a certain *state* a certain *event* occurs, and certain *local conditions* hold, then certain *effects* should occur>

It is worth pointing out that such requirements directly reflect the structure of the formal and semi-formal models. In other words, in the UML design each transition that generates certain effects when activated under certain conditions, has a corresponding requirement that precisely specifies the relationship between conditions and effects without ambiguities, redundancies, or inconsistencies.

Not all the new requirements on the internal functional behavior of the system have the above *state:event[guard]/effect* structure.
Several other requirements, typically those expressing complex data flow relations (like requirements over SAI sequence numbers, ack management, ECS counter management), do not have an immediate correspondence with the state machine structure.
Also in this case, however, passing through a temporal logic encoding of the whole property and relying on explicit model checking does not seem to be an advisable (if ever feasible) approach.
For our experience, a combination of code inspections, statespace exploration and minimizations, and the use of the model checker like a debugger for the analysis of simple variable related properties is a much more viable solution.

[10] The complete list of new requirements is shown in Appendix C.
[11] The graphical representation of the abstract SAI and CSL state machines are shown in Appendix B).

We have previously described two important parts of system component specification, that is:

- The assumptions on the other system components that are needed to guarantee the correct behavior of the component;
- the guarantees that the component can ensure to the rest of the system.

Clearly, these two aspects are correlated: if component C1 assumes property P from component C2, necessarily component C2 should guarantee property P to the environment.
Formal verification can be a useful technique for verifying that a component actually guarantees the assumption on which other components rely.

In our case, the requirements in D2.3 are not very precise in stating the assumptions/guarantees associated with the various components. This is probably also a consequence of the fact that for each side, the CSL and SAI components are supposed to be developed by the same provider, which is likely to have a complete and detailed knowledge of the whole architecture on its side.
Nevertheless, we believe that it is useful to make explicit the dependencies between the components on the same side because, even if developed by the same provider, the two components might be actually developed by different teams, and a clear documentation of the dependencies between components is surely welcome.

> One of these assumptions, not clearly stated in the D2.3 requirements, is that the initiator SAI should always reply with a SAI_DISCONNECT.indication message to a SAI_DISCONNECT.request.
> In the absence of such a reply, the CSL would remain forever in the NOCOMMS Waiting state.
> In this case, the proof of this guarantee on the SAI side can easily be obtained by just observing the transitions in the SAI statechart that are triggered by the SAI_DISCONNECT.request signal.
> Another assumption which underlies our SAI design is that the EuroRadio sublayer should always, eventually, reply with a Sa_CONNECT.confirm or Sa_DISCONNECT.indication to a Sa_CONNECT.request. While in state Connecting, in fact, the SAI discards further connection requests from the CSL while the current one is still in progress. Failure to reply from the ER side would therefore create a deadlock.

<u>Encoding properties with temporal logics operators over state and event predicates</u>

We have already seen in Section 5.2 some simple examples of reachability properties that can be encoded and verified without much effort.

Even if, in most cases, using model checking for verifying simple structural property can be just overkilling, because just a plain observation of the CSL state machine diagram would allow us to easily check the property, sometimes we might be interested to still formally check functional properties that can be expressed in logical terms by composing state and event predicates.

For instance, let us consider, for example, the property[12]:

[12] This is a property which refers directly to the UMC model

"it never happens that the initiator CSL forwards an RBC User data request to the SAI when not in state COMM"

In our formal frameworks this property can be formalized as:

in UMC[13] as: *not EF (not inState(COMM) and <ISAI_DATA_request>)*
in ProB as: *not F (not {ICSL_STATE=COMM} and [R8_ICSL_saidatareq])*
in LNT: *not expressible without changes in the model.*[14]

In other cases, the property of interest cannot be directly mapped on temporal operators over state or event predicates, and its encoding can require rather advanced formalization capabilities (e.g., parametric fix point operators) and some more advanced knowledge of the theory behind the used formal methods.

For instance, let us consider the property:

"the messages received by the called RBC contain a continuously growing value" (i.e., no reordering occurs).

In our formal frameworks this property can be expressed as:

in UMC as:

```
AG ( [CRBC_User_Data_Indication($v1)]
        not EF {CRBC_User_Data_Indication($v2)} (%v2  <= %v1) )
```

in LNT as:

```
mu X (n : nat := 0).
  ( [ true ] false
    or  ( [ { CRBC_User_Data_Indication(?m:nat)} ] if m >= n then X(m+1) else false end if
          and  [ i ] X (n)  )  )
```

in ProB: *not expressible.*[15]

When the system becomes rather big, as in our case happens when we try to analyse the complete system composed by all the seven components, it is likely that full statespace exploration becomes impossible or very expensive in terms of time and resources.
On-the-fly model checking techniques, may allow to verify system properties without requiring the generation and analysis of the full statespace.

In our scenario ERnice_irbcdata_V53 we have modelled an environment in which:

- The initiator RBC waits for a Connect indication and sends five messages to the other side,
- The ER level is a "nice" one which does only introduce acceptable delays, does not lose or reorder messages, and does not autonomously abort the safe connections.

In this context we would expect that the five messages sent from the initiator side will all arrive to

[13] Appropriate "Abstractions" must be defined in UMC to make these basic predicates observable.
[14] The values of local variables of a LNT process specification are not observable from the supported logic.
[15] In ProB there are no parametric fix points in the logic, and it is not possible to express relations between the values of parameters or local varables in different states.

the called RBC side and in the correct sequence, and that no Error reports nor Disconnect indications are ever generated.
After an appropriate fine-tuning of scenario timeout parameters (receive timeout, send timeout, initialization timeout, connection timeout, ack-response timeout), we can verify that this is precisely what happens by evaluating the following formulas (shown in the UMC style):

- "*The called RBC does never receive any message before a Connect indication*".
  A[{not CRBC_User_Data_indication} U { CRBC_User_Connect_indication}]

- "*The called RBC , after receiving a Connect indication, always receives as first data message the first data message sent by the initiator RBC*".
  AF {CRBC_User_Connect_indication}
  A[{not CRBC_User_Data_indication} U { CRBC_User_Data_indication(1)}]

- "*The called RBC , after receiving a first data message, always receives as second data message the second data message sent by the initiator RBC*".
  AF {CRBC_User_Data_indication(1)}
  A[{not CRBC_User_Data_indication} U { CRBC_User_Data_indication(2)}]

- "*The called RBC , after receiving a second data message, always receives as third data message the third data message sent by the initiator RBC*".
  AF {CRBC_User_Data_indication(2)}
  A[{not CRBC_User_Data_indication} U { CRBC_User_Data_indication(3)}]

- "*The called RBC , after receiving a third data message, always receives as forth data message the fourth data message sent by the initiator RBC*".
  AF {CRBC_User_Data_indication(3)}
  A[{not CRBC_User_Data_indication} U { CRBC_User_Data_indication(4)}]

- "*The called RBC , after receiving a fourth data message, always receives as fifth data message the fifth data message sent by the initiator RBC*".
  AF {CRBC_User_Data_indication(4)}
  A[{not CRBC_User_Data_indication} U { CRBC_User_Data_indication(5)}]

- "*Until the all the five messages have been received, the called RBC does never receive a Disconnect indication*".
  A[true {not CRBC_User_Disconnect_indication} U {CRBC_User_Data_indication(5)}]

- "*Until the all the five messages have been received by the RBC, the called CSL does never receive any error report*".
  A[true {not CCSAI_Error_report } U {CRBC_User_Data_indication(5)}]

Notice that the above properties can be checked by observing only 16million states (a check that

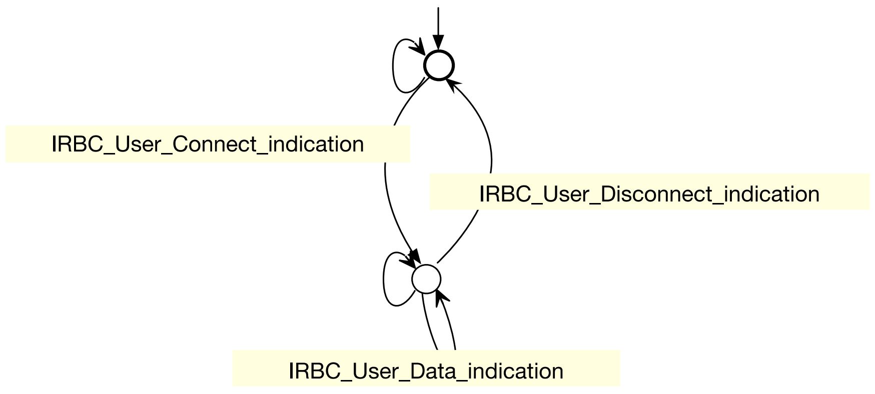
can be done in a bunch of minutes) while the complete statespace has still an unknown size but in the order of several hundred-million states.
The violation of this properties, and the observation of the counter-examples, has also allowed to detect and correct several (some of which severe) implementation errors in the SAI model.

Verifying with Observers

In many cases, like the one related to the check of the growing values in arriving messages, the simplest solution is that of building a specific scenario where the environment acts as an "observer" of the intended property. This is what is done, for example, for the verification of the previously mentioned property. In this case, we need to define a (*called*) RBC User element that saves the last value received and makes a comparison between it and the current value each time a new message arrives, notifying the error if the check fails.
In this way, the complex to encode property becomes a simple to write/understand reachability property. This solution might not always be feasible, but when it is possible it can solve the problems of hard encoding and verification of logical properties. Encoding a complex logical property can really be a very error-prone task.

Abstract Overviews of the System Behavior at the interfaces: Minimized information flows

While reasoning on the possible behavior of a system component (or group of components), it is sometimes useful to observe all the possible information flows, regarding a small set of selected messages, that may occur, e.g., at the interface between two components.
For example, let us suppose that we want to observe, at the interface between the CSL and the RBC User at the initiator side, all the possible sequences of messages flowing from the CSL to the RBC. Starting from an architectural description of the system that includes both sides of the CSL (in a given scenario), it is possible to mechanically extract and visualize all the possible streams of RBC connect, disconnect, and data indications at the initiator side[16], obtaining the picture shown in Fig. 10.


IRBC_User_Connect_indication
IRBC_User_Disconnect_indication
IRBC_User_Data_indication
Figure 10: all the possible messages flow from CSL and RBC (initiator side)

By just observing this picture, several properties of the scenario can be observed. For instance,

[16] In Appendix E (Model reduction techniques) is shown the theory and practice of this approach.

- It is possible that no connection ever occurs (self-loop in the initial state);
- a disconnection indication is always preceded by a previous connection indication;
- data indications may arrive only after a connection, in no disconnection has occurred in the meanwhile;
- data indications might never arrive;
- disconnect indications might never arrive (the CSL remains connected forever);
- after a disconnection, there is no guarantee that a new connection will follow.

All these properties might also be explicitly verified by encoding them as logical formulas and evaluating them in this scenario, but the mechanical generation of the description of the possible sequences of interest might be a more friendly approach to the system analysis.
The advantage of the full model checking approach with respect to the model reduction approach, is that in the first case we might also ask for an explanation of the result of the verification and obtain, for example, a detailed sequence diagram that shows how it can happen that the system is never successful in establishing a communication line. Moreover, the model reduction approach requires the traversal and analysis of the full statespace, and in case of complex systems this might become unfeasible.

6 Conclusions

"The actual goal of our demonstrator"
We have associated our initial requirements with a precise, executable SysML/UML model. Then we have translated it into rigorous formal specifications, after which we have embedded the formalized components in specific verification architectures and scenarios[17], and started making rigorous analysis upon them. Our goal, however, is neither to complete the "validation" of the initial system requirements, nor to provide a generic "proof of correctness" of the formal design. The actual goal of the 4SECURail demonstrator is just to show *if and how* certain tools and methods can *improve* our confidence that *specific* properties (about safety, interoperability, functionality) are guaranteed by our formal models and, therefore, *likely* supported by our system requirements. Indeed, considering the role of the demonstrator inside the whole project, we are interested to show the *kind* of question we can study, the *kind* of answer we can obtain, the *difficulty* of the process, and the *kind* of feedback returned to the user by this activity.

"The output produced by our demonstrator"
The choice of selected case study and structure of the formal methods demonstrator process have proved to be very effective for illustrating, in a qualitative way, the advantages that can be obtained by the adoption of semi-formal and formal methods in the early phase of system requirements specification, as well as the difficulties that can be encountered in this activity.

Starting from requirements defined in D2.3, the application of our formal methods demonstrator process has allowed us to derive:

- A new rigorous / formally backed requirements specification of the system. Descriptions are provided in Appendix B, Appendix C, and their discussions are present in Section 5.5).
- A list of weaknesses in the initial D2.3 requirements. These are reported in Appendix D.

The experience gained with the design and exercising of the demonstrator with our case study has also allowed us to highlight several "takeaways" that we have observed and found most relevant during the activity of Task 2.3, which are summarized below.

"System requirements definition and analysis are very different from system implementation"
The activity of transforming the designer's intentions into a system requirements document is very different from taking a system requirement document and developing an executable system from it. In particular, the role that formal and semi-formal methods play is very different between these two activities.
For the development phase, the focus is likely to be on the "correctness" of the developed product with respect to its requirements. If formal methods are adopted in this development phase, they are focused on guaranteeing the correct transformation of a semi-formal design into executable code (e.g., by formal refinements).
For the requirement construction phase, the focus is likely to be on two other aspects:

[17] Architectures and scenarios are discussed in Section 5.4.

- The *precision* (i.e., completeness, non-ambiguity, safety, internal consistency) of the requirement document.
- The *external consistency* (i.e., interoperability) of the system specification with respect to the other systems with which it must interact.

In the absence of *precise* and *consistent* requirements, any effort on the developer side to adopt formal methods during the development phase risks being useless or counter-productive because a rigorous implementation of misleading or non-interoperable requirements will likely lead to implementation errors.

"The need of UML design guidelines to support simple, well defined UML design"
Indeed, as widely recognized in many papers (many of them already cited in D2.1 and D2.2) and project results (e.g., X2RAIL2), the use of UML as a specification language for System of Systems can be very problematic because of its generality. Too many "hidden" assumptions are concealed within the UML designs and might have a strong impact on the expected behavior of the system. This problem can be overcome if:

- We take care of explicitly stating all the otherwise hidden assumptions in the design.
- We restrict the use of UML features to those which currently have a clear semantics and for which there is a clear and simple way to be translated into a (one or more) formal notation.

This is precisely what we have tried to do with our demonstrator.

"The need of mechanical generation of formal models"
Mechanical translations from UML designs to formal specification languages are not just highly preferable (as already stated in D2.1 and D2.2), but reveal to be *mandatory* for any exploitation of formal methods from semiformal UML designs. In our specific demonstrator case, we would not have been able to deal with our more than 50 refinement/correction steps if we had not developed our translators from the UMC to the ProB and LNT notations. From our point of view, the ideal source for this transformation should not be a *vendor-specific* XMI representation of the UML design as generated by any commercial MBSD framework (PTC, Sparx-EA, IBM-Rapsody, ...) but a *human-oriented, vendor-independent* textual description of the system[18].

"Inadequacy of existing MBSD frameworks to support formal analysis of system requirements"
With the experience gained so far in the demonstrator, the role of the selected Sparx-EA MBSD platform is limited to providing some help in the generation of readable, well-formatted documentation and (sometimes) in creating executable system designs that satisfy a first pass of static analysis. From the point of view of usability towards modelling high-level requirements and performing a rigorous analysis or verification of them, this MBSD platform, despite its animation

[18] The problems with XMI are twofold: 1) it is apparently a standard format, while in practice makes impossible the migration of models among different frameworks, with our first-hand experience in import/export XMI from Sparx EA to Cameo Modeling tool, and 2) it is not a human oriented format usable to directly communicate in a simple, textual, easily reusable way a model design.

capabilities, has resulted rather useless as it does not allow to explore all the theoretically allowed system evolutions. The situation is likely to be the same with other platforms like Magic Draw or PTC, until eventually all these platforms are enriched with mechanical facilities to translate UML model designs into formal notations.

"Semi-formal and formal methods can be exploited at many levels of detail and with different degrees of effort"
We have observed that many of the weaknesses present in a plain natural language system requirements document, mostly related to ambiguity or imprecision of the requirements, can already be revealed during the initial attempt to generate an *operational model* of the system. The formal modelling and analysis steps greatly improve the depth of analysis on the system, allowing to discover further hidden design defects potentially leading to non-uniformity of implementations and interoperability problems. Formal analysis can be done with different levels of effort resulting in different levels of confidence about the correctness of the design.
The three verification platforms that have been used (UMC, Prob, CADP) are frameworks upon which we already had some experience in other projects and in our professional careers. Nevertheless, for lack of time and experience, only a small fraction of the features made available by these frameworks have been exploited. Becoming an *expert* in the use of any formal method and its supporting framework is a task that goes much further than simply being able to obtain some results with it. However, it is not mandatory to be a real expert to (partially) benefit from the gains that formal methods can give. While it is recognized the difficulty (and error proneness) of translating requirements and properties into temporal logics properties for formal verification, there are semi-automatic verification approaches dealing with deadlocks, coverage, consistency checking, absence of runtime-errors, invariants preservation, abstraction of the system behavior at the interfaces, that may greatly improve the confidence on the design with a relatively low effort.
In our case, the formalization/verification focus has been posed on three specific frameworks; however, these specific choices did not seem to play a particularly relevant role in the overall formal analysis process. Any other formal framework which guarantees an advanced level of static analysis, an interactive exploration method that allows experimenting *all* the possible system evolutions, and some kind of property verification strategy could easily replace/complement any of our adopted methods (SPIN, NuSMV, mCRL2, FDR4, just to mention some).

"Usefulness of formal methods diversity"
Another confirmation that we have had from our demonstrator process activity is that the "diversity of approaches" in formal modelling and verification improves the flexibility of the analysis and the reliability of results. Many errors in the translation programs have been quickly put in evidence when different behaviors and different statespaces resulted from the translation of the UML model into the ProB / LNT / UMC notations. Moreover, different points of view can be exploited with formal methods diversity in the analysis of the expected properties of the system under design.

"From natural language to formal models and back"

If the used UML features are appropriately constrained, it might also become possible to re-associate a rigorous, clear, well-structured natural language description[19] to the semi-formal and formal models of the systems. Such natural language description should communicate in a natural way to the developers the intended internal behavior of the system, the properties that each component is supposed to guarantee to the other ones, and the assumptions about the dependencies over the other components behavior.

"Formal methods are not a silver bullet: many difficulties still exist."
The introduction of formal methods in the system requirements specification phase still has to face several technical difficulties. Our case study has clearly put in evidence three main difficulties:

- Statespace explosion: This typically arises when we have to deal with the integration of different subsystems or with operations carrying wide range data.
- Parameterized specification: The adopted model-checking approach can only work on non-parametric systems. We had to explicitly define specific scenarios upon which to make the analysis, by setting specific values for the various parameters of the system components. But this analysis may not cover the full range of system configurations.
- Interfaces with wide-range data values: When a system is composed of subsystems interacting through messages containing data values (in our case, the CSL exchanging DATA_requests/ Data_indications), the benefits of a compositional approach may be severely endangered. The possible statespace describing of a CSL-standalone can be larger than the statespace of the integrated system where the CSL component is composed of specific (limited) data producers.

"The evidences from the demonstrator"
The evidences revealed by the application of our formal methods demonstrator process to our specific case study have clearly highlighted the potential advantages, in terms of requirements specification quality, gained with the - possibly lightweight - introduction of formal methods in the system requirements definition process[20].
However, the introduction of formal methods, in order to be fully exploited, would require additional support of formal methods from industry-ready MBSD frameworks, further efforts from the designers of formal verification tools to simplify the integration of their features in industrial settings, and further efforts from standardization entities like OMG for the rigorous definition of their specifications.

[19] like the one shown in Appendix C.
[20] see, e.g., the annotations to the D2.3 requirements in Appendix D, and the revised system specifications shown in Appendix B and C.

7 References

[ACTLX] R. De Nicola and F.W Vaandrager. Three Logics for Branching Bisimulation. Journal of the Association for Computing Machinery, 1990.

[BOX] Box, G. E. P. (1976), "Science and statistics" (PDF), Journal of the American Statistical Association, 71 (356): 791–799,

[CADP] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT), 15(2):89–107, April 2013.

[E-LOTOS] International Standard ISO-15437:2001.

[EN50159] CENELEC-EN 50159:2018 “Railway applications – Communication, signalling and processing systems - Safety related electronic systems for signalling”.

[Compositional] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compositional Verification of Asynchronous Concurrent Systems Using CADP. Acta Informatica, 52(4):337–392, April 2015.

[Combining] Frédéric Lang, Radu Mateescu, and Franco Mazzanti: Compositional Verification of Concurrent Systems by Combining Bisimulations, in Formal Methods in System Design, Springer, 18 February 2021.

[D2.1] Deliverable 2.1 of Task 2.1 of 4SECURail project,
"Formal development demonstrator prototype 1st release",
https://projects.shift2rail.org/download.aspx?id=560cdd44-83e7-4f5d-879e-d8dcdf2e2b1b

[D2.2] Deliverable 2.3 of Task 2.2 of 4SECURail project,
"Formal development demonstrator prototype 1st release"
https://projects.shift2rail.org/download.aspx?id=1761f4fa-c701-4321-b40c-3e67146ed482

[D2.3] Deliverable 2.3 of Task 2.2 of 4SECURail project,
"Case study requirements and specification"
https://projects.shift2rail.org/download.aspx?id=6917d0da-122f-41cb-8194-5f3e5029516b

[DBR] Rob J. van Glabbeek and W. Peter Weijland. Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM, 43(3):555–600, 1996.

[EULYNX] The Eulynx project site. https://eulynx.eu/

[FMDR] F.Mazzanti, A.Ferrari, G.O. Spagnolo "Towards formal methods diversity in railways: an experience report with seven frameworks", International Journal on Software Tools for Technology Transfer (STTT) volume 20 2018,
https://link.springer.com/article/10.1007/s10009-018-0488-3

[FMRMAP] A. Ferrari and M.H. ter Beek, Formal Methods in Railways: a Systematic Mapping Study. arXiv:2107.05413 [cs.SE], 2021. https://arxiv.org/abs/2107.05413

[GRA] Graphviz - Graph Visualization Software, https://www.graphviz.org/

[LDBR] Radu Mateescu and Anton Wijs. Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. Science of Computer Programming, 96(3):354–376, 2014.

[LNT] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Fr ́ed ́eric Lang, Christine McKinty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding, "Reference Manual of the LNT to LOTOS Translator",
https://cadp.inria.fr/ftp/publications/cadp/Champelovier-Clerc-Garavel-et-al-10.pdf

[MAAP2019] Shift2Rail Multi-Annual Action Plan – Part B (2019) https://shift2rail.org/wp-content/uploads/2019/05/ /Draft-Shift2Rail-Multi-Annual-Action-Plan-Part-B-20.5.2019.pdf

[OMG-SysML] Object Management Group, "SysML 1.6 Specification", November 2019. http://www.omg.org/spec/SysML/1.6/

[OMG-UML] Object Management Group "Unified Modelling Language" version 2.5.1, December 2015, https://www.omg.org/spec/UML/About-UML/

[OMG-PSSM] Object Management Group "Precise Semantics of UML State Machine" version 1.0, May 2019, https://www.omg.org/spec/PSSM/1.0/About-PSSM/

[PlantUML] PlantUML website, https://www.plantuml.com (also https://www.planttext.com)

[PROB] ProB website, https://www3.hhu.de/stups/prob/

[SHARP] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. Sharp congruences Adequate with Temporal Logics Combining Weak and Strong Modalities. In Armin Biere and Dave Parker, editors, Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2020 (Dublin, Ireland), Lecture Notes in Computer Science. Springer, 2020.

[SPARX] Sparx Systems Enterprise Architect https://sparxsystems.com/products/ea/index.html

[STRONG] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, Theoretical Computer Science, volume 104 of Lecture Notes in Computer Science, pages 167–183. Springer, March 1981.

[SUB-037] UNISIG - "EuroRadio FIS "- SUBSET-037 - 15-12-2015 (Issue 3.2.0)

[SUB-039] UNISIG - "FIS for the RBC/RBC Handover "- SUBSET-039 - 17-12-2015 (Issue 3.2.0)

[SUB-098] UNISIG - "RBC/RBC Safe Communication Interface" - SUBSET-098 - 21-05-2007

[SVL] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea, pages 377– 392. Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[UMC1] KandISTI project website http://fmt.isti.cnr.it/kandisti

[UMC2] UMC project website http://fmt.isti.cnr.it/umc

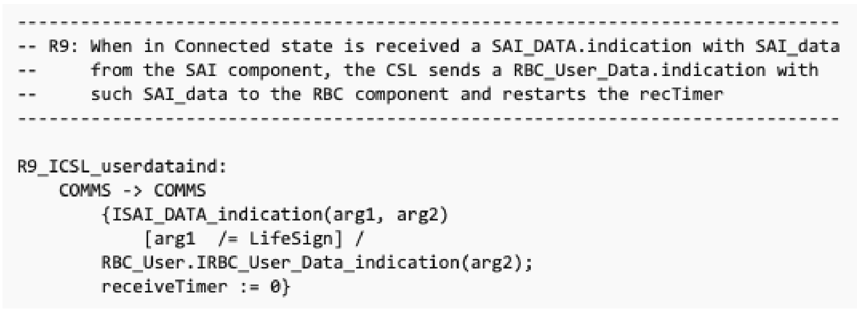
[X2RAIL2] X2Rail2 project website https://projects.shift2rail.org/s2r_ip2_n.aspx?p=X2RAIL-2

[ZenodoWP2] http://doi.org/10.5281/zenodo.4280773

Appendix A: Formal notations and transformations

UMC modelling

The main characteristic of UMC is that a simple textual notation is used to specify the state transitions of a UML state machine. In Figure A.1 one of these transitions is depicted.



```
--------------------------------------------------------------------------------
-- R9: When in Connected state is received a SAI_DATA.indication with SAI_data
--     from the SAI component, the CSL sends a RBC_User_Data.indication with
--     such SAI_data to the RBC component and restarts the recTimer
--------------------------------------------------------------------------------

R9_ICSL_userdataind:
    COMMS -> COMMS
        {ISAI_DATA_indication(arg1, arg2)
            [arg1  /= LifeSign] /
        RBC_User.IRBC_User_Data_indication(arg2);
        receiveTimer := 0}
```

Figure A.1 : Example of UMC rule

Each transition definition is defined by:

- an optional transition name (R9_ICSL_userdataind in Figure A.1),
- the source and target states of the transition (COMMS, COMMS in Figure A.1),
- A block { } containing:
    - the *triggering* event of the transition (ISAI_DATA_indication in Figure A.1), possibly with parameters and guards,
    - the sequence of actions to be performed as an *effect* of the transition (the sending of the IRBC_User_Data_indication signal to the RBC User component and the assignment to the receiveTimer variables in Figure A.1).

The names that appear inside a transition definition can refer to names of the other components constituting the system, the possible parameter of the triggering event, and to local variables of the state machine.

With UMC it is possible to check if/how a given transition is eventually fired, if/when a certain signal is sent, if/when a certain variable is modified, or a certain state reached.

For example, we can ask for an explanation about when the transition shown in Figure A.1 is fired (i.e., when it happens that the signal SAI_CONNECT_confirm causes the sending of the RBC_User_Connect_indication), and the answer can be observed in term of a sequence diagram, as shown in Figure A.2 (the graphical layout is automatically generated by UMC using the PlantUML online services [PlantUML].

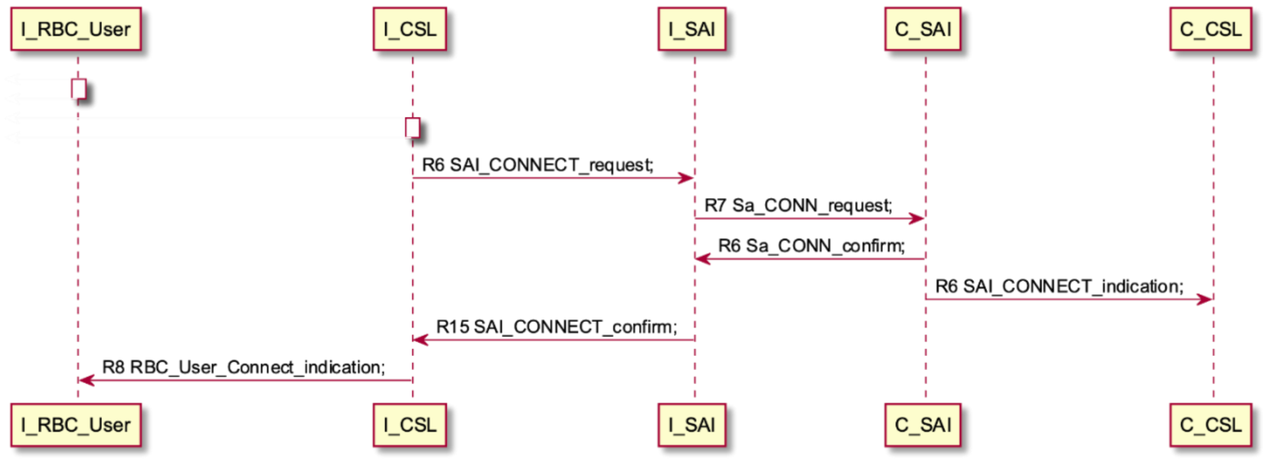


Figure A.2: A sequence diagram generated with UMC

The complete set of UMC models and some related audio-visual material can be retrieved from the public Zenodo repository [ZenodoWP2].

Several programs have been developed to facilitate the integration of the various tools used in the demonstrator. In Section 5.1.5 we have already mentioned the translator that mechanizes the transformation of UMC [UMC1, UMC2] models into ProB [PROB] and LNT [LNT] models.

ProB modelling

A system specification is structured in ProB as a "B machine". In our case, since the system under analysis is composed of several mutually interacting state machines (and the B language is not able to deal with this concept), we need to "merge" all these components into a unique, global state machine. This has several implications:

- The separate class attributes of UML state machines must be merged into a single B state machine. This may require the prefixing of the variable names with the component names to avoid name clashes (e.g., while in UML we have the I_CSL and C_CSL classes making use of their own "send_timer" attribute, in B we will have the two attributes "icsl_send_timer" and "ccsl_send_timer"). The same needs to be done for the operation names (transition labels in UMC) and the other entities that require duplication.

- The currently active state of a UML state machine is represented in B by the current value of an ad-hoc variable *statemachine_STATUS*. There is one such variable for each UML state machine.

- Within the B machine structure, all types, constants, and variable definitions and initializations must appear at the beginning of the machine definition. This disrupts the original structure of the system forcing to spread the UML state machine definition into several places in the B machine specification.

- In UML State Machines the event pool (a buffer implementing asynchronous communications that contains at each moment the set of signals arrived in a state machine but not yet dispatched or discarded) is part of the engine support and thus is not explicitly modelled. In B these event-pool components must be explicitly modelled. This is because, contrary to UMC, B is not a tool designed for handling UML State Machines. Our UML/UMC/ProB assumption is that these pools are instantiated as First-In First-Out (FIFO) queues (this is the default implementation suggested by UML standard), therefore a "buffer" variable representing the state machine event pool is added to the B model. Consequently, the action of sending a signal to another state machine will be modelled with the insertion of a value to the corresponding variable buffer, and the dispatching of a signal to trigger a transition will be modelled with the extraction of the first element of such a buffer.

- Each transition rule definition of the UMC state machine design is mapped onto an equivalent operation of the B machine.
  This mapping is at this point very direct as shown below:

```
UMC transition
```

```
R4_ICSL_userconnind
NOCOMMSconnecting -> COMMS
 { ISAI_CONNECT_confirm /
   RBC_User.IRBC_User_Connect_indication;
   connect_timer := max_connect_timer;
   receive_timer := 0;
   send_timer := 0; }
```

```
B machine operation
```

```
R4_ICSL_userconnind =
 PRE
   ICSL_buff /= [] &
   first(ICSL_buff) = ISAI_CONNECT_confirm &
   ICSL_STATE = ICSL_NOCOMMSconnecting
 THEN
  IRBC_buff := IRBC_buff <- IRBC_User_Connect_indication;
  ICSL_connect_timer := ICSL_max_connect_timer;
  ICSL_receive_timer := 0;
  ICSL_send_timer := 0;
  ICSL_buff := tail(ICSL_buff);
  ICSL_STATE := ICSL_COMMS
END;
```

<u>**LNT modelling**</u>

LNT [LNT] is one of the formal notations accepted by the CADP [CADP] verification framework. The notation is a simplified variant of E-LOTOS [E-LOTOS], of which preserves the expressiveness but adopting a more user-friendly and regular notations borrowed from imperative and functional programming languages. A system is described in LNT as a parallel composition of (parametric) processes, which synchronize upon a statically defined set of gates. A process can have a local set of variables on which can operate with classical imperative instructions and statements.
The global environment is constituted by the data types and functions used by the processes.
A LNT specification is internally translated into the LOTOS algebraic notation and can be analyzed using the CADP toolbox.

The schema adopted for the translation into the LNT language (*umc2lnt*) is instead quite different from the one adopted in the ProB case. This time each UMC state machine is associated with an independent LNT process. All the processes do not share any memory and interact through

synchronous actions in the typical style of process algebras. Each process handles a local event pool modelled as a FIFO buffer and is always available to accept synchronizations from other processes willing to push a new message. Beyond accepting incoming messages, the LNT process can internally evolve, performing internal steps that transform the local status of synchronizing with other processes when sending messages towards other state machines.
The final system is finally obtained by composing in parallel all the processes which synchronize the corresponding actions of sending and receiving a message.
The code below shows a sample fragment of the LNT transformation.

UMC state machine

```
Class ICSL is
  ...
  ...
Behavior
   ...
R4_ICSL_userconnind
NOCOMMSconnecting -> COMMS
 { SAI_CONNECT_confirm /
   RBC_User.IRBC_User_Connect_indication;
   connect_timer := max_connect_timer;
   receive_timer := 0;
   send_timer := 0; }
   ...
end;
```

LNT process

```
process ICSL [..]  is
        ...
var mybuff: ICSL_BUFF, ...  in
  loop
    select
       -- R4_ICSL_userconnind
       only if
         mybuff /= nil and
         head(mybuff) = ISAI_CONNECT_confirm and
         STATE = NOCOMMSconnecting
       then
         IRBC_User_Connect_indication;
         connect_timer := ICSL_max_connect_timer;
         receive_timer := 0;
         send_timer := 0;
          mybuff := tail(mybuff);
         STATE := COMMS
       end if
    []
       ...
    end select
  end loop
end var
end process
```

Translation Tools
It is outside of the project goals the generic implementation of translators for full UML (or full UMC subset). For the purpose of this project our goal is limited to the translation of the set of features used in our models. This initial approach may constitute the base for further developments.
Due to the drastic simplifications which have been made in defining the subset of features to be used in the initial UML designs (e.g., no composite states, no parallel states, no deferred events, no competition between triggered and completion transitions), the final effect of the transformations is the generation of formal models which have almost the same readability than the original UML model; this is helped by the fact that also the original comments present in the UMC code are preserved in the generated ProB and LNT encodings.

The transformation of UMC models into ProB and LNT models are not the only programs that have been developed. In order to compare and reason upon the formal semantics of the generated formal models, several other translators have been considered useful. There is, in particular, an explicit format of Labellel Transition Systems (LTS) that fits well the need of cross-platform analysis: this is the *.aut* format, invented at INRIA (FR) and widely recognized by several frameworks.
The KandISTI/UMC framework allows to save the statespace of a model in the .aut format, and the

same occurs in the CADP[CADP] framework for the LNT language. What was missing is just the possibility to save the Prob statespace of a system model in the same *.aut* format. Since ProB already allows to save the model statespace in a simple textual format, we have developed a *probspace2aut* program that just transforms that native Prob statespace in the *.aut* format.
These three translations have allowed us, starting from an initial UMC model, to compare the statespaces of the UMC, ProB, LNT formal models and formally verify their equivalence.

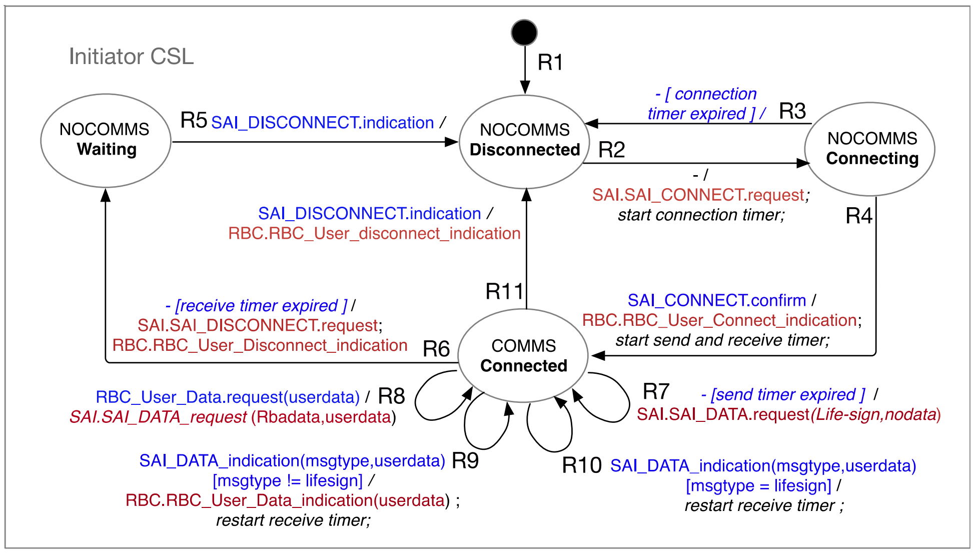
Several other auxiliary tools, still operating on the .aut format have been developed to support the formal analysis process. For instance,
- *aut2fmc* -- - transformation of explicit LTS statespace into code for KandISTI/FMC model checker
- *plainaut2dot* -- graphical visualization of LTS with the .dot Graphvix [GRA] notation.
- *wtprepare* -- transformation of explicit LTS with the identification of deadlocks and infinite loops of non-observable actions.

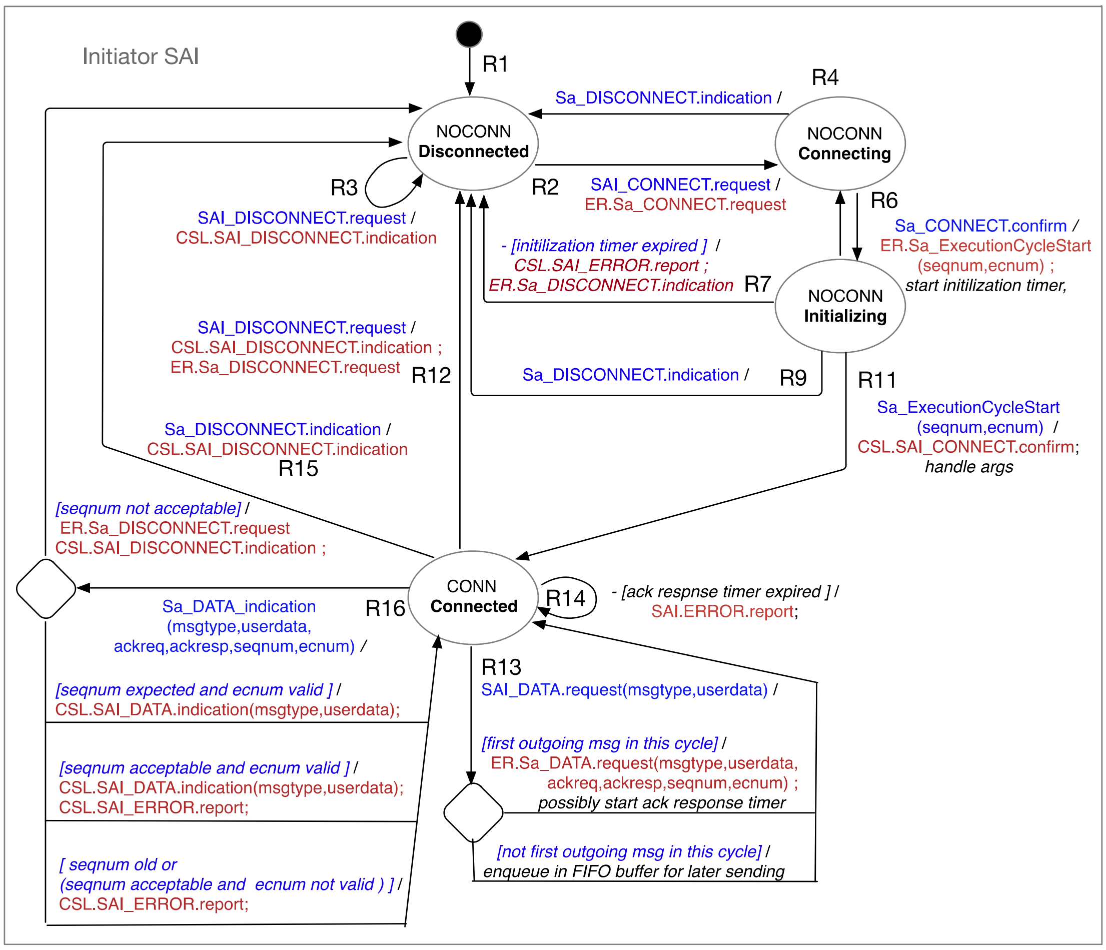
These tools complement the already mentioned:
- *umc2prob*
- *umc2lnt*
- *procstatespace2aut*
and the *probtrace2sd* tool (mentioned in D2.2) that can be used to display a ProB history trace in the more user-friendly form of a message sequence diagram.

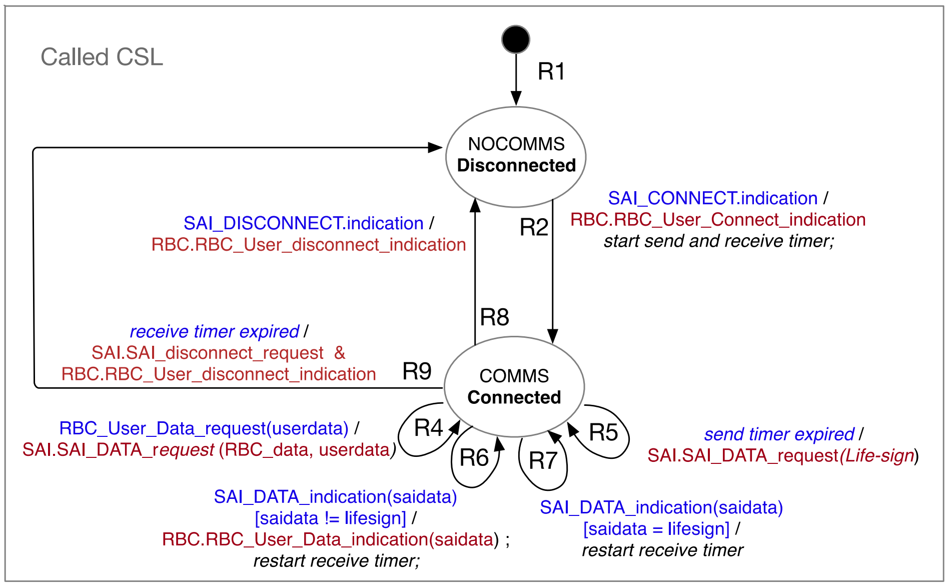
All these tools will be freely available, open-source, and retrievable from the Zenodo [ZenodoWP2] repository containing all the WP2 complementary material (including all the developed models in the various notations).

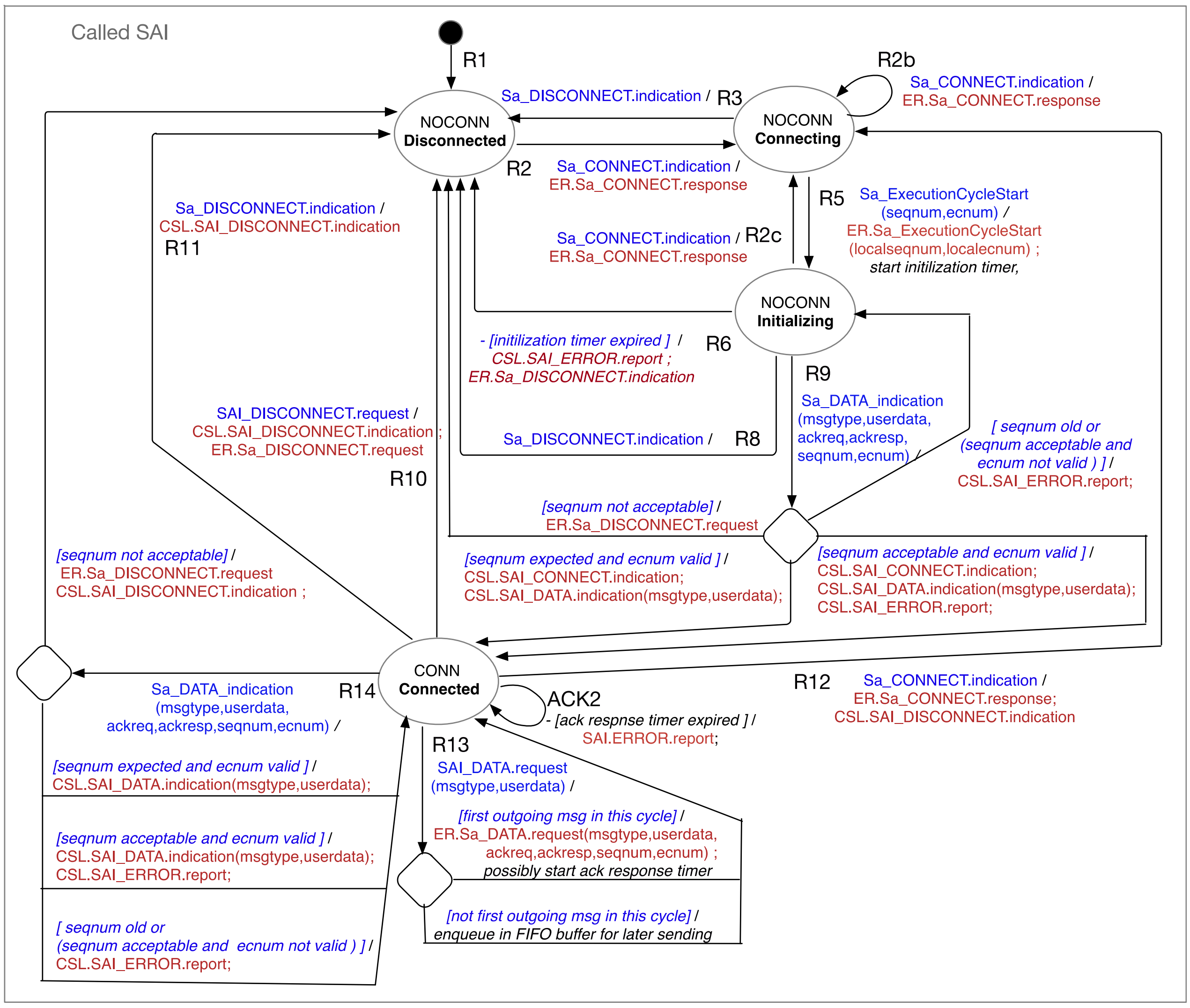
Horizon 2020
European Union Funding
for Research & Innovation

Appendix B: Graphical semiformal UML state machine designs


Initiator CSL
R1
NOCOMMS Waiting
R5 SAI_DISCONNECT.indication /
NOCOMMS Disconnected
- [connection timer expired] / R3
NOCOMMS Connecting
R2
- /
SAI.SAI_CONNECT.request;
start connection timer;
R4
SAI_DISCONNECT.indication /
RBC.RBC_User_disconnect_indication
R11
- [receive timer expired] /
SAI.SAI_DISCONNECT.request;
RBC.RBC_User_Disconnect_indication
R6
COMMS Connected
SAI_CONNECT.confirm /
RBC.RBC_User_Connect_indication;
start send and receive timer;
RBC_User_Data.request(userdata) / R8
SAI.SAI_DATA_request (Rbadata,userdata)
R7 - [send timer expired] /
SAI.SAI_DATA.request(Life-sign,nodata)
SAI_DATA_indication(msgtype,userdata) R9
[msgtype != lifesign] /
RBC.RBC_User_Data_indication(userdata) ;
restart receive timer;
R10 SAI_DATA_indication(msgtype,userdata)
[msgtype = lifesign] /
restart receive timer ;


Initiator SAI
R1
R4
Sa_DISCONNECT.indication /
NOCONN Disconnected
NOCONN Connecting
R3
R2
SAI_CONNECT.request /
ER.Sa_CONNECT.request
R6
SAI_DISCONNECT.request /
CSL.SAI_DISCONNECT.indication
- [initilization timer expired] /
CSL.SAI_ERROR.report ;
ER.Sa_DISCONNECT.indication R7
Sa_CONNECT.confirm /
ER.Sa_ExecutionCycleStart
(seqnum,ecnum) ;
start initilization timer;
NOCONN Initializing
SAI_DISCONNECT.request /
CSL.SAI_DISCONNECT.indication ;
ER.Sa_DISCONNECT.request R12
Sa_DISCONNECT.indication / R9
R11
Sa_ExecutionCycleStart
(seqnum,ecnum) /
CSL.SAI_CONNECT.confirm;
handle args
Sa_DISCONNECT.indication /
CSL.SAI_DISCONNECT.indication
R15
[seqnum not acceptable] /
ER.Sa_DISCONNECT.request
CSL.SAI_DISCONNECT.indication ;
CONN Connected
R14 - [ack respnse timer expired] /
SAI.ERROR.report;
Sa_DATA_indication
(msgtype,userdata,
ackreq,ackresp,seqnum,ecnum) /
R16
R13
SAI_DATA.request(msgtype,userdata) /
[seqnum expected and ecnum valid] /
CSL.SAI_DATA.indication(msgtype,userdata);
[first outgoing msg in this cycle] /
ER.Sa_DATA.request(msgtype,userdata,
ackreq,ackresp,seqnum,ecnum) ;
possibly start ack response timer
[seqnum acceptable and ecnum valid] /
CSL.SAI_DATA.indication(msgtype,userdata);
CSL.SAI_ERROR.report;
[not first outgoing msg in this cycle] /
enqueue in FIFO buffer for later sending
[seqnum old or
(seqnum acceptable and ecnum not valid)] /
CSL.SAI_ERROR.report;


Called CSL
R1
NOCOMMS Disconnected
SAI_CONNECT.indication / RBC.RBC_User_Connect_indication start send and receive timer;
R2
SAI_DISCONNECT.indication / RBC.RBC_User_disconnect_indication
R8
receive timer expired / SAI.SAI_disconnect_request & RBC.RBC_User_disconnect_indication
R9
COMMS Connected
RBC_User_Data_request(userdata) / SAI.SAI_DATA_request (RBC_data, userdata)
R4
R6
R7
R5
send timer expired / SAI.SAI_DATA_request(Life-sign)
SAI_DATA_indication(saidata) [saidata != lifesign] / RBC.RBC_User_Data_indication(saidata) ; restart receive timer;
SAI_DATA_indication(saidata) [saidata = lifesign] / restart receive timer


Called SAI
R1
R2b
Sa_CONNECT.indication / ER.Sa_CONNECT.response
Sa_DISCONNECT.indication / R3
NOCONN Disconnected
NOCONN Connecting
R2
Sa_CONNECT.indication / ER.Sa_CONNECT.response
Sa_DISCONNECT.indication / CSL.SAI_DISCONNECT.indication
R11
R5
Sa_ExecutionCycleStart (seqnum,ecnum) / ER.Sa_ExecutionCycleStart (localseqnum,localecnum) ; start initilization timer,
Sa_CONNECT.indication / R2c ER.Sa_CONNECT.response
NOCONN Initializing
- [initilization timer expired] / CSL.SAI_ERROR.report ; ER.Sa_DISCONNECT.indication
R6
R9
Sa_DATA_indication (msgtype,userdata, ackreq,ackresp, seqnum,ecnum) /
[seqnum old or (seqnum acceptable and ecnum not valid)] / CSL.SAI_ERROR.report;
SAI_DISCONNECT.request / CSL.SAI_DISCONNECT.indication ; ER.Sa_DISCONNECT.request
R10
Sa_DISCONNECT.indication / R8
[seqnum not acceptable] / ER.Sa_DISCONNECT.request
[seqnum not acceptable] / ER.Sa_DISCONNECT.request CSL.SAI_DISCONNECT.indication ;
[seqnum expected and ecnum valid] / CSL.SAI_CONNECT.indication; CSL.SAI_DATA.indication(msgtype,userdata);
[seqnum acceptable and ecnum valid] / CSL.SAI_CONNECT.indication; CSL.SAI_DATA.indication(msgtype,userdata); CSL.SAI_ERROR.report;
CONN Connected
Sa_DATA_indication (msgtype,userdata, ackreq,ackresp,seqnum,ecnum) /
R14
ACK2
- [ack respnse timer expired] / SAI.ERROR.report;
R12
Sa_CONNECT.indication / ER.Sa_CONNECT.response; CSL.SAI_DISCONNECT.indication
R13
SAI_DATA.request (msgtype,userdata) /
[seqnum expected and ecnum valid] / CSL.SAI_DATA.indication(msgtype,userdata);
[first outgoing msg in this cycle] / ER.Sa_DATA.request(msgtype,userdata, ackreq,ackresp,seqnum,ecnum) ; possibly start ack response timer
[seqnum acceptable and ecnum valid] / CSL.SAI_DATA.indication(msgtype,userdata); CSL.SAI_ERROR.report;
[not first outgoing msg in this cycle] / enqueue in FIFO buffer for later sending
[seqnum old or (seqnum acceptable and ecnum not valid)] / CSL.SAI_ERROR.report;

Appendix C: Structured Natural Language Requirement Specifications

Requirements Specification for the ***Initiator CSL*** Component

Configuration Parameters
System parameters,

- *connect_timer;*
- *send_timer;*
- *receive_timer.*

External Interactions
The *Initiator CSL* can receive from the *Initiator RBC* component the following message:

- *RBC_User_Data.request(RBC_data_value);*

and can send to the *RBC* component the following messages:

- *RBC_User_Connect.indication;*
- *RBC_User_Disconnect.indication;*
- *RBC_User_Data.indication(RBC_data_value).*

The *CSL* can receive from the *Initiator SAI* component the following messages:

- *SAI_CONNECT.confirm;*
- *SAI_DISCONNECT.indication;*
- *SAI_DATA.indication(message_type*[21]*, SAI_data_value);*
- *SAI_ERROR.report;*

and can send to the *SAI* component the following messages:

- *SAI_CONNECT.request;*
- *SAI_DISCONNECT.request;*
- *SAI_DATA.request(message_type, SAI_data_value).*

States
The *CSL* can be in the following four main states:

- *Disconnected*, when the communication is unactive;
- *Connecting*, when the communication is in the establishment phase;
- *Connected*, when the communication is active;
- *Waiting*, when the communication is between the *Connected* and *Disconnected* states.

External Guarantees

- *CSL* sends *RBC_User_Data.indication* messages only after an *RBC_User_Connect.request* not followed by *RBC_User_Disconnect.indication*;
- *CSL* sends to the *RBC* component an *RBC_User_Disconnect.indication* message only after an *RBC_User_Connect.request* message not already followed by *RBC_User_Disconnect.indication*;
- *CSL* sends to the *RBC* component an *RBC_User_Connect.indication* message only as first message or after an *RBC_User_Disconnect.indication* not already followed by *RBC_User_Connect.indication*;

[21] *message_type* may refer to either *life_sign* or *RBC_data.*

- the first message (possibly) sent to the *RBC* component is an *RBC_User_Connect.indication* message;
- the initiator *CSL* periodically sends to the *SAI* component either *SAI_CONNECT.request* or *SAI_DATA.request* messages;
- if the initiator *CSL*, while in *Connected* (*COMMS*) state, does not receive any *SAI_DATA.indication* message from the *SAI* for a certain specified amount of time, a *SAI_DISCONNECT.request* message is sent to the *SAI*;
- the initiator CSL may send a *SAI_DISCONNECT.request* message only when in *Connected* (*COMMS*) state;
- incoming messages are buffered and served with *FIFO* policy.

External Assumptions

- The *SAI* always replies with a *SAI_DISCONNECT.indication* message to *SAI_DISCONNECT.request* messages issued by the *CSL*.

Behavioral Requirements

R1: At startup, the *CSL* is in *Disconnected* state.

When in Disconnected State

R2: When in *Disconnected* state, the *CSL* immediately sends a *SAI_CONNECT.request* to the *SAI* component, starts a *connTimer*, and moves to the *Connecting* state.

When in Connecting State

R3: When in *Connecting* state the *connTimer* expires, the *CSL* moves to *Disconnected* state.

R4: When in *Connecting* state is received a *SAI_CONNECT.confirm* from the *SAI* component, the *CSL* sends an *RBC_User_Connect.indication* to the *RBC* component, starts both the *sendTimer* and the *recTimer*, and moves to *Connected* state. It is allowed to set the sendTimer so that an initial lifesign is sent without delay.

When in Waiting State

R5: When in *Waiting* state is received a *SAI_DISCONNECT.indication* from the *SAI* component, the *CSL* moves to *Disconnected* state.

When in Connected State

R6: When in *Connected* state the *recTimer* expires, the *CSL* sends a *SAI_DISCONNECT.request* to the *SAI* component, an *RBC_User_Disconnect.indication* to the *RBC* and moves to *Waiting* state.

R7: Each time that in *Connected* state the *sendTimer* expires, the *CSL* sends a *SAI_DATA.request* with a *life_sign* to the *SAI* component.

R8: When in *Connected* state is received an *RBC_User_Data.request* with *RBC_data* from the *RBC* component, the *CSL* sends a *SAI_DATA.request* with such *RBC_data* to the *SAI* component.

R9: When in *Connected* state is received a *SAI_DATA.indication* with *SAI_data* from the *SAI* component, the *CSL* sends an *RBC_User_Data.indication* with such *SAI_data* to the *RBC* component and restarts the *recTimer*.

R10: When in *Connected* state is received a *SAI_DATA.indication* with *a life_sign* from the *SAI* component, the *CSL* restarts the *recTimer*.

R11: When in *Connected* state is received a *SAI_DISCONNECT.indication* from the *SAI* component, the *CSL* sends an *RBC_User_Disconnect.indication* to the *RBC* component and moves to *Disconnected* state.

Discarding of Messages

RD1: When in *Connecting* state, the *CSL* discards any message except for *SAI_CONNECT.confirm* from the *SAI* component.

RD2: When in *Waiting* state, the *CSL* discards any message except for *SAI_DISCONNECT.indication* from the *SAI* component.

RD3: When in *Connected* state, the *CSL* component discards only *SAI_CONNECT.confirm* and *SAI_ERROR.report* messages from the *SAI* component.

Requirements Specification for the ***Initiator SAI*** Component

Configuration Parameters

Initialization kind: *Execution Cycle* option.

System parameters,

- for *Execution Cycle* procedure:
  - *maximum initialization delay*
  - *Mec* (limit of the execution cycle counters)*;*
  - *K* (max acceptable transmission delay for a message);
- for *ACK* procedure:
  - *ack_request_period;*
  - *ack_response_timeout;*
- for *sequence number*:
  - *N* (limit of acceptable, consecutive message losses, *N = 1* means no losses)*;*
  - *M* (limit of the sequence number values, which have range *0..M-1*).

External Interactions

The *Initiator SAI* can receive from the *Initiator CSL* component the following messages:

- *SAI_CONNECT.request*;
- *SAI_DISCONNECT.request*;
- *SAI_DATA.request (message_type*[22]*, RBC_data_value)*;

and can send to the *CSL* component the following messages:

- *SAI_CONNECT.confirm*;
- *SAI_DISCONNECT.indication*;
- *SAI_DATA.indication(message_type, RBC_data_value)*;
- *SAI_ERROR.report*.

The *SAI* can receive from the *EuroRadio* Safety Layer (henceforth *ER-SL*) the following messages:

- *Sa_CONNECT.confirm*;
- *Sa_DISCONNECT.indication*;
- *Sa_DATA.indication(message_type, data_value, ack_request, ack_response, sequence_number, execution_cycle_number)*;
- *Sa_ExecutionCycleStart(sequence_number, execution_cycle_counter)*;

and can send to the *ER-SL* the following messages:

- *Sa_CONNECT.request*;
- *Sa_DISCONNECT.request*;
- *Sa_DATA.request(message_type, data_value, ack_request, ack_response, sequence_number, execution_cycle_number)*;
- *Sa_ExecutionCycle(sequence_number, execution_cycle_counter)*.

Internal Variables

- *sequence_number;*
- *execution_cycle_counter;*

[22] *message_type* may refer to either *life_sign* or *RBC_data.*

- *last_received_sequence_number;*
- *last_received_execution_cycle_counter;*
- *execution_cycle_OFFSET.*

States

The *SAI* can be in the following four main states:

- *Connected*, when the communication is active;
- *Connecting*, when the communication is in the establishment phase;
- *Initializing*, while performing the execution cycle start procedure;
- *Disconnected*, when the communication is unactive.

External Guarantees

- The *SAI* always replies with a *SAI_DISCONNECT.indication* message to *SAI_DISCONNECT.request* messages issued by the *CSL*;
- the data messages delivered to the *CSL* are valid (i.e., arrived with a limited delay), not duplicated, not reordered messages;
- no more than one data message per execution cycle is sent to the *ER-SL*;
- incoming messages are buffered and served with *FIFO* policy.

External Assuptions

- The *ER-SL* always eventually replies either with a *Sa_DISCONNECT.indication* or with a *SAI_CONNECT.confirm* to *Sa_CONNECT.request* messages issued by the SAI;
- the initiator CSL, after having sent a *SAI_CONNECT.request* message to the SAI, does not send a *SAI_DISCONNECT.request* message until *SAI_CONNECT.indication* messages is received.

Behavioral Requirements

R1: At startup, the *SAI* is in *Disconnected* state.

When in Disconnected State

R2: When in *Disconnected* state is received a *SAI_CONNECT.request* from the *CSL* component, the *SAI* sends a *Sa_CONNECT.request* to the *ER-SL* and moves to *Connecting* state.

R3: When in *Disconnected* state is received a *SAI_DISCONNECT.request* from the *CSL* component, the *SAI* replies with a *SAI_DISCONNECT.indication* to the *CSL* component.

When in Connecting State

R4: When in *Connecting* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.

R6: When in *Connecting* state is received a *Sa_CONNECT.confirm* from the *ER-SL*, the *SAI* replies with a *Sa_ExecutionCycle(seqnum, ecnum)* to the *ER-SL* and moves to the *Initializing* state, waiting for a *Sa_ExecutionCycleStart* message from the *ER-SL* within a *maximum initialization delay*. The management of the *Sa_ExecutionCycleStart* parameters is done according to the requirements in the following *Sequence Numbers Management* and *Execution Cycle Counters Management* sections.

When in Initializing State

R7: When in *Initializing* state the *maximum initialization delay* expires, the *SAI* sends an *SAI_ERROR.report* to the *CSL* component, a *Sa_DISCONNECT.request* to the *ER-SL* and moves to *Disconnected* state.

R9: When in *Initializing* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.

R11: When in *Initializing* state is received a *Sa_ExecutionCycleStart(seqnum, ecnum)* from the *ER-SL*, the *SAI* sends a *SAI_CONNECT.confirm* to the *CSL* component and moves to *Connected* state. The received *seqnum* is accepted as initial remote sequence number and the *ecnum* is accepted as initial value of the remote execution cycle counter. The *execution_cycle_OFFSET* variable is set as the difference between the current execution cycle counter and the received execution cycle counter. While the *last_received_sequence_number* variable is set to the received sequence number.

When in Connected State

R12: When in *Connected* state is received a *SAI_DISCONNECT.request* from the *CSL* component, the *SAI* replies with a *SAI_DISCONNECT.indication* to the *CSL* component, sends a *Sa_DISCONNECT.request* to the *ER-SL*, and moves to *Disconnected* state.

R13a: When in *Connected* state is received a *SAI_DATA_request(msgtype, data)* from the *CSL* component, and yet no other data message has been sent in this cycle, the *SAI* sends a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* to the *ER-SL*.
The *ackreq* and *ackresp parameters are set according to* REQ_ACKs.
The *seqnum* parameter is set according to SEQ_NUMs and the *ecnum* parameter is set according to REQ_ECNUMs.

R13b: When in *Connected* state is received a *SAI_DATA.request(msgtype, data)* from the *CSL* component, but another data message has already been sent in this cycle, the *SAI_DATA.request is saved in* a *FIFO dataout buffer (see also REQ_OUTDATABUFF).*

R14: Each time that in *Connected* state the *set_ack_response* expires, the *SAI* sends a *SAI_ERROR.report* to the *CSL* component.

R15: When in *Connected* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* sends a *SAI_DISCONNECT.indication* to the *CSL* component and moves to *Disconnected* state.

R16: When in *Connected* state is received a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *ER-SL* we can have four cases, depending on the received *seqnum* and *ecnum* values (see SEQ_NUMs and REC_ECNUMs Management).
***** The *seqnum* is the one EXPECTED and *ecnum* is VALID: In this case the *SAI* sends *a SAI_DATA.indication(msgtype, data)* to the *CSL* component.
***** The *seqnum* is ACCEPTABLE and the *ecnum* is VALID: in this case the *SAI* sends a *SAI_DATA.indication(msgtype, data)* and a *SAI_ERROR.report* to the *CSL* component.

* The *seqnum* is OLD or (the *seqnum* is ACCEPTABLE and the *ecnum* is VALID): In this case the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and discards the *Sa_DATA.indication* message.
*   The *seqnum* is NOT_ACCEPTABLE: In this case the *SAI* component sends a *Sa_DISCONNECT.request* to ER-SL and a *SAI_DISCONNECT.indication* to the *CSL* component, and then moves to *Disconnected* state.

OUTDATA Buffer Management

REQ_OUTDATABUFF1: At the beginning of each cycle, if the *dataout buffer* is not empty, the first *SAI_DATA.request(msgtype, data) in* the queue is removed and its data are used to send a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* to the *ER-SL.*
The ackreq, ackresp, seqnum, ecnum parameters are set according to *REQ_ECNUM, REQ_ACK*, and *REQ SEQNUM* requirements.

REQ_OUTDATABUFF2: When the *SAI* moves from the *Connected* state to the *Disconnected* state, the *dataout buffer is* emptied and the possibly waiting messages are discarded.

Execution Cycle Counters Management

REQ_ECNUM1: When entering in the *Initializing* state, the initial value of the *execution cycle counter* is set to 0.

REQ_ECNUM2: While in the *Initializing* or *Connected* state, the *execution cycle counter* is incremented modulo Mec at every cycle.

REQ_ECNUM3: When sending a *Sa_ExecutionCycleStart(seqnum,ecnum)* message or a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* the value of the *ecnum* parameter is set to the current value of the *execution cycle counter*.

REQ_ECNUM4: When receiving a *Sa_ExecutionCycleStart(seqnum,ecnum)* message from the *ER-SL,* the value of the *ecnum* parameter is used to compute the *EC_OFFSET* as difference between the current value of the *execution cycle counter and* the received *seqnum* value.

REQ_ECNUM5: When receiving a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message from *ER-SL*, the message in considered VALID if the message delay is less than *K*, where the message delay is computed as follows[23]:

```
message_delay = (execution_cycle_counter - EC_OFFSET) mod Mec[24]) - ecnum;
if message_delay < -Mec/2 then
        message_delay ≔  message_delay + Mec;
 elsif message_delay > Mec/2 then
        message_delay ≔   message_delay - Mec;
end if
```

[23] This is a simplification from what required by UNISIG-098 as we assume that the EC period is 1 cycle for both *SAI* sides.
[24] Also when applied to negative numbers, (N mod M) is assumed to be equal to ((N+M) mod M).

<u>*Sequence Numbers Management*</u>

SEQ_NUM1: When entering in the state *Connected*, the *sequence_number* is set to 0.

SEQ_NUM2: When in *Connecting* state a *Sa_ExecutionCycleStart(seqnum,ecnum)* message is sent to the *ER-SL,* the *seqnum* parameter is set to the current value of *sequence_number*.

SEQ_NUM3: When in the *Initializing* or *Connected* state a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message is sent to the *ER-SL, the seqnum* parameter is set to the current value of *sequence_number*, and the *sequence_number* is incremented by *1 mod M*.

SEQ_NUM4: When in *Initializing* state is received a *Sa_ExecutionCycleStart(seqnum,ecnum)* message from *ER-SL*, the value of the *seqnum* parameter is saved as *last_received_sequence_number*.

SEQ_NUM5: When in the *Initializing* or *Connected* state is received a *Sa_DATA.indication (msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *ER-SL*, the distance of the current message from the last received one is computed as follows*:*

```
distance := seq_num - last_received_sequence_number;
if  (distance <  -M/2) then {distance :=  distance + M};
else if  (distance  >  M/2) then {distance := distance - M};
```

If the distance value is equal to 1, the *seqnum* is considered EXPECTED.
If the distance value is lower than 1, the *seqnum* is considered OLD.
If the distance value is greater than 1 and less or equals to *N*, the *seqnum* is considered ACCEPTABLE.
If the distance value is greater than *N*, the *seqnum* is considered NOT_ACCEPTABLE.

<u>*ACK Management*</u>

REQ_ACK1: When in *Connected* state, the *SAI* periodically (with a configurable *ack_request_period*) sets an *ackreq* flag to the first *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message to be forwarded to the *ER-SL* and starts an *ack_response_timer* with a *max_response_delay* limit.
The *ackreq* flag is not set and the timer is not started if the *SAI* is still waiting for the response to a previous ack request.

REQ_ACK2: When the *ack_response_timeout* expires, if a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message with an *ackresp* parameter set has not yet been received from the *ER-SL*, the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and restarts the *ack request timer*.

REQ _ACK3: While in *Connected* or *Initializing* state, when it is received a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *CSL* component, the *SAI* sets the *ackresp* parameter in next *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message to be sent to the *ER-SL*.

<u>*Discarding of Messages*</u>

RD1: When in *Disconnected* state the *SAI* discards any message except for,

- *SAI_CONNECT.request* and *SAI_DISCONNECT.request* from the *CSL* component.

RD2: When in *Connecting* state, the *SAI* discards any message except for,

- *Sa_DISCONNECT.indication*, and *Sa_CONNECT.confirm* from the *ER-SL*;

RD3: When in *Initializing* state, the *SAI* discards any message except for,

- *Sa_DISCONNECT.indication and Sa_ExecutionCycleStart* from the *ER-SL*;

RD4: When in *Connected* state, the *SAI* discards any message except for,

- *Sa_DISCONNECT.indication* and *Sa_DATA.indication* from the *ER-SL*;
- *SAI_DISCONNECT.request, SAI_DATA.request* from the *CSL* component.

Requirements Specification for the ***Called CSL*** Component

Configuration Parameters

System parameters,

- *send_timer;*
- *receive_timer.*

External Interactions

The *Called CSL* can receive from the *Called RBC* component the following message:

- *RBC_User_Data.request(RBC_data_value);*

and can send to the *RBC* component the following messages:

- *RBC_User_Connect.indication;*
- *RBC_User_Disconnect.indication;*
- *RBC_User_Data.indication(RBC_data_value).*

The *CSL* can receive from the *Called SAI* component the following messages:

- *SAI_CONNECT.indication;*
- *SAI_DISCONNECT.indication;*
- *SAI_DATA.indication(message_type*[25]*, SAI_data_value);*
- *SAI_ERROR.report;*

and can send to the *SAI* component the following messages:

- *SAI_CONNECT.request;*
- *SAI_DISCONNECT.request;*
- *SAI_DATA.request(message_type, SAI_data_value).*

States

The *CSL* can be in the following two states:

- *Disconnected (NOCOMMS)*, when the communication is unactive;
- *Connected (COMMS)*, when the communication is active.

External Guarantees

- The frequency of messages being sent by *CSL* to *RBC* is limited by an upper bound;
- the frequency of messages being sent by *CSL* to *SAI* is limited by an upper bound;
- *CSL* sends *RBC_User_Data.indication* messages only after an *RBC_User_Connect.request* not followed by *RBC_User_Disconnect.indication*;
- *CSL* sends to the *RBC* component an *RBC_User_Disconnect.indication* message only after an *RBC_User_Connect.indication* message not already followed by *RBC_User_Disconnect.indication*;
- the first message (possibly) sent to the *RBC* component is an *RBC_User_Connect.indication* message;
- *CSL* sends to the *RBC* component an *RBC_User_Connect.indication* message only as first message or after an *RBC_User_Disconnect.indication* not already followed by *RBC_User_Connect.indication*;
- the called CSL, while in the *Connected* (*COMMS*) state periodically sends to the *SAI* component *SAI_DATA.request* messages;

[25] *message_type* may refer to either *life_sign* or *RBC_data.*

- if the called CSL, while in the *Connected* (*COMMS*) state, does not receive any *SAI_DATA.indication* message from the SAI for a certain specified amount of time, a *SAI_DISCONNECT.request* message is sent to the SAI;
- incoming messages are buffered and served with *FIFO* policy.

Behavioral Requirements

R1: At startup, the *CSL* is in *Disconnected* state.

When in Disconnected State

R2: When in *Disconnected* state is received a *SAI_CONNECT.indication* from the *SAI* component, the *CSL* sends an *RBC_User_Connect_indication* to the *RBC* component*,* starts both the *sendTimer* and the *recTimer*, and moves to *Connected* state. It is allowed to set the sendTimer so that an initial lifesign is sent without delay.

When in Connected State

R4: When in *Connected* state is received an *RBC_User_Data.request*(*userdata*) from the *RBC* component, the *CSL* sends a *SAI_DATA.request(RBC_data,userdata*) to the *SAI* component.

R5: Each time that in *Connected* state the *sendTimer* expires, the *CSL* sends a *SAI_DATA.request* with a *life_sign* to the *SAI* component.

R6: When in *Connected* state is received a *SAI_DATA.indication* with *a life_sign* from the *SAI* component, the *CSL* restarts the *recTimer*.

R7: When in *Connected* state is received a *SAI_DATA.indication* with *SAI_data* from the *SAI* component, the *CSL* sends an *RBC_User_Data.indication* with such *SAI_data* to the *RBC* component and restarts the *recTimer*.

R8: When in *Connected* state is received a *SAI_DISCONNECT.indication* from the *SAI* component, the *CSL* sends an *RBC_User_Disconnect.indication* to the *RBC* component and moves to *Disconnected* state.

R9: When in *Connected* state the *recTimer* expires, the *CSL* sends a *SAI_DISCONNECT.request* to the *SAI* component, an *RBC_User_Disconnect.indication* to the *RBC* component and moves to *Disconnected* state.

Discarding of Messages

RD1: When in *Disconnected* state the *CSL* does not accept any kind of message except for *SAI_CONNECT.indication* from the *SAI* component.

RD2: When in *Connected* state the *CSL* discards *SAI_CONNECT.indication* and *SAI_ERROR.report* messages from the *SAI* component.

Requirements Specification for the ***Called SAI*** Component

Configuration Parameters
Initialization kind: *Execution Cycle* option.
System parameters,

- for *Execution Cycle* procedure:
    - *maximum initialization delay*
    - *Mec* (limit of the execution cycle counters)*;*
    - *K* (max acceptable transmission delay for a message);
- for *ACK* procedure:
    - *ack_request_period;*
    - *ack_response_timeout;*
- for *sequence number*:
    - *N* (limit of acceptable, consecutive message losses, *N = 1* means no losses)*;*
    - *M* (limit of the sequence number values, which have range *0..M-1*).

External Interactions
The *Called SAI* can receive from the *Called CSL* component the following messages:

- *SAI_DISCONNECT.request*;
- *SAI_DATA.request(message_type*[26]*, RBC_data_value)*;

and can send to the *CSL* the following messages:

- *SAI_CONNECT.indication*;
- *SAI_DISCONNECT.indication*;
- *SAI_DATA.indication(message_type, RBC_data_value)*;
- *SAI_ERROR.report*.

The *SAI* can receive from the *EuroRadio* Safety Layer (henceforth *ER-SL*) the following messages:

- *Sa_CONNECT.indication*;
- *Sa_DISCONNECT.indication*;
- *Sa_DATA.indication(message_type, SAI_data_value, ack_request, ack_response, sequence_number, execution_cycle_number)*;
- *Sa_ExecutionCycleStart(sequence_number, execution_cycle_counter)*;

and can send to the *ER-SL* the following messages:

- *Sa_CONNECT.response*;
- *Sa_DISCONNECT.request*;
- *Sa_DATA.request(message_type, SAI_data_value, ack_request, ack_response, sequence_number, execution_cycle_number)*;
- *Sa_ExecutionCycle(sequence_number, execution_cycle_counter)*.

Internal Variables

- *sequence_number;*
- *execution_cycle_counter;*
- *last_received_sequence_number;*

[26] *message_type* may refer to either *life_sign* or *RBC_data.*

- *last_received_execution_cycle_counter;*
- *execution_cycle_OFFSET.*

States

The *SAI* can be in the following four main states:

- *Connected*, when the communication is active;
- *Connecting*, when the communication is in the establishment phase;
- *Initializing*, while performing the execution cycle start procedure;
- *Disconnected*, when the communication is unactive.

External Guarantees

- The data messages delivered to the *CSL* are valid (i.e., arrived with a limited delay), neither duplicated nor reordered;
- no more than one data message per execution cycle is sent to the *ER-SL*;
- incoming messages are buffered and served with *FIFO* policy.

Behavioral Requirements

R1: At startup, the *SAI* is in *Disconnected* state.

When in Disconnected State

R2: When in *Disconnected* state is received a *Sa_CONNECT.indication* from the *ER-SL*, the *SAI* replies with a *Sa_CONNECT.response* to the *ER-SL* and moves to *Connecting* state.

When in Connecting State

R2b: When in *Connecting* state is received a *Sa_CONNECT.indication* from the *ER-SL*, the *SAI* replies with a *Sa_CONNECT.response* to the *ER-SL* and remains in the *Connecting* state.

R3: When in *Connecting* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.

R5: When in *Connecting* state is received a *Sa_ExecutionCycleStart(seqnum, ecnum)* from the *ER-SL*, the *SAI* replies with a *Sa_ExecutionCycle(seqnum, ecnum)* to the *ER-SL*, starts an *initTimer* set to the *maximum initialization delay, and* moves to *Initializing* state. The management of the *Sa_ExecutionCycleStart* parameters are done according to the rules in the *Sequence Numbers Management* and *Execution Cycle Counters Management* sections.

When in Initializing State

R2c: When in *Initializing* state is received a *Sa_CONNECT.indication* from the *ER-SL*, the *SAI* replies with a *Sa_CONNECT.response* to the *ER-SL* and moves to *Connecting* state.

R6: When in *Initializing* state the *maximum initialization delay* expires, the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and moves to *Disconnected* state.

R8: When in *Initializing* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* moves to *Disconnected* state.

R9: When in *Initializing* state is received a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *ER-SL* may have four cases, depending on the received *seqnum* and *ecnum* values (see REQ SEQ_NUMs and REC_ECNUMs).
* The *seqnum* is the one EXPECTED and *ecnum* is VALID: In this case the *SAI* moves to *Connected* state and sends both a *SAI_CONNECT.indication* and a *SAI_DATA.indication(msgtype, data)* to the *CSL* component.
* The *seqnum* is ACCEPTABLE and the *ecnum* is VALID: in this case the *SAI* moves to *Connected* state and sends a *SAI_CONNECT.indication*, a *SAI_DATA.indication(msgtype, data) and* a *SAI_ERROR.report* to the *CSL* component.
* The *seqnum* is NOT_ACCEPTABLE: in this case the *SAI* component sends a *Sa_DISCONNECT.request* to *ER-SL* and moves to *Disconnected* state.
* The *seqnum* is OLD or (the *seqnum* is ACCEPTABLE and the *ecnum* is VALID): In this case the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and discards the *Sa_DATA.indication* message.

<u>When in Connected State</u>

R10: When in *Connected* state is received a *SAI_DISCONNECT.request* from the *CSL* component, the *SAI* replies with a *SAI_DISCONNECT.indication* to the *CSL* component, sends a *Sa_DISCONNECT.request* to the *ER-SL*, and moves to *Disconnected* state.

R11: When in *Connected* state is received a *Sa_DISCONNECT.indication* from the *ER-SL*, the *SAI* sends a *SAI_DISCONNECT.indication* to the *CSL* component and moves to *Disconnected* state.

R12: When in *Connected* state is received a *Sa_CONNECT.indication* from the *ER-SL*, the *SAI* replies with a *Sa_CONNECT.response* to the *ER-SL*, sends a *SAI_DISCONNECT.indication* to the *CSL* component, and moves to *Connecting* state.

R13a: When in *Connected* state is received a *SAI_DATA_request(msgtype, data)* from the *CSL* component, and yet no other data message has been sent in this cycle, the *SAI* sends a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* to the *ER-SL*.
The *ackreq* and *ackresp parameters are set according to* REQ_ACKs.
The *seqnum* parameter is set according to SEQ_NUMs and the *ecnum* parameter is set according to REQ_ECNUMs.

R13b: When in *Connected* state is received a *SAI_DATA.request(msgtype, data)* from the *CSL* component, but another data message has already been sent in this cycle, the *SAI_DATA.request is saved in* a *FIFO dataout buffer (see also REQ_OUTDATABUFF).*

R14: When in *Connected* state is received a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *ER-SL* we can have four cases, depending on the received *seqnum* and *ecnum* values (see SEQ_NUMs and REC_ECNUMs).

* The *seqnum* is the one EXPECTED and *ecnum* is VALID: In this case the *SAI* sends a *SAI_DATA.indication(msgtype, data)* to the *CSL* component.
  Depending on the received values of the *ackreq* and *ackresp* parameters, appropriate actions are performed (see REQ_ACKs).
* The *seqnum* is ACCEPTABLE and the *ecnum* is VALID: in this case the *SAI* sends a *SAI_DATA.indication(msgtype, data) and* a *SAI_ERROR.report* to the *CSL* component.

Depending on the received values of the *ackreq* and *ackresp* parameters, appropriate actions are performed (see REQ_ACKs).

* The *seqnum* is OLD or (the *seqnum* is ACCEPTABLE and the *ecnum* is VALID): In this case the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and discards the *Sa_DATA.indication* message.
* The *seqnum* is NOT_ACCEPTABLE: In this case the *SAI* component sends a *Sa_DISCONNECT.request* to ER-SL, a *SAI_DISCONNECT.indication* to the *CSL* component, and then moves to *Disconnected* state.

OUTDATA Buffer Management

REQ_OUTDATABUFF1: At the beginning of each cycle, if the *dataout buffer* is not empty, the first *SAI_DATA.request(msgtype, data)* in the queue is removed and its data are used to send a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* to the *ER-SL.*
The ackreq, ackresp, seqnum, ecnum parameters are set according to *REQ_ECNUM, REQ_ACK*, and *REQ SEQNUM* requirements.

REQ_OUTDATABUFF2: When the *SAI* moves from the *Connected* state to the *Disconnected* state, the *dataout buffer is* emptied and the possibly waiting messages are discarded.

Execution Cycle Counters Management

REQ_ECNUM1: When entering in the *Initializing* state, the initial value of the *execution cycle counter* is set to 0.

REQ_ECNUM2: While in the *Initializing* or *Connected* state, the *execution cycle counter* is incremented modulo Mec at every cycle.

REQ_ECNUM3: When sending a *Sa_ExecutionCycleStart(seqnum,ecnum)* message or a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* the value of the *ecnum* parameter is set to the current value of the *execution cycle counter*.

REQ_ECNUM4: When receiving a *Sa_ExecutionCycleStart(seqnum,ecnum)* message from the *ER-SL,* the value of the *ecnum* parameter is used to compute the *EC_OFFSET* as difference between the current value of the *execution cycle counter and* the received *seqnum* value.

REQ_ECNUM5: When receiving a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message from *ER-SL*, the message in considered VALID if the message delay is less than *K*, where the message delay is computed as follows[27]:

```
message_delay = (execution_cycle_counter - EC_OFFSET) mod Mec[28]) - ecnum;
if message_delay < -Mec/2 then
        message_delay := message_delay + Mec;
elsif message_delay > Mec/2 then
        message_delay := message_delay - Mec;
end if
```

[27] This is a simplification from what required by UNISIG-098 as we assume that the EC period is 1 cycle for both *SAI* sides.

[28] Also when applied to negative numbers, (N mod M) is assumed to be equal to ((N+M) mod M).

Sequence Numbers Management

SEQ_NUM1: When entering in the state *Connected*, the *sequence_number* is set to 0.

SEQ_NUM2: When in *Connecting* state a *Sa_ExecutionCycleStart(seqnum,ecnum)* message is sent to the *ER-SL,* the *seqnum* parameter is set to the current value of *sequence_number*.

SEQ_NUM3: When in the *Initializing* or *Connected* state a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message is sent to the *ER-SL, the seqnum* parameter is set to the current value of *sequence_number*, and the *sequence_number* is incremented by *1 mod M*.

SEQ_NUM4: When in *Initializing* state is received a *Sa_ExecutionCycleStart(seqnum,ecnum)* message from *ER-SL*, the value of the *seqnum* parameter is saved as *last_received_sequence_number*.

SEQ_NUM5: When in the *Initializing* or *Connected* state is received a *Sa_DATA.indication (msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *ER-SL*, the distance of the current message from the last received one is computed as follows*:*

```
distance ≔ last_received_sequence_number – seq_num;
if (distance <  -M/2) then {distance :=  distance + M };
else if (distance > M/2) then {distance := distance - M };
```

If the distance value is equal to 1, the *seqnum* is considered EXPECTED.
If the distance value is lower than 1, the *seqnum* is considered OLD.
If the distance value is greater than 1 and less or equals to *N*, the *seqnum* is considered ACCEPTABLE.
If the distance value is greater than *N*, the *seqnum* is considered NOT_ACCEPTABLE.

ACK Management

REQ_ACK1: When in *Connected* state, the *SAI* periodically (with a configurable *ack_request_period*) sets an *ackreq* flag to the first *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message to be forwarded to the *ER-SL* and starts an *ack_response_timer* with a *max_response_delay* limit.
The *ackreq* flag is not set and the timer is not started if the *SAI* is still waiting for the response to a previous ack request.

REQ_ACK2: When the *ack_response_timeout* expires, if a *Sa_DATA.indication(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message with an *ackresp* parameter set has not yet been received from the *ER-SL*, the *SAI* sends a *SAI_ERROR.report* to the *CSL* component and restarts the *ack request timer*.

REQ _ACK3: While in *Connected* or *Initializing* state, when it is received a *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* from the *CSL* component, the *SAI* sets the *ackresp* parameter in next *Sa_DATA.request(msgtype, data, ackreq, ackresp, seqnum, ecnum)* message to be sent to the *ER-SL*.

Discarding of Messages

RD1: When in *Disconnected* state the *SAI* discards any message except for *Sa_CONNECT.indication* from the *ER-SL*.

RD2: When in *Connecting* state, the *SAI* discards any message except for,
Sa_DISCONNECT.indication and *Sa_ExecutionCycleStart* from the *ER-SL*;

RD3: When in *Initializing* state, the *SAI* discards any message except for, *Sa_DISCONNECT.indication* and *Sa_DATA.indication* from the *ER-SL*;

RD4: When in *Connected* state, the *SAI* discards any *Sa_ExecutionCycleStart* message from the *ER-SL*.

Appendix D: Difformities with respect to the D2.3 Requirements

In this Appendix we summarize the semantic differences between the initial D2.3 and new D2.5 requirements in terms of completeness, consistency, and implementation dependent choices. These are highlighted as **Remarks:** annotations.
Some observations are also made on the initial D2.3 requirements about minor presentation points related to not clearly presented aspects. These are highlighted as **Presentation**: annotations.

CSL

REQ_001	*If configured as initiator, when switched on (communication in state NOCOMMS), the CSL is responsible to send to underlying Layers the command for the establishment of a safe connection with the partner RBC, and to command re-establishment of safe connection when it is considered lost (communication in state NOCOMMS).*
	Presentation: The requirement overlaps with *REQ_012*.
	Remarks: For *CSL* configured as *initiator* the *NOCOMMS* state is logically constituted by three substates, that is: *Waiting*, *Ready*, and *Connecting*. When switched on the *CSL* is in *NOCOMMS Ready* substate.

REQ_002	*After sending the command for the establishment of the connection, a timer shall be started by the initiator. If the timer expires before the connection is established, a new connection request shall be generated.*
	-
	-

REQ_003	*If configured as called, the CSL shall wait for report from underlying Layers that a safe connection is established.*
	Presentation: The requirement overlaps with *REQ_014*.
	-

REQ_004	*The CSL shall discard any message either from User functions or from partner CSL before a confirmation of successful clock offset estimation (TTS option) or EC initialization has been received from SAI sublayer.*
	Presentation: There is no need to refer to the specific *SAI* option for initializing the safe connection. Data messages are always discarded when not in *COMMS* state.
	-

REQ_005	*The CSL shall forward a received User message to RBC User functions only if all checks specified in supervision functions (7.2.2) are passed.*
	Presentation: Checks over User messages are performed by *SAIs* instead of *CSLs*. The requirement overlaps with *REQ_017*.
	Remarks: When in *NOCOMMS* states User messages are not forwarded.

REQ_006	*Loss of safe connection shall be detected by the CSL reading reports from the underlying SFM (SAI_DISCONNECT.indication).*
	-

	-

REQ_007	*If a report from underlying Layers is received that safe connection is lost, the CSL shall consider the communication in state NOCOMMS.*
	Remarks: The *initiator CSL* moves to *NOCOMMS Ready* state, while the *called CSL* moves to *NOCOMMS* state.

REQ_008	*TTS option: after reception of report from SAI that the clock offset procedure has been completed, the CSL shall ensure that a message is sent to the partner RBC at the expiration of a configurable transmit time interval (reset at the sending of any message). If no User message needs to be sent, CSL is responsible to send a life sign message (see Figure 4);* *EC option: After reception of report from SAI that the EC initialization procedure has been completed, the sending of messages is scheduled cyclically every (configurable) TC. If no request to send messages from User application is pending, a life sign is sent by CSL. If requests are pending, only one message per cycle is sent.*
	Presentation: The requirement mixes both *SAI* and *CSL* aspects.
	Remarks: When moving in *COMMS* state there is no User message to be sent; a first life sign can be sent without waiting for the expiration of the transmit time (implementation freedom).

REQ_009	*After reception of report from SAI that the clock offset procedure or EC initialization has been completed, the condition where no valid messages are received within a configurable time shall be recognized by the CSL. This is achieved by means of a configurable receive timer (started at the reception of report from SAI on completion of initializations and reset at the reception of any message); if no message (User or life sign) is received within such configurable receive time interval, the communication shall be considered in state NOCOMMS.*
	Presentation: The requirement mixes both *SAI* and *CSL* aspects.
	-

REQ_010	*When communication is in state NOCOMMS, the CSL shall not accept/forward messages neither from its own RBC User functions nor from partner RBC; when switching to NOCOMMS, if the safe connection is still active, the CSL shall send a termination order (SAI_DISCONNECT.request).* *Note: when informed that the communication is in state NOCOMMS, the User functions will terminate all transactions.*
	Presentation: This requirement overlaps with *REQ_015*. The note describes an aspect not related to the CSL behavior.
	-

REQ_011	*CSL can switch the communication from state NOCOMMs to state COMMS only when underlying Layers confirm the re-establishment of a safe connection.* *Note: communication in state COMMS is communicated to User functions, that will be able to restart management of transactions.*
	Presentation: The requirement partly overlaps with *REQ_019*.
	-

REQ_012	*If configured as initiator, at start-up, and when loss of safe connection is detected, the CSL shall send safe connection init order to from SFM (SAI_CONNECT.request).*
	Presentation: The requirement overlaps with *REQ_001*.
	-

REQ_013	*If configured as initiator, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.confirm).*
	Presentation: The requirement overlaps with *REQ_008*, *REQ_009*, *REQ_011*, and *REQ_019*.
	Remarks: *SAI_CONNECT.confirm* message is accepted in *NOCOMMS Connecting* state only.

REQ_014	*If configured as called, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.indication).*
	Presentation: The requirement overlaps with *REQ_003*.
	-

REQ_015	*In case loss of communication is detected due to no valid messages received within a configurable time, the CSL shall send a safe connection termination order to SFM (SAI_DISCONNECT.request).*
	Presentation: The requirement overlaps with *REQ_010*.
	-

REQ_016	*While the safe communication is active (state COMMS), the CSL is responsible of sending User messages received from RBC User functions to partner RBC.*
	Presentation: *CSL* sends to the *SAI* User messages coming from its own *RBC*. *CSL* sends to its own *RBC* User messages coming from *SAI*. The requirement overlaps with *REQ_020*.
	-

REQ_017	*While the safe communication is active (state COMMS), the CSL is responsible of checking User messages received from partner RBC and forwarding (if checks are passed, see 7.2) to RBC User functions.*
	Presentation: The requirement describes unnecessary *SAI* related aspects. *CSL* receives *RBC* User messages from the *SAI*. The requirement overlaps with *REQ_021*.
	-

REQ_018	*The CSL is responsible of reading reports from SFM.*
	Presentation: The requirement overlaps with several other requirements without reporting anything relevant.
	-

REQ_019	*The CSL is responsible of sending reports to RBC User functions about state of communication (COMMS/NOCOMMS).*
	Presentation: The requirement partly overlaps with *REQ_011*, *REQ_022*, and *REQ_023*.
	Remarks: State switching like *NOCOMMS Waiting to NOCOMMS Ready, to NOCOMMS Connecting*, and vice versa are not reported to *RBC* User functions.

REQ_020	*CSL shall receive from User functions the messages to be forwarded to peer RBC User when in state COMMS.*
	Presentation: The requirement overlaps with *REQ_016*.
	-

REQ_021	*CSL shall forward to User functions the forwarding of messages received from communication partner.*

	Presentation: The requirement overlaps with *REQ_017*.
	-

REQ_022	*CSL shall send to User functions the reports on loss of communication (missing life sign - state NOCOMMS).*
	Presentation: The requirement overlaps with *REQ_019*.
	-

REQ_023	*CSL shall send to User functions the reports on state of safe connection state change (COMMS/NOCOMMS).*
	Presentation: The requirement overlaps with *REQ_019*.
	-

REQ_024	• *SAI_CONNECT.request shall be used by initiator CSL to command the establishment of a safe connection* • *SAI_CONNECT.indication shall be used by called SAI to notify to the CSL the connection establishment request* • *SAI_CONNECT.response shall be used by called CSL to accept the connection request.* • *SAI_CONNECT.confirm shall be used by the initiator SAI entity to inform the CSL about the successful establishment of the safe connection.*
	-
	-

REQ_025	• *SAI_DATA.request shall be used by CSL to transmit data to the peer entity.* • *SAI_DATA.indication shall be used to indicate to the CSL that data have been received successfully from the peer entity.*
	-
	-

REQ_026	• *SAI_DISCONNECT.request shall be used by the CSL to enforce a release of the safe connection.* • *SAI_DISCONNECT.indication shall be used to inform the CSL about a safe connection release.*
	-
	-

REQ_027	*SAI Error Report shall be sent from SAI to CSL in case of errors detection by SAI (deletion, resequencing, delay, repetition).*
	Presentation: The requirement describes only aspects related to *SAI*.
	-

SAI

REQ_028	*If SAI receives a command to establish a safe connection from CSL (CSL configured as initiator), SAI shall forward this order to ER Layer.*
	-
	Remarks: The command to establish a safe connection should be discarded by *SAI* if it arrives in a state different from *NOCONN Disconnected*. Furthermore, once the order to establish a safe connection to the *ER Layer* has been forwarded, the *SAI* moves to *NOCONN Connecting* state. At startup, the *SAI* is in *NOCONN Disconnected* state.

REQ_029	*In case initiator, when SAI receives a confirmation of safe connection established from ER Layer, SAI shall start the initialization procedure (initial clock offset estimation for TTS option or initialization for EC option).*
	Presentation: The requirement partly overlaps with *REQ_044*.
	Remarks: A confirmation of safe connection should be discarded by *SAI* if it arrives in a state different from *NOCONN Connecting*. When in *NOCONN Connecting* state is received a connect confirmation from the *ER Layer*, the *SAI* moves to *NOCONN Initializing* state.

REQ_030	*In case called, if SAI receives a safe connection establishment indication from the ER Layer, SAI shall send a confirmation to ER Layer and wait for the start of the initialization procedure (initial clock offset estimation for TTS option and initialization for EC option).*
	-
	Remarks: When in *NOCONN Disconnected, NOCONN Connecting*, or *NOCONN Initializing* state is received a *Sa_CONNECT.indication* from the *ER* Layer, the *SAI* replies with a *Sa_CONNECT.response* to the ER Layer and moves to *NOCONN Connecting* state. If a *Sa_CONNECT.indication* is received in *NOCONN Connected* state, the *SAI* sends a *Sa_CONNECT.response* to the *ER Layer* and a *SAI_DISCONNECT.indication* to the *CSL*.

REQ_031	***(Robustness requirement)*** *Considering that the communicating RBCs might be affected by loss of communication at different time, the called RBC protocols shall accept the re-establishment of a safe connection even if they are still considering the communication not lost.*
	Presentation: The requirement does not report any behavior of the *SAI*.
	Remarks: Robustness requirements are inconsequential compared to those describing a precise behavior of the component.

From *REQ_032* to *REQ_043* is described the *Triple Time Stamp* (*TTS*) option, which has not been considered in this exercising of the demonstrator.

REQ_044	*When ER sublayer reports the successful establishment of safe connection, SAI initiating the safe connection establishment (initiator) shall send an ExecutionCycleStart message containing its initial value of EC counter and the EC period.*
	Presentation: The requirement partly overlaps with *REQ_029*, *REQ_068*, and *REQ_069*. The required management of the parameters is not explicitly specified. It is necessary to investigate the *UNISIG-SUB-98* standard for understanding their use. In fact, based on the *UNISIG-SUB-98* the *ExecutionCycleStart* message also carries a sequence number parameter.
	Remarks: When in *Connecting* state is received a *SA_Connect.confirm* from the ER-SL, the *SAI* (configured as initiator) replies with a *Sa_ ExecutionCycleStart(sequence number, current EC value)* to the *ER-SL* and moves to *Initializing* state. In the modelled scenarios, the *EC* period is assumed to be either implicitly equal to 1 for both sides or removed from the actual parameters. The flowing of time is modelled in an approximate way. SAI should start the safe connection establishment phase only when, in *NOCONN Connecting* state, is received a connect confirmation from the *ER Layer*. Otherwise, it just discards the message. The expected use of *EC* values and sequence numbers is not explicitly described. The *UNISIG-SUB-98* does not completely specify the way in which *EC* values are used in the computation of *OFFSET* and delays as well. Especially when we are in presence of values which have overflowed from their maximum allowed value.

REQ_045	*The responder SAI shall answer to an ExecutionCycleStart message with an ExecutionCycleStart message containing its initial value of EC counter and the EC period and report to the CSL that the initialization procedure has been completed.*
	Presentation: Part of the requirement overlaps with *REQ_051, REQ_068, and REQ_069*. The required management of the parameters is not explicitly specified. It is necessary to investigate the *UNISIG-SUB-98* standard for understanding their use. In fact, based on the *UNISIG-SUB-98* the *ExecutionCycleStart* message also carries a sequence number parameter.
	Remarks: On the responded side, the initialization procedure is considered completed NOT when the called SAI replies to the *ExecutionCycleStart message with another ExecutionCycleStart* message, but when it receives a first *Sa_DATA.indication from the ER Layer*. When the first *Sa_DATA.indication* from the *ER Layer* is received the called SAI moves to the *CONN Connected* state and sends a *SAI_CONNECT.indication* to the *CSL*, notifying that the initialization procedure has been completed. The expected use of *EC* values and sequence numbers is not explicitly described. The *UNISIG-SUB-98* does not completely specify the way in which *EC* values are used in the computation of *OFFSET* and delays as well. Especially when we are in presence of values which have overflowed from their maximum allowed value.

REQ_046	*After sending any of the above listed messages, the SAI shall start a timer with configurable time out. If the time out expires before the reception of a new message (that is a User message or a life sign, in the case of the responder SAI) the procedure is cancelled, and the error is reported to the CSL.*
	-
	Remarks: If the initialization procedure does not complete within the required timeout, the *SAI* moves from *NOCONN Initializing* state to *NOCONN Disconnected* state and sends an *SAI_Error.report* to the *CSL*.

REQ_047	*At the reception of the message from the responder, the initiator SAI shall inform the CSL that the initialization procedure has been completed.*
	-
	Remarks: The SAI sends a SAI*_CONNECT.confirm* to the CSL and moves to *CONN Connected* state only when the ExecutionCycleStart reply arrives in *NOCONN Connecting* state. Otherwise, the reply is simply discarded without further actions by SAI.

REQ_048	*With a configurable period, by each communicating party, SAI shall ensure that an application message with request of ACK is sent and start a timer (note: here the ACK specified in message type is meant, not the ACK managed at User application level inside the User messages).*
	-
	Remarks: If the configurable period for requesting an ack expires and the previous ack request has not received a response yet, the request is not repeated.

REQ_049	*At the request of an ACK, the responding SAI shall ensure that an application message with ACK is sent.*
	-
	-

REQ_050	*If the application message with ACK is not received before expiration of the timer an error is reported to the CSL.*
	-
	-

REQ_051	*For TTS option: No User message shall be accepted by SAI neither from CSL nor from ER sublayer if the clock offset estimation has not been completed (report from SAI to CSL).* *For EC option : No User message shall be accepted by SAI neither from CSL nor from ER sublayer if the initialization procedure for EC parameters has not been completed (report from SAI to CSL).*
	Presentation: For *EC* option, the requirement overlaps with *REQ_045*.
	-

REQ_052	*For messages sent by CSL (including life sign messages), SAI shall recognize the destination of the message from the content of request received from CSL.*
	-
	Remarks: In our scenario we have only one initiator side statically connected with a called side. The destination of messages is implicit and not subject to changes.

REQ_053	*Messages originated by SAI itself (e.g., clock offset estimation) shall contain indication of destination.*
	-
	Remarks: In our scenario we have only one initiator side statically connected with a called side. The destination of messages is implicit and not subject to changes.

REQ_054	*Messages originated by SAI itself (e.g., clock offset estimation) shall comply with SUBSET-098.*
	Presentation: Not self-contained requirement. Some *SAI* information can only be acquired by reading the *SUBSET-098*.
	-

REQ_055	*The SAI shall add a message type to User data to be sent. See SUBSET-098.*
	Presentation: Not self-contained requirement. Some *SAI* information can only be acquired by reading the *SUBSET-098*.
	-

REQ_056	*The SAI shall add a sequence number to User data to be sent; the sequence number shall be increased by one at any new message sent (irrespective of its type).*
	-
	Remarks: The requirement should mention the initial sequence number exchanged between *SAIs*, which is carried by an *EC* message. After reaching their maximum value, the sequence number series restarts from 0.

	Sequence numbers are added onto *Sa_DATA.request* and *ExecutionCycleStart* messages.

REQ_058	*For EC option, the SAI shall add to the User data to be sent the current value of the cycle counter EC.*
	-
	-

REQ_059	*A received User message shall be forwarded to CSL only if all checks specified in Supervision functions are passed.*
	-
	-

REQ_060	*When an order for termination is received from CSL, SAI shall forward it to ER sublayer.*
	-
	Remarks: The *initiator SAI* should always reply to a *SAI_DISCONNECT.request* sent by the *CSL* with a *SAI_DISCONNECT.indication,* also when such message is received in the *NOCONN Disconnected* state. When in *NOCONN Disconnected* state the called SAI receives an order of termination from *CSL*, such order is discarded.

REQ_061	*When an indication of disconnection is received from ER sublayer, SAI shall forward it to the CSL.*
	-
	Remarks: If an indication of disconnection (*Sa_DISCONNECT.indication)* is received from the *ER* when the *initiator or called SAI* is in a state different from the *NOCONN Connected* one, the indication of disconnection is not forwarded to the CSL.

REQ_062	*The receiver SAI shall accept any value for the sequence number of the first message after establishment of safe communication.*
	-
	-

REQ_063	*If N (configurable) consecutive messages are missing in the sequence of the received messages, i.e., if a message whose sequence number is greater that the sequence number of the last correctly received message + N, the message shall be ignored, and the SAI shall send an order to terminate the safe connection to ER sublayer and report its state to CSL.*
	-
	Remarks: The arithmetic for counting the number of consecutive missing messages should also take into consideration the possibility of overflows (i.e., sequence numbers restarting from zero after reaching their allowed maximum value). This aspect is overlooked in the UNISIG_SUBSET_098 standard as well.

REQ_064	*In case the sequence number of a received message is greater than the sequence number of the last correctly received message + 1 and lower than the sequence number of last correctly received message + N, the message shall not be discarded, and SAI shall report to CSL the occurrence of the communication error.*
	-
	Remarks: The arithmetic for counting the number of consecutive missing messages should also take into consideration the possibility of overflows (i.e., sequence numbers restarting from zero after reaching their allowed maximum value). This aspect is overlooked in the UNISIG_SUBSET_098 standard as well. The order in which error reports and the data messages should be sent is not specified.

	The implementation must take a choice, but any choice is not likely to create interoperability problems.

REQ_065	*If the sequence number of the received message is lower or equal to the sequence number of an already received message, the new message shall be discarded, and SAI shall report to CSL the occurrence of the communication error.*
	-
	Remarks: The arithmetic for counting the number of consecutive missing messages should also take into consideration the possibility of overflows (i.e., sequence numbers restarting from zero after reaching their allowed maximum value). This aspect is overlooked in the UNISIG_SUBSET_098 standard as well.

REQ_066	*The SAI of the receiver entity shall recognize a message that, after sending, has been delayed in the communication channel for a time greater than a configurable value.*
	-
	-

REQ_068	*For EC option the acceptance of a message shall be checked according to SUBSET-098.*
	Presentation: Not self-contained requirement. Some *SAI* information can only be acquired by reading the *SUBSET-098*.
	Remarks: The arithmetic for evaluating the occurred delay should take into consideration the possibility of overflows (Execution Cycles numbers restarting from zero after reaching their allowed maximum value). This aspect is overlooked in the UNISIG_SUBSET_098 standard as well.

REQ_069	*For EC option the corrections specified in SUBSET-098.*
	Presentation: Not self-contained requirement. Some *SAI* information can only be acquired by reading the *SUBSET-098*.
	Remarks: *EC* corrections are not modelled.

The following requirements add details to the description of SAI behaviors provided by the previous ones, by mapping logical events to specific interface signals (e.g., *Sa_DISCONN.indication*, *SAI_DATA.request*, etc.).

REQ_070	*At start-up, and when loss of safe connection is detected (Sa_DISCONN.indication), the SAI, if configured as initiator, shall wait for order from CSL.*
	-
	-

REQ_071	*At start-up, and when loss of safe connection is detected (Sa_DISCONN.indication), the SAI, if configured as called, shall wait for reception of safe connection established confirmation from ER sublayer (Sa_CONN.indication).*
	-
	-

REQ_072	*In case loss of safe connection is detected, the SAI shall send a safe connection report to CSL (SAI.DISCONN.indication).*
	-
	-

REQ_073	*The SAI shall be responsible of Sending User messages received from CSL (SAI_DATA.request) to partner RBC (through Sa_DATA.request).*
	-
	-

REQ_074	*The SAI shall be responsible of Checking User messages received from partner RBC (through Sa_DATA.indication) and forwarding (if checks are passed) to CSL (SAI_DATA.indication).*
	-
	-

REQ_075	*The SAI shall be responsible of Reading reports from ER sublayer (Sa_DISCONNECT.indication).*
	-
	-

REQ_076	*The SAI shall be responsible of Sending reports to CSL (SAI.DATA.indication, SAI.CONNECT.indication and SAI.DISCONNECT.indication).*
	-
	-

REQ_077	*Sa_CONNECT.request shall be used by initiator SAI to command the establishment of a safe connection.* *Sa_CONNECT.indication shall be used by called ER to notify to the SAI the connection establishment request.* *Sa_CONNECT.response shall be used by called SAI to accept the connection request. The response shall always be sent automatically without any authorization from upper layers.* *Sa_CONNECT.confirm shall be used by the initiator ER entity to inform the SAI about the successful establishment of the safe connection.*
	-
	-

REQ_078	*Sa_DATA.request shall be used by SAI to transmit application data to the peer entity.* *Sa_DATA.indication shall be used to indicate to the SAI that data have been received successfully from the peer entity.*
	-
	-

REQ_079	*Sa_DISCONNECT.request shall be used by the SAI to enforce a release of the safe connection.* *Sa_DISCONNECT.indication shall be used to inform the SAI about a safe connection release.*
	-
	-

Appendix E: Model reduction techniques

The possibility of representing all the possible evolutions of a system in the form of an explicit LTS (e.g., in the *.aut* format) paves the way to the exploitations of the many results that have been accumulated through the years upon these structures.

The most basic to minimize a single LTS is to reduce it according to the so-called *strong bisimulation*. This minimization essentially reduces the statespace removing duplicated branches but preserving the same logical structure.
An example of this equivalence/reduction is shown in the Figure 9.

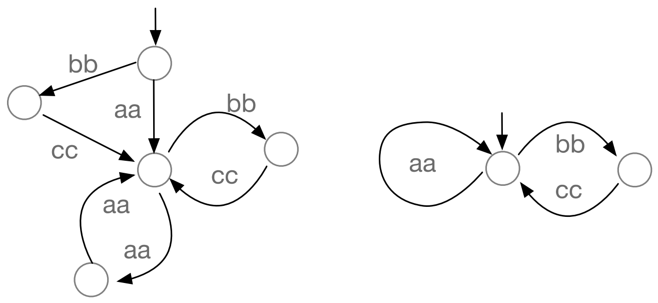
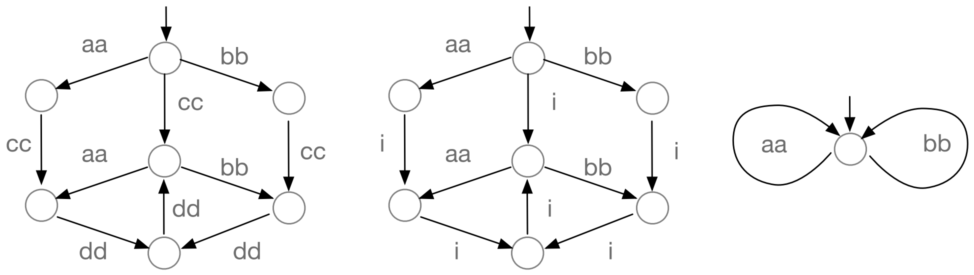


Figure 9 two strongly equivalent LTS

The nice property of *strong* equivalence [STRONG] is that the two behaviors are completely equivalent, i.e., there *is no property reasoning on the labels of the LTS that is satisfied by one model but not by the other*. All action-based temporal logics are adequate w.r.t. this equivalence.
Moreover, this equivalence is also a congruence w.r.t. parallel composition [Compositional]. This means that we have a system composed as P1 // P2, we can separately minimize P1 and P2, and the resulting system P1min // P2min is still strongly equivalent to P1 // P2.
In our case, we can prove that the UMC, ProB, and LNT models are strongly equivalent[29].

Much greater reductions can be obtained if not all the possible labels are relevant for the evaluation of a certain property. In this case, we might "hide" (i.e., replace the actual label with an unobservable symbolic label "i" or "tau") all the irrelevant labels and minimize the system even more. A bisimulation/minimization which is particularly well-fitting for this purpose is the so-called divbranching bisimulation [DBR]. Figure Y shows an example of the use of divbranching minimization: suppose that on the process of the side we want to check the property that is "it is always eventually possible to generate an event *aa* or an event *bb*. We might first "hide" all the irrelevant labels *cc* and *dd*, obtaining the LTS in the middle, and then applying the divbranching minimization obtaining the LTS of the right.

[29] one we appropriately align the labels in the LTS, and eventually skip additional initial setup steps.



However, not all properties are preserved by this divbranching minimization. Some of the action-based temporal logic that can be safely used for this purpose are ACTL-X[ACTLX], Lmu-db[LDBR], and various weak fragments of UCTL, LTL, PDL.
An example of the application of this process within the CADP framework is shown by the following SVL [SVL] script:

```
"minimizedsystem.bcg" = divbranching reduction of
hide all but  aa, bb in
  "originalsystem.bcg"
end hide;

property AA_BB_ALWAYS_EVENTUALLY_POSSIBLE
  "it is always eventually possible to generate an event aa or an event bb"
is
"minimizedsystem.bcg"  |=
    with evaluator4
    library actl_x.mcl end_library
        AG((AF(aa) and (AF(bb));
    expected TRUE
end property
```

Further improvements of this approach, which extend the set of properties that can be verified, have been introduced with the introduction of sharp bisimulations [SHARP] and by the possibility to mix different compatible bisimulations during the final parallel composition of the components of a system [Combining].

Finally, there is a last minimization that might be taken into consideration, at least for documentation purposes. This is the *complete-divergence-sensitive-weak-trace* minimization. Actually, no framework directly supports this minimization, but it can be obtained by applying the classical weak-trace minimization to an LTS which has been enriched with explicit information about deadlocks and infinite self-loops of hidden actions. The program *wtprepare* mentioned in Appendix 8.1.2 has precisely the purpose of preparing an LTS in .aut format for such minimization. The result of this minimization is an LTS that describes in the most compact way all the possible execution traces of the system, completely removing all hidden transitions except those leading to infinite self-loops.
A simple example of this minimization is shown in the Figure 10:

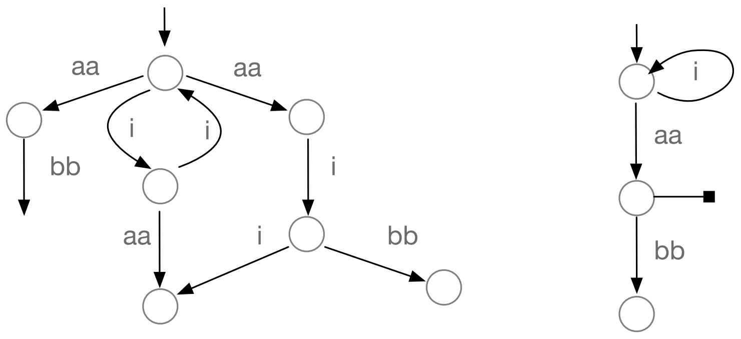


Figure 10: example of complete-div-sensitive-weak trace minimization

Since the graph of all the possible sequences of events occurring at an interface with a system is usually information of interest to a system designer, this minimization can be very useful for documentation purposes. However, since the original branching structure of the system is lost, only a few formal temporal properties (e.g., weak action-based LTS fragments) are preserved by this minimization.

Appendix F: Analysing the behavior at the interfaces

Once we are confident that the operational model correctly reflects the intended internal behavioral requirements, we might proceed in verifying that the stated behavioral requirements (i.e., the formal model) imply the stated external guarantees of the system component, and the expected designer objectives. Several architecture and scenarios have been defined for this purpose, and the corresponding models can be retrieved from the public Zenodo repository [ZenodoWP2].

ICSLtesting_V27_continuosdata

For example, the scenario *ICSLtesting_V27_continuosdata*[30] is one of those developed early in the analysis process, and is used to analyze the behavior of the initiator CSL in a standalone way (i.e., providing abstract SAI RBC components as part of the stimulating environment). In this scenario the RBC component waits connect indications from the CSL and, as long as connected sends one messages every two timeslots, with a max of N messages. The SAI component is a very abstract one which simply accepts all orders from the CSL, and at each cycle randomly sends connect, disconnect, rbc user data, life sign indications, and error reports.

In this scenario, all the I_CSL transitions appear to be eventually triggered (i.e., we have achieved a 100% coverage), even if when the I_CSL is integrated with all the other, more realistic, system components, several transitions might no longer appear as reachable.

One way to observe the external behavior of the component in one scenario is just to observe all the possible traces of messages flowing between the components.
For example, if we want to observe all the possible message flows from the ICSL towards the RBC, we can take the LTS describing all possible evolutions of with our scenario, hide all the labels not belonging to the set of interactions we want to observe, and minimize the resulting LTS (as shown in Section 8.1.3) with weak (complete, divergence sensitive) trace equivalence.
This analysis process can be carried out within the CADP framework[31] with the SVL script shown in Figure 14. The result can be observed in Figure 15[32].

Looking of the Figure 15 we can easily observe the satisfaction of several ICSL external guarantees of those mentioned in Appendix C. In particular:

- ICSL can send to I_RBC an *RBC_Data_indication* message only after a *RBC_Connect_indication* not followed by *RBC_Disconnect_indication*.
- ICSL can send to I_RBC an *RBC_Disconnect_indication* message only after an

[30] All the analysed scenarios and architecture are available in the Zenodo data repository [ZENODO].
[31] The same result can be obtained using the *umc2aut* and *ltsconvert* tools from the free KandISTI /UMC and mCRL2 frameworks.
[32] The same effect can be achived using the online version of UMC, selecting the appropriate filters for the messages to be observed and using the command "Draw Abstract Traces".

RBC_Connect_indication not already followed by *RBC_Disconnect_indication*.

- The first message (possibly) sent to I_RBC is an *RBC_Connect_indication* message
- ICSL sends to I_RBC an *RBC_Connect_indication* message only as first messages or after an *RBC_Disconnect_indication* not already followed by *RBC_Connect_indication*.

```
% umc2lnt ICSLtesting_V27_continuosdata.umc continuosdata.lnt;
"continuousdata.bcg" =
    generation of "continuosdata.lnt";
"continuous_rbcflow_dbmin.bcg" =
    divbranching reduction of
    gate hide all but
      IRBC_User_Connect_indication,
      IRBC_User_Disconnect_indication,
      IRBC_User_Data_indication
   in  "continuousdata.bcg";
% bcg_io continuous_rbcflow_dbmin.bcg continuous_rbcflow_dbmin.aut;
% wtprepare -i continuous_rbcflow_dbmin.aut continuous_rbcflow_wtready.aut
% bcg_io continuous_rbcflow_wtready.aut continuous_rbcflow_wtready.bcg
"continuous_rbctraces.bcg" =
   weak trace reduction of
   multiple rename
       "IRBC_USER_DATA_INDICATION !.*"  -> "IRBC_USER_DATA_INDICATION "
   in "continuous_rbcflow_wtready.bcg";
% bcg_io continuous_rbctraces.bcg continuous_rbctraces.aut;
% aut2dot continuous_rbctraces.aut continuous_rbctraces.dot
% dot -Tsvg continuous_rbctraces.dot -o continuous_rbctraces.svg
```

Figure 14: A SVL script for generation of ICSL-RBC traces

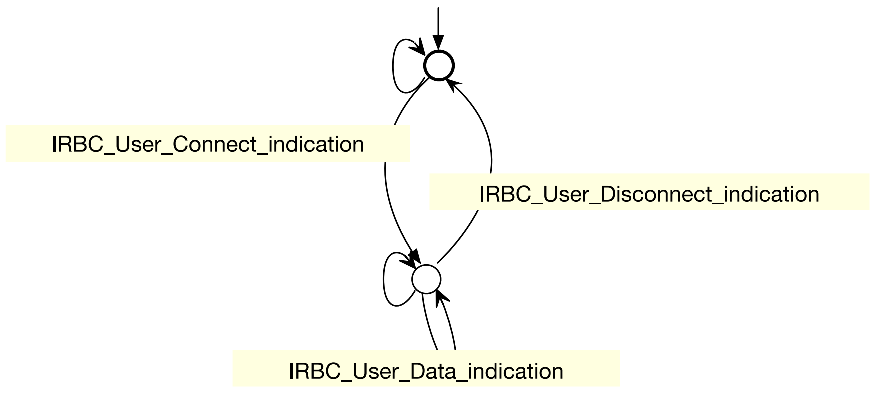

IRBC_User_Connect_indication
IRBC_User_Disconnect_indication
IRBC_User_Data_indication

Figure 15: An ICSL->RBC message flow in the scenario *ICSLtesting_V27_continuosdata*

Other properties that can be observed from these traces are that in this scenario there is no guarantee that a communication line is ever established, and that even if established, there is no guarantee that any message arrives, and no guarantee that the connection is eventually terminated.
Notice that the stated properties hold for *all* the possible system evolutions, therefore it is not a problem if the *actual* system evolutions, when the abstract environment components are replaced by more concrete system fragments, are just a subset of those here analysed.

We might have verified the above properties by translating them into temporal logic formulas and verifying them with CADP, ProB, or UMC, but with a relatively greater effort.

When the graphical representation of all the possible message flows becomes bigger, the approach of just observing the picture might not be feasible, and the formal encoding and verification of the formulas risks to remain the only reliable approach.

Suppose that, in the same architecture/scenario, we want to analyze the other "external I_CSL" guarantee:

ICSL periodically sends to I_SAI either SAI_Connect_request or SAI_Data_request messages.

We might repeat the same process described above for observing all the possible message flows from I_CSL towards ISAI involving Data or Connect requests, obtaining as result the description all the possible traces shown in Figure 16.

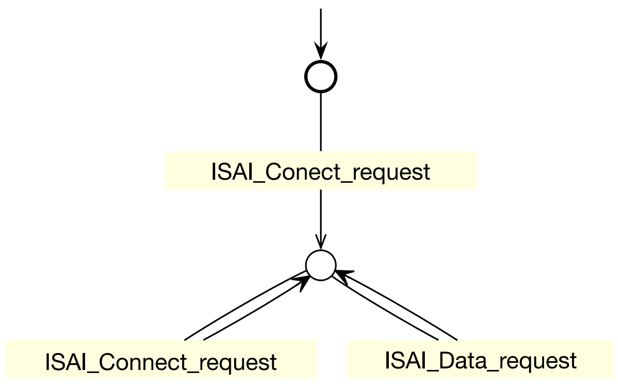


Figure 16: A ICSL->SAI message flow in the scenario *ICSLtesting_V27_continuosdata*

As we can easily see, there is no unlabeled loop (originated by hiding of actions different from a DATA or e CONNECT request) in the computed messages flow.

The ProB LTL formulas directly checking this property would be instead:

```
-- UMC-UCTL:   AG AF {ISAI_DATA_request or ISAI_CONNECT_request} true
-- PROB-LTL:  G F ( [R8_ICSL_saidatareq] or [R7b_ICSL_saidatareq] or [R2_ICSL_connecting])
```

ICSLtesting_V27_incrdata.

Let us now analyze the flow of messages between RBC and SAI (just looking at ICSL s black box). We want to observe also the identity of messages to check that no repetitions or mis-ordering are introduced by the CSL. Messages are sent by RBC only after having received a connection indication not followed by a disconnection indication. For this, we are using the scenario *ICSLtesting_V27_incrdata*. observing only the exchange of the RBC data messages (no life sign, or connect/disconnect events). Figure 17 shows the generated flow of such messages.

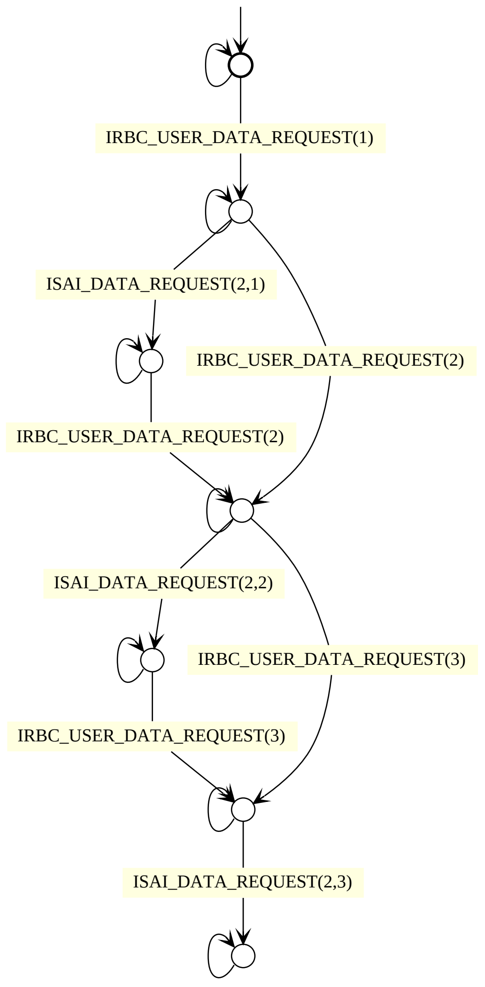

IRBC_USER_DATA_REQUEST(1)
ISAI_DATA_REQUEST(2,1)
IRBC_USER_DATA_REQUEST(2)
IRBC_USER_DATA_REQUEST(2)
ISAI_DATA_REQUEST(2,2)
IRBC_USER_DATA_REQUEST(3)
IRBC_USER_DATA_REQUEST(3)
ISAI_DATA_REQUEST(2,3)

Figure 17: Flow of RBC Data requests messages through the ICSL

Two interesting things can be easily observed in this picture:

- There is no guarantee that any message is sent (see the loop in the initial node). Indeed, there is no guarantee that the ICSL ever has succeeded in establishing an active communication line.

- Even if active communication line is established, and a message sent (i.e., after the RBC has received a connection indication and before receiving any disconnect indication), there is no guarantee that the message is passed to the SAI.

At first, this might look surprising and in contrast with the ICSL REQ8:

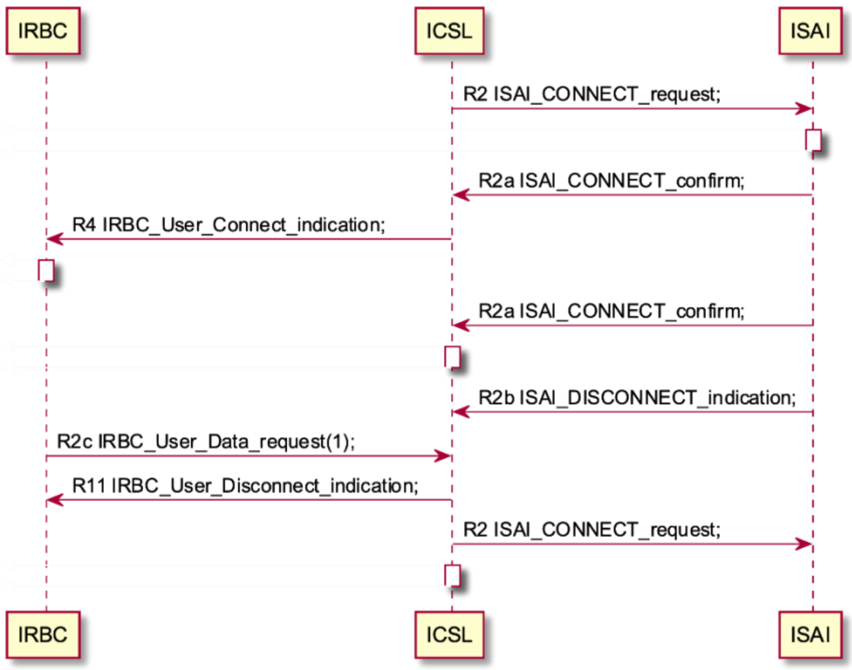
R8: *When in COMMS state is received an RBC_User_Data_request(userdata), I_CSL forwards a SAI_Data_request(userdata) with the same data to the SAI.*

But this is a possible valid behavior because the current RBC view of the ICSL status might not precisely reflect the realstatus of CSL. The ICSL might pass in the NOCOMM state just before receiving the RBC_Data_request, which can be sent just before receiving the Disconnect indication from the CSL. In this case, the RBC_Data_request might arrive when ICSL is in the NOCOMM state, and the Data_Request message would be discarded.

This can be checked, in the UMC framework, with the evaluation of the formula:

EF {R12a_ICSL_discuserdata or R12aa_ICSL_discuserdata}

which states that eventually one of two transitions discarding the Data_request message is triggered.
The formula is satisfied, and the following trace is presented as explanation[33] [34]:


IRBC
ICSL
ISAI
R2 ISAI_CONNECT_request;
R2a ISAI_CONNECT_confirm;
R4 IRBC_User_Connect_indication;
R2a ISAI_CONNECT_confirm;
R2b ISAI_DISCONNECT_indication;
R2c IRBC_User_Data_request(1);
R11 IRBC_User_Disconnect_indication;
R2 ISAI_CONNECT_request;
IRBC
ICSL
ISAI

Notice that the RBC User will surely receive the disconnect indication ... but a little later.

ISAI_testing_ERdata

Another architecture/scenario of interest might be that one constitutes the real initiator SAI component and by abstract ER and RBC components. In this case we are interested to test if the

[33] We have removed from the trace the interactions with the Timer object.

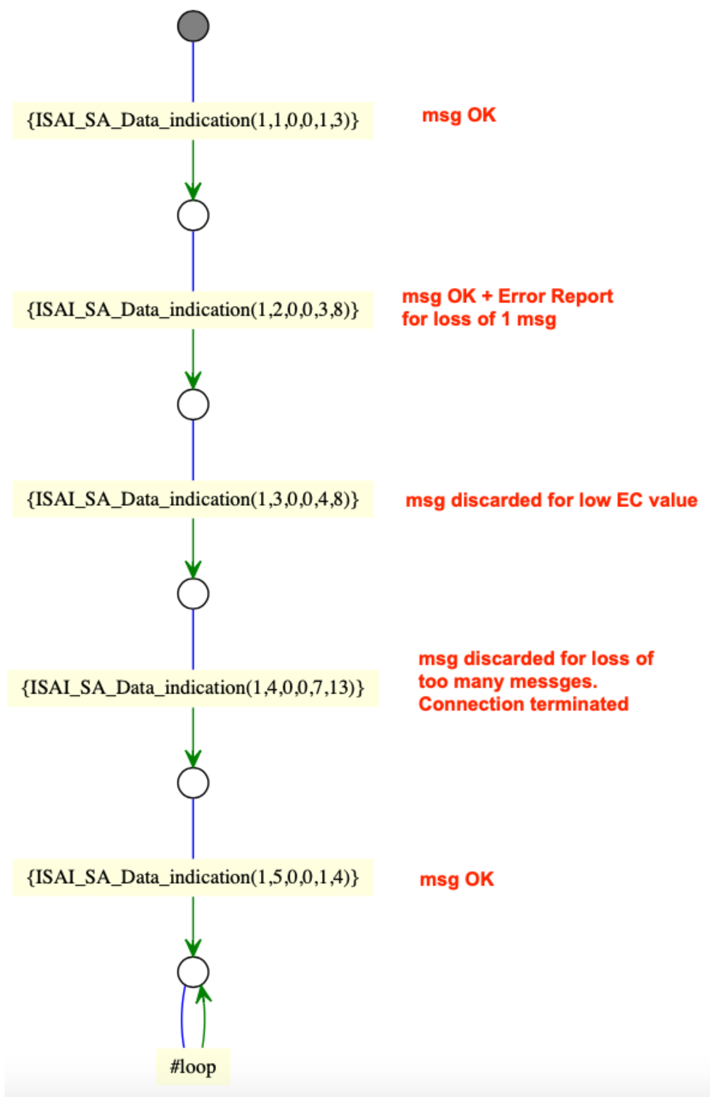
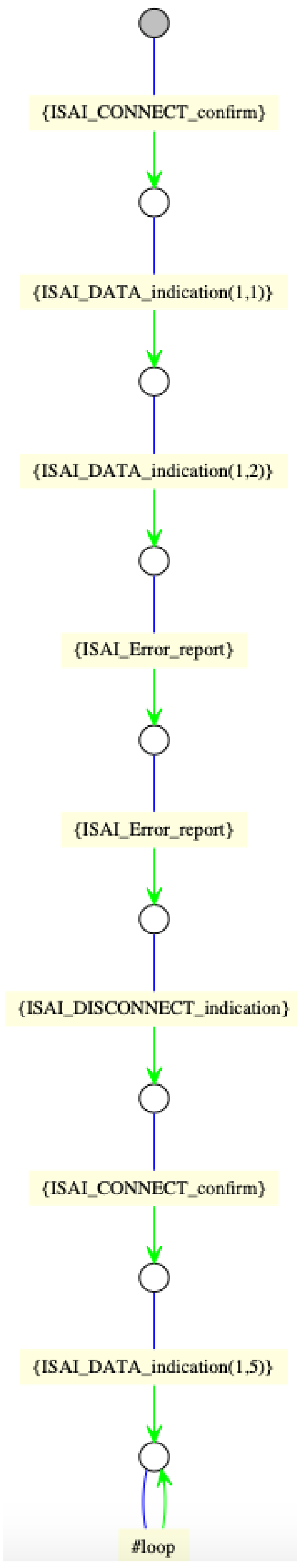
[34] Notice that no assumption prevents our "chaos" SAI model to randomly send connection confirmations.

SAI behavior in terms of protection from duplications/reordering, and excessive delay of data messages actually works as expected. In this case we have constructed an ER component that accepts and initializes connection requests from the SAI, and one connected sends a Sa_DATA.indication message with custom generated sequence and EC numbers.

We remember that the structure of Sa_DATA.indications messages arriving from the ER have the structure: ISAI_SA_Data_indication(arg1,arg2,arg3,arg4,arg5,arg6) where:

--	arg1 = message type,	arg2 = data value,
--	arg3 = ack request,	arg4 = ack response,
--	arg5 = sequence number,	arg6 = EC number

When the data flow sent by the ER Layer is, for example, the one shown in Figure 18, the corresponding data flow from SAI to CSL appear to be one shown in figure 19.


{ISAI_SA_Data_indication(1,1,0,0,1,3)}
msg OK
{ISAI_SA_Data_indication(1,2,0,0,3,8)}
msg OK + Error Report for loss of 1 msg
{ISAI_SA_Data_indication(1,3,0,0,4,8)}
msg discarded for low EC value
{ISAI_SA_Data_indication(1,4,0,0,7,13)}
msg discarded for loss of too many messges. Connection terminated
{ISAI_SA_Data_indication(1,5,0,0,1,4)}
msg OK
#loop

{ISAI_CONNECT_confirm}
{ISAI_DATA_indication(1,1)}
{ISAI_DATA_indication(1,2)}
{ISAI_Error_report}
{ISAI_Error_report}
{ISAI_DISCONNECT_indication}
{ISAI_CONNECT_confirm}
{ISAI_DATA_indication(1,5)}
#loop
Figure 18 and 19: data flows ER->SAI and SAI-CSL