Technical Informative Note 15
Preliminary progress Report
Formal development Demonstrator prototype

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 16 (31 March 2021)
Actual submission date:	31-05-2021
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*
v1	25-02-2021	first structure of content
v2	06-03-2021	first draft version with some of the contents
v3	12-04-2021	first version with most of the contents
v4	22-05-2021	fully revised version
v5	31-05-2021	final version

Report contributors		
Name	Beneficiary Short Name	Details of contribution
Franco Mazzanti	CNR	Overall Structure and Content
Dimitri Belli	CNR	Contributions in Annexes
Alessio Ferrari	CNR	Comments and suggestions
Alessandro Fantechi	CNR	Comments and suggestions
Davide Basile	CNR	Comments and suggestions
Stefania Gnesi	CNR	Comments and suggestions
Andrea Piattino	SIRTI	Comments and suggestions
Laura Masullo	MERMEC-STE	Comments and suggestions
Daniele Trentini	MERMEC-STE	Comments and suggestions

Table of Contents

1 Executive Summary

This Technical Informative Note describes the *progress of the activity* of Work Package 2 / Task 2.3 in the months 12-17 of the project 4SECURail.
The *final results* of Task 2.3 will be described in Deliverable 2.5, due at month 20 (end of July 2021). This Technical Informative Note is likely to already contain *most of the interesting results* that will appear in the final deliverable, together with other less important internal progress details that for readability issues will not appear in the final version.
The overall final purpose of the whole experimentation is the *observation of the impact*, in our *specific case,* i.e. applying our specific tools and methodologies[1] to our specific case study[2], of the adoption of *formal methods* towards the improvement of the *quality of the system specifications* under construction.

The activity of Task 2.3 builds upon a preliminary application[3] of the same demonstrator prototype to an initial fragment of the case study.
The activity performed within these first five months of Task 2.3 has been centred 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 points of this revision concern:
   - The choice to *complement* the initially selected formal method with a second one.
   - The choice to *mechanically* generate formal models from their semi-formal description.
   - The definition of a *structured logical framework* inside which to experiment the formal analysis activity.
2) A revision/refinement extension of the initial fragment of *the case study* progressing toward its full 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 Technical Informative Note we also describe some early *observations* resulting from this demonstration activity. In particular, from our perspective, the most important takeaway concern:

- The way in which UML/SysML artifacts can be effectively used as a complement to the specification of system requirements.
- The importance of *multiple, mechanical* generation of formal models of different types.
- The observation of the effective impact of semi-formal modelling and formal analysis on the identification of weaknesses in the initial natural language system requirements definition.
- The observation of the difficulties and the limits in which the exploitation of formal methods in

[1] the selected tools and methodologies constituting the "formal development demonstrator prototype" are initially defined in Task 2.1/Deliverable 2.1, and here refined,
[2] the selected case study is described in Task 2.2/Deliverable 2.3.
[3] described in Deliverable 2.2 of Task 2.1

the requirement specification phase may incur.
- The importance of a *consistent and integrated* set of rigorous natural language descriptions, UML-based semi-formal artifacts, and formal models, to consolidate the overall quality of 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 this demonstration activity from the conceptual issues to which they are related. As far as possible, all the technical details have been moved inside the Annexes.

2 Abbreviations and acronyms

Abbreviation / Acronyms	Description
CBA	Cost-Benefit Analysis
CSL	Communication Supervision Layer
EA	Enterprise Architect
EC	Execution Cycle
ER	EuroRadio
FIFO	First-In-First-Out
FM	Formal Methods
IM	Infrastructure Manager
LTS	Labelled Transition System
MAAP	Multi-Annual Action Plan
MBSD	Model Based Software/System Development
NRBC	Neighbour RBC
OMG	Object Management Group
RBC	Radio Block Centre
SAI	Safe Application Intermediate sub-Layer
SFM	Safe Functional Module
SoS	Systems of Systems
TD	Technology Demonstrator
TTS	Triple Time Stamp
UML	Unified Modelling Language
UNISIG	Union industry of signalling
WP	Work Package

3 Background

The present document constitutes a preliminary version of Deliverable D2.5 "*Formal development Demonstrator prototype*" of Task 2.3 of 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 D2.7 "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 behaviour, interoperability and safety, and standard interfaces are needed to increase market competition and standardization, reducing long-term life cycle costs."

For our purposes, the project scenario considers the Infrastructure Managers (IM) applying formal and semi-formal methods to build robust and verifiable specifications of system requirements, which will make the procurement of systems and equipment - compliant with legal requirements and needs of operators - possible and suitable for easy integration in the existing railway subsystems. This will contribute to moving 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 (this approach is supported by the Eulynx initiative [EULYNX]). In this context of modular systems, the use of formal methods is a solid support to the definition of more standard interfaces.

According to [MAAP2019], the Shift2Rail Innovation Programme 2 (IP2) will focus 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 Technology Demonstrator TD2.7 aims to contribute to the enabling of two Innovation Capabilities (IC) of the Shift2Rail Innovation Programme 2 (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 will contribute to the above Building Block achievement with the demonstration and evaluation of techniques based on formal methods to reduce life-cycle costs and improve the

global availability of the railway systems.

4 Objective/Aim

One of the objectives of the 4SECURail project is to perform a costs and benefits 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.
The objective of Task 2.3 is to exercise the process of system requirements definition that exploits the use of semi-formal and formal methods to improve the quality of the specifications written by the railway IM. The definition and overall structure of this process have been initially given in D2.1 of Task 2.1. The purpose of this deliverable is to check and apply the specification process described in D2.1 to the signalling system case study defined in D2.2 of Task 2.3.
This activity is aligned with the objective of TD2.7 [MAAP2019] *Formal Methods and standardisation for smart signalling,* which focuses 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 IM.*
This is done by observing, in our specific case, the effects of applying our *specific tools and* our *specific case study*. I.e., we take the point of view of an Infrastructure Manager that intends to define the system requirements specification document to be used in tenders, exploiting the use of formal methods for *improving the confidence* that the document *clearly* and unambiguously reflects the intentions of the designers and that the implementations eventually deriving from it will correctly *interoperate* with the other system components with which the system is expected to interact.
Notice that we are talking of two very different kinds of goals: the first one is related to the *precision* (clarity/completeness/consistency/safety) of a specific subsystem specification targeted to become a specific tender to the providers. The second one is the goal of improving the confidence that what is specified is *precisely what is needed*, i.e., something that really corresponds to the designer ideas and that which we can expect will correctly interoperate with the other components of the railway framework, which is essentially a system of systems.

The activity described in this document is strictly correlated with three previous deliverables:
- Deliverable 2.1 [D21] describes the planned structure of our formal development demonstrator process and the rationale behind it.
- Deliverable 2.3 [D23] describes the planned case study for testing the application of the formal development demonstrator process.

- Deliverable 2.2 [D22] presents a first attempt to apply the demonstrator process to an initial fragment of the case study, to gain some early experience, possibly leading to the improvement of the process itself.

The planned structure of the formal development demonstrator process initially described in D2.1 is based on two main steps:
- A first step in which, starting from the initial natural language requirements of the system, is developed a SysML operational model, where each system component is described by an appropriate UML state machine.
- A second step where the SysML model is encoded/translated into formal notation suitable for formal analysis. The initially selected formal notation is the “B" notation of the ProB tool.

The experience gained in the early application of the process to the case study fragment had led to several improvements of our demonstrator process:
- It has been recognized the actual need to rely on mechanical translations between the SysML models and the formal notations. This point is discussed in Section 5.1.2 and Annex 8.1. This mechanical translation effort has been made possible by the direct generation of SysML designs with the UMC tool bypassing the Sparx-EA modelling steps.
- It has been recognized the usefulness to take into consideration also an alternative formal modelling approach based on process algebras, which can exploit advanced state-of-art model reduction and compositional verification techniques. This point is discussed in Section 5.1.1 and Annexes 8.1 and 8.2.
- It has been recognized the need to define in a more structured way the process of performing the formal analysis steps, classifying the kind of verifications which can be exploited using the selected formal methods (push-button, model reductions, logical encoding of properties), the way in which the verification scenarios can be built, and the kind of feedback which can be expected from the process. These aspects are described in Sections 5.1.3, 5.2, 5.3, and Annexes 8.3, 8.4, 8.5, 8.6.

The initial fragment of the case study taken into consideration in D2.2 is moreover being extended towards the goal of the modelling and analysis of the full case study. This aspects are described in Section 5.2.

5.1 Improving the initial modelling and analysis process

Our demonstrator process (see D2.1) is supposed to *start* from a requirements definition document written in natural language, to *associate* to it an initial semi-formal model described in a standardized notation like SysML/UML, and finally *to generate* from it one or more formal specifications to be used for formal analysis.

5.1.1 Revisiting the formal notations used for the system design and analysis

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 interacting - 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 artifacts, as exemplified by our initial requirements in [D2.3].
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 designs. This second approach is based on the LNT [LNT] specification language of the CADP [CADP] toolset. One interesting aspect of this second 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 of help in reducing the risk of state explosion, improving the overall scalability in the analysis of the system properties.
Another interesting aspect of the CADP framework is that the structure of models of which it makes use is based on events, 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.

5.1.2 Revisiting how formal models are generated

Once we have associated our initial requirements with one (or more) operational UML models, in order to reason on all the possible behaviours of the system in a rigorous way, it is necessary to transform the UML model into a formal specification amenable to formal analysis.
Many possible target specification languages can be selected, and even once the target notation has been chosen, many different translation schemes can be adopted. This translation step may involve a further level of abstraction and approximation of the system as already occurred when firstly defining our initial UML operational model. The result is finally a system description which is based on a mathematical notation for which it is possible to express and verify properties of interest.
During the activity of Task 2.1, the translation from the UML model into the selected formal notation (B machine) has been performed *manually*, and this has proved to be a very time-consuming and error-introducing activity. Moreover, since the initial UML design is not

necessarily a stable one (as experimented in our trial application of the demonstrator), the translation effort had to be continuously repeated, raising even more the effort needed and the risk of introducing errors.
In these first months of T2.3, we have opted for the development of a tool for the automatic translation of the UML model (in the UMC format) into our selected formal notations. This has allowed greater flexibility in the maintenance of the UML models and much greater reliability of the generated encodings.
Also in the case of CADP/LNT has been developed a tool for the automatic translation of source UML models. This "diversity in formal methods application" has 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 can generate two different models using different notations and *formally* verify that the two target frameworks operate on the same model.
Actually, the formal frameworks which can be proved to agree on precisely the same model might be considered to be three, if we consider also the UMC framework, despite its prototypical status, as a legitimate tool for system analysis.
More details on the mechanical UMC to ProB and UMC to LNT translation are given in Annex 8.1.
The developed models and the source code of the developed translators are publicly accessible from [ZenodoWP2].

5.1.3 A more structured approach to the formal analysis

When first applying the formal methods demonstrator process to our initial fragment of the case study (Task 2.1/D2.2), the greatest attention has been posed to how an operational UML model of the system might have been designed, how this model might have been translated into a B-machine, how the various tools could have been composed, the role that the semiformal SPARX-EA tool might play in the process, and how the formal analysis phase on the system being specified might be carried on. In this section, we would like to discuss in a more structured way what *might be the IM interest* in applying formal methods to the process of construction / analysis of a system requirements document.

In our case, the input of the analysis has been the set natural language requirements (and associated graphics) for the various components of the case study as described in D2.3.
From this input, a semiformal operational model in terms of UML state machines has been defined, formally encoded, and analyzed.
We believe, however, that the initial system requirements designers should be provided not only with the technical artifacts in terms of the generated formal models and formal properties, but possibly also with an alternative natural language description of the various subsystems strictly connected with the structure of the formal models, that might represent the main feedback in a natural user friendly, but still rigorous, notation of the results of the formal analysis (see Fig 1).

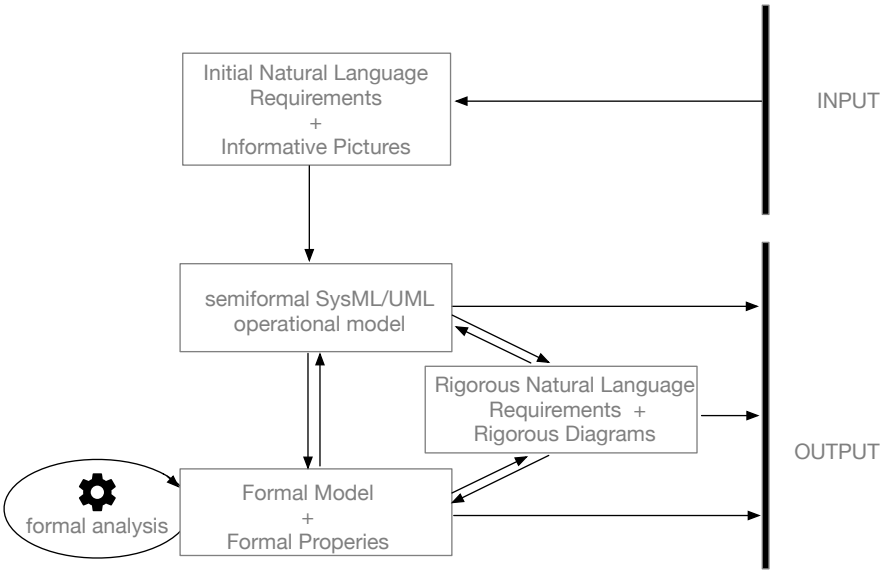


Figure 1 Input and Outputs of Formal Analysis Process

In particular, such natural language description for each system component under design should describe in a precise way:
1) - The *parametric aspects of the component*
2) - The *interface* towards the outside of the component (i.e., the messages sent and received)
3) - The *assumptions* on the external environment which underlie the component definition
4) - The *requirements* on the internal behaviour of the component
5) - The *guarantees* that the component should ensure towards the external environment.

Most of this information is somewhat already present also in our initial input requirements, but often in a not-well-structured or unclear form. This rigorously structured natural language system specification plays a relevant role during formal analysis by stating the properties that are expected to be satisfied by the system, still using the natural language, but in a form more amenable to confirmation by formal analysis.
The produced natural language feedback might be enriched with easily readable graphics (e.g. state machines, sequence diagrams), in a way similar to the graphics initially provided with the input requirements, but in this case also backed/confirmed by the underlying formal modelling and analysis effort.

In Annex 8.2 we present an example of such feedback related to the two CSL components, which have been the first target of the formal modelling and analysis during Task 2.1 and the initial part of Task 2.3. This example of feedback includes graphical overviews in terms of UML state machines of the behaviour of the CSL components.

We should note that almost all the *requirements on the internal behaviour* of the CSL component have the form:

< When a certain *event* occurs,

and certain *local conditions* hold,
then certain *effects* should occur>

and are directly reflecting the structure of the formal/semi-formal models.
I.e. for each transition in the UML design, activated under certain conditions and generating certain effects, there is a corresponding requirement that specifies precisely that relation between condition and effects without ambiguities, redundancies, or inconsistencies.
This aspect is illustrated in more detail in Annex 8.4 ("Analysing the internal behaviour of the CSL component").
In the case of the initiator CSL, the initial CSL requirements do not explicitly mention any particular *assumption* on the external environment. The description presented in Annex 8.3 is instead far more precise than the initial requirements, clarifying the expected behaviour of the communications between the components, and the expected constraint on the behaviour of the external components.
The initial natural language requirements are also not very clear about the expected *guarantees* of the CSL components towards the other parts of the system.
We can, however, exploit formal verification techniques either to extrapolate them from the model and check if they correspond to what is implicitly expected, or to formally verify if the formal model satisfies them.
The assumptions made by component A upon component B should be matched by corresponding guarantees of component B towards component A.

The different kinds of guarantees that can be associated with a system component are well exemplified by the SAI component.
From one side we have a set of expected guarantees related to the expected *high-level purpose* of the component like:

- To accept requests to *establish* or *terminate* a safe connection with the other side.
- While the connection is active, to *transmit* all the requested data upon this connection, and return all the incoming data from the connection discarding all the data that arrives with excessive delay, duplicated, or out of order.
- To communicate back the failure when such a safe connection is lost.

From the other side we have a set of more technical/behavioural guarantees that match the expected assumptions about the *interoperability* with respect to the other components like:

- Once a connection has been established, a confirmation is sent back.
- When a termination request is received, a notification is sent back.

- When a message with an excessive delay is received, the delivery of the message is replaced by the notification of an error.

It is not uncommon that guarantees of this kind are sometimes left implicit and overshadowed by the other explicit internal behavioural requirements.
These aspects are further discussed in Annexes 8.4 ("Analysing the external behaviour of the CSL component") and 8.5 ("Analysing the external behaviour of the whole CSL layer").

5.2 Refining the initial fragment of the case study

5.2.1 Overall Structure and Assumptions

Our case study (see D2.3) deals with the *communications* between two RBCs during the execution of the RBC-RBC-Handover protocol. Its overall structure is shown in Figure 2.
From the point of view of the communications, we have an "initiator" side and a "called" side. It is likely that the two RBC sides are developed by different providers, so this case study fits well the need of having a case study reflecting the point of view of an infrastructure manager interested not only in the fact that the implementation of the two sides satisfies their system requirements but also on the fact that the system requirements are *sufficiently precise* and *guaranteeing the correct interoperability* of the two delivered products.

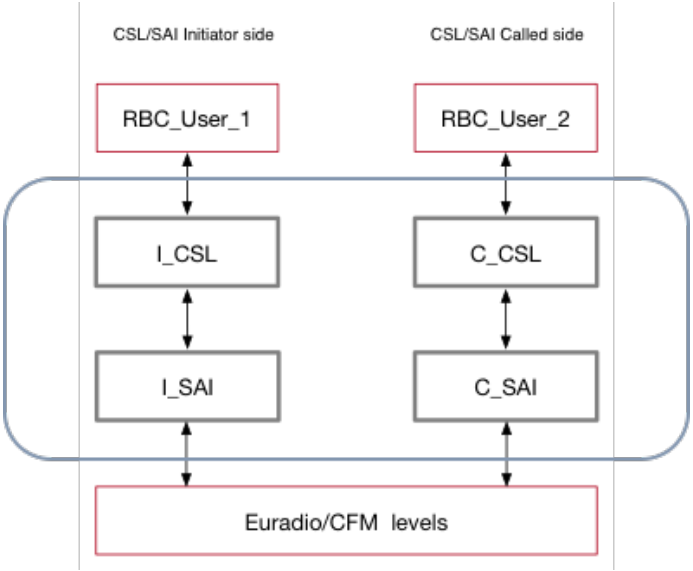


Figure 2 The signalling case study structure

In our case study we have seven logical components:
- The two sides of the "RBC" components handling the RBC-RBC-handover transactions,
- The two sides of the "Communication Supervision Layer" (CSL) responsible for the creation and supervision of RBC communications.
- The two sides of the "Safe Application Interface" (SAI) handling the creation and maintenance of the safe connection on the top of the EuroRadio layer
- The underlying EuroRadio layer abstracting the physical communication line between the two RBC sides.

Only the CSL and SAI components are the object of the requirements specifications to be analysed, for which we have an initial natural language description in [D2.3].
The RBC and EuroRadio components act as elements of the execution environment that are needed to stimulate and receive data from our system components. More than one version of these environment elements can be imagined, allowing us to model and analyse different

scenarios in which our CSL and SAI components might have to operate.

All these components will be modelled as UML state machines that execute concurrently and asynchronously, and the following assumptions are supposed to hold[4]:
- All the pairwise communications between all these state machines (EuroRadio included) are nonblocking and occurring through unbounded FIFO message queues (we have one memory buffer for each component).
- There are no delays, message loss, duplication issues in the communications between these state machines[5].
- The frequency of messages being sent by the RBC_User components is bounded.
- Every message accepted by the EuroRadio layer, if delivered, is delivered (possibly more than once) within a maximum time.

Our preliminary instantiation of demonstrator in Task 2.1 has been focused only on the two CSL subsystems, leaving the detailed modelling and analysis of the SAI components to the subsequent Task2.3.
To allow the analysis of the interactions between the CSL components, the lower SAI and EuroRadio components and the upper RBC-Users components have been simulated in Task 2.1/D2.2 by initial very approximate abstract stubs, as shown in Figure 3.
An additional "Timer" component has been added to allow an asynchronous modelling of the concurrent execution of the various components while preventing excessive variation or relative speed among them. This is one of the aspects which have been approximated in the model with respect to the real system, another being the precise structure and format of the communication messages being exchanged through the communications lines.

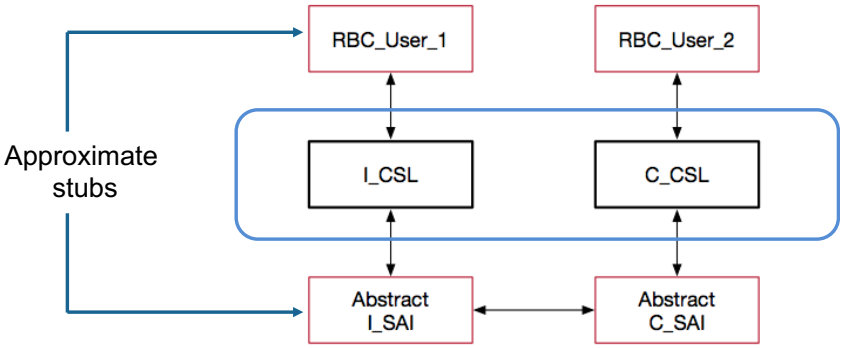


Figure 3 D2.2 modelling of the case study fragment

5.2.2 Revisiting the (initiator) CSL disconnection process

[4] In D2.2 is also mentioned the assumption that the RTC steps of the semiformal UML state machines should be considered globally atomic (i.e. non overlapping) with respect to whole system. This assumption has been removed because considered too strong.
[5] Therefore loss, duplication or delay of messages are modelled internally inside the Euroradio component.

In the (initiator) CSL design of D2.2, if the communication line is active but a receive-timeout expires (i.e., too much time is passed without receiving any message), the CSL moves to a "no communications" state (notifying the RBC of this change of status and requesting to the SAI the termination of the connection) and immediately tries to re-activate the communication line.

After further interactions with the requirements designers, it has been discovered that the current design is not what is desired. What was really intended by the designers is that the CSL, after moving to the "no communications" state, should wait for a notification from the SAI side that the connection has been terminated before trying to re-activate it.
This behavior is not explicitly requested by the CSL requirements in D2.3, and only hinted at by a sequence diagram associated with requirements, reported here as Figure 4.
We can observe in Figure 3 that the "SAI_CONN.req" message is actually sent by CSL after receiving a "SAI_DISCONN.ind". The picture "seems to suggest" that this is a mandatory behavior (not precisely prescribed in the natural language requirements) and not just one of the allowed behaviors. We have the impression that this is a nice example of both "incomplete requirements", "ambiguous diagram", and "mismatch between textual and graphical notations".
In any case, the ambiguity has been resolved in a meeting with the designers, and the semi-formal and formal designs correspondingly updated.

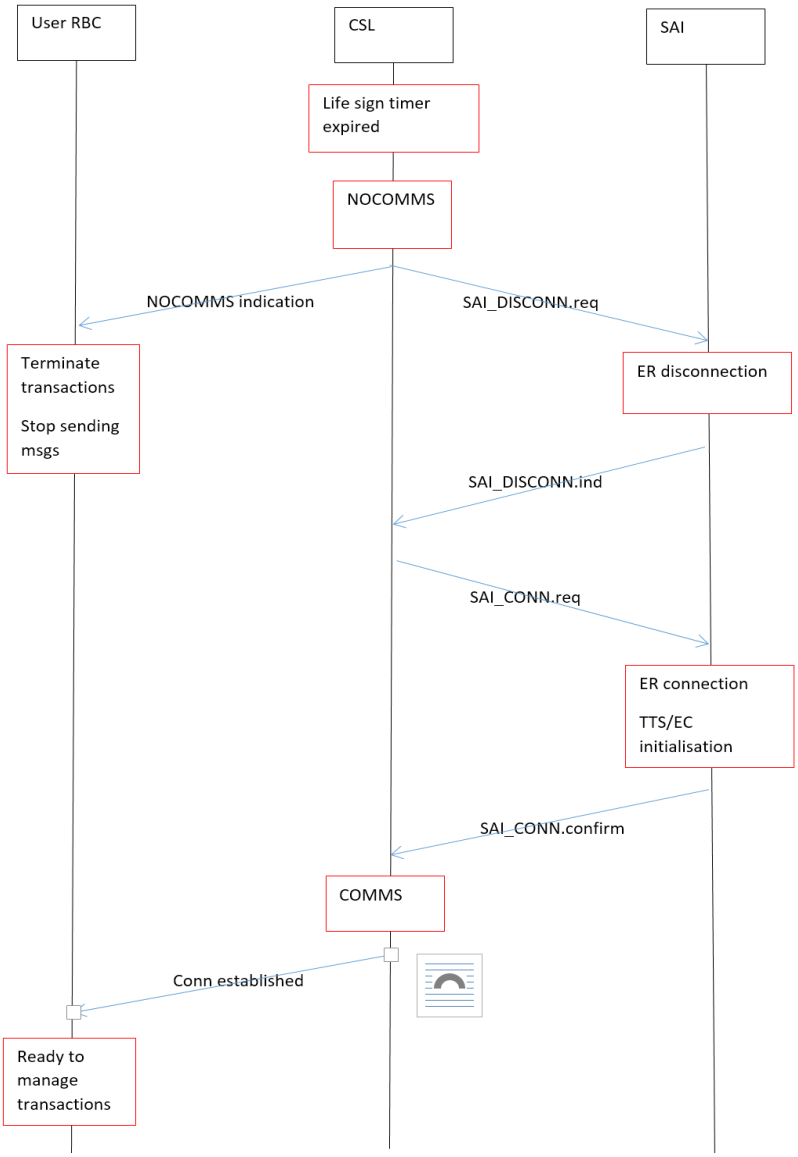


Figure 4 Loss of communication (Fig 8 from D2.3)

5.2.3 Towards the refinement of the SAI layer

In the D2.2 modelling of the case study, as shown in Figure 3, the SAI layer was represented by a pair of approximate stubs. Moving towards more detailed modelling, we have refined that design by explicitly introducing the EuroRadio component and accordingly refining the SAI models(see Figure 5). The current version of the SAI components is still an abstract one that still needs to be appropriately structured according to the detailed D2.3 requirements, activity which is still in progress. In particular, the current version still does not model the introduction of

sequence numbers and cycles count information in the messages and models just a dummy initialisation phases of the safe connections. As a consequence, the SAI component at the current stage is not able to evaluate the actual "invalidity" of a message (when it arrives with an excessive delay) or the excessive sequential loss of messages (causing the termination of the connection). These checks are therefore modelled as nondeterministic choices.
Several versions of the EuroRadio environment component can now be separately designed to stimulate the system in different scenarios.

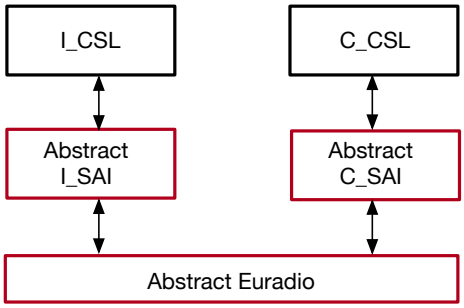


Figure 5. Second refinement of case study with explicit SAI/EuroRadio layers

5.3 Definition and analysis of the verification architectures

5.3.1 Defining the architectures

When reasoning on our CSL components, we have (at least) two ways to build verification architectures for our analysis. The first one (see Figure 6) is to build an architectures in which a single system component (CSL) interacts with abstract models of the environment (RBC and SAI), which satisfy just the minimal set of required assumptions to be consistent with the CSL design and which stimulate all the possible interactions. This kind of architecture remembers the "single component stress testing" of a module, with the difference that with model checking *all* the possible component behaviors are analysed.

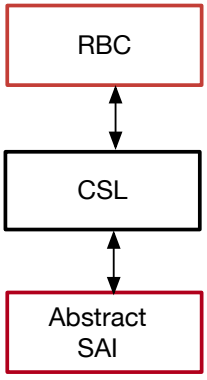


Figure 6 Testing CSL component in isolation

This kind of scenarios has the advantage of being simple and 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.
In order to analyse the interacting behaviour of the two (initiator and called) CSL components, we need more complex architectures that integrate all the needed components as shown, for example, in Fig. 3 and Fig. 7.

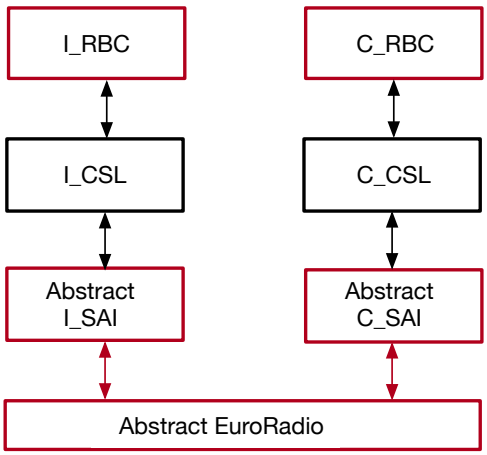


Figure 7 A full integrated scenario for the analysis of the CSL components

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. In principle, however, it is also possible to translate in the formal model only the system components of interest and take advantage of the environment modelling features supported by the specific formal framework considered. This second approach is likely to be more efficient but is also likely to be more prone to the introduction of mistakes and to require much bigger expertise in the use of formal methods. Our choice is that of modelling all the scenarios as UML/SysML designs.

The following is a partial list of some scenarios that have been so far generated and used for the first steps of model analysis. Notice that being the SAI level not fully modelled. The obtained results may give just a rough idea of the system behaviour (but already sufficient to find early design errors or inconsistences).
Notice that all the components are heavily parametric. For each kind of scenario, several versions can be given according to the actual values of the parameters being used.

SC0_nodata_V27 (e.g. 16.477.549 states, with CSL rec-limit=2, send-limit=1, conn-limit=2)
This is a fully integrated scenario, where the two RBC environment components are "silent", i.e. they do not exchange RBC_User messages but just accept incoming indications on the

communications status. Therefore, the only exchanged messages between the two CSL components are life-signs. The SAI components are still partially modelled, and do not completely represent the connection initialization phase, and do not handle the message sequence numbering and model nondeterministically the effects of delays or message loss.
The EuroRadio level introduces arbitrary delays and message loss, but no reordering or duplications, and is limited to have e memory of no more than two messages in transit from one side to the other.

SC1_nice_nodata_V27 (e.g. 47.369 states, with CSL rec-limit=6, send-limit=1, conn-limit=5)
This is a fully integrated scenario like the previous one. The main difference being the EuroRadio model, which introduces only very small delays (one time slot) and does not lose messages, and the SAI level, which does not nondeterministically reintroduce the modelling of delays or loss of messages.

Scenario SC1_niceER_irbcdata_V27 (e.g. 791.675 states, with M=2, CSL rec-lim=6, send-lim=1, conn-lim=5))
This is again a fully integrated scenario with the same "nice" EuroRadio component. In this case the initiator RBC, once received the notification of active connection, sends at most M different messages. The receiver RBC checks that no reordering in the arrived messages actually occurs.

Scenario SC2 immediateroundtrip (30.876.725 states, with CSL rec-limit=2, send-limit=1, conn-limit=2)
This is a fully integrated scenario where the initiator RBC, after receiving the notification of the activation of the communication line, sends one message and waits for a reply from the called RBC. The called RBC wait for an incoming message and replies to it.
The EuroRadio component may lose or delay messages.

ICSLtesting_V27_continuosdata: (90.751 states, with CSL rec-limit=2, send-limit=1, conn-limit=2)
This is a single component testing scenario. The initiator RBC waits for a connect_indication and, as long as connected, continuously sends the same message with a given frequency.
The abstract ISAI always accepts incoming requests, replying with DISCONNECT_indication to DISCONNECT_requests. Periodically sends to the CSL either e Life-Sign, or some RBC data, or disconnection indications or error reports.
No EuroRadio or called side components are modelled.

Scenario ICSLtesting_V26_rbcdata (350.984 states, with M=10, CSL rec-limit=2, send-limit=1, conn-limit=2)
This is single component testing scenario. The initiator RBC waits connect_indication and, as long as connected, continuously sends M different messages with a given frequency.
The abstract SAI behaves as in the previous case.
No EuroRadio or called side components are modelled.

5.3.2 Analysing the Scenarios

Once the scenario of interest has been designed and formalized, several kinds of analysis can be

carried over on it. In this section we summarise the main points, referring to Annexes 8.4 and 8.5 for a more detailed presentation.

Push Button Checks

The simplest check consists in analyzing the full statespace looking for the presence of deadlocks, violation of invariants (in our case related to the types of variables), loss of events (in our case the presence of a not-handled message causes a deadlock). This can be done in ProB with just the pushing of a button ("Model Check ..."). The same can be done in LNT and UMC by just requesting the generation of the full statespace.
This simple check already allows finding many simple encoding and design errors. When the translations towards the ProB and LNT encoding were done manually, this simple check was able to find most of the mistakes introduced by the translation. Fortunately, this whole category of errors has been later removed by exploiting the mechanical translations of the models.

Temporal Logics Encodings

A second kind of formal analysis can be done by formally evaluating the validity of properties expressed in terms of logical formulas. This task may require some more advanced knowledge of the theory behind the used formal methods.

In some cases, the encoding of a property in terms of logical formulas can be simple.
E.g. for 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 also allows (in the ProB and UMC cases) to display one of the requested execution paths in terms of a user-friendly sequence diagram usable for documentation purposes.

In some other cases, the property of interest may be directly reflecting one of the system functional requirements, and can be expressed in logical terms by composing state and event predicates. E.g. the property
"it never happens that the CSL send a data_request to the SAI when not in state COMM"
can be expressed:
in UMC as: *not EF (not inState(ICSL_COMM) and <ISAI_DATA_request>)*
in ProB as : *not F (not {ICSL_STATE=COMM} and [R8_ICSL_saidatareq])*
in LTL: *not expressible.*
In most cases like this one, however, using model checking for verifying the property can be just overkilling, because just a plain observation of the CSL state machine diagram (see Fig. Z in

[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

Annex 8.3.1) allows to easily check that the only transition triggering an ISAI_DATA_request is the transition R8 which has COMM as source state.

In further cases, the property of interest cannot be directly associated with the structure of the model, and its encoding can require rather advanced formalization capabilities.
E.g. let us consider the property:
"the messages received by the called RBC contain a continuously growing value" (i.e. no reordering occurs). This can be expressed
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.*

Explicit Observers
In many cases, like the one mentioned above 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, in the scenario **Scenario SC1_niceER_irbcdata_V27** previously mentioned. In this case, we just need to define a (called) RBC_User environment element which saves the last value received and compared it with the current one 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 possible solves the problem of hard to generate and logical properties, and also the problem of ensuring that the given encoding is actually the correct one (encoding a complex logical property can really be a very error-prone task).

Minimized information flows

While reasoning on the possible behaviour of a system component, it is sometimes useful to just 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 the formal description of the whole system (in a given scenario), it is possible to mechanically extract and visualize all the possible streams of this kind, as briefly described in Annex 8.2 (Model reduction techniques). More specifically, if, in the context of the scenario **ICSLtesting_V27_continuosdata**, we request to visualize all the possible streams of

CONNECT, DISCONNECT and DATA indications, we obtain the picture shown in Fig. 8.

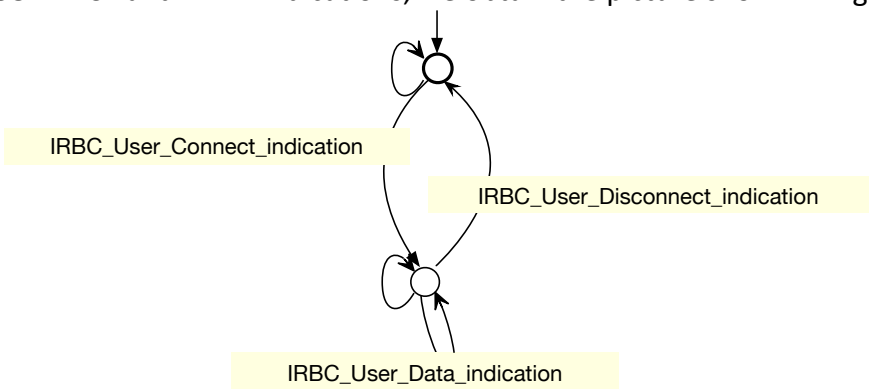


Figure 8: all the possible messages flows from CSL and RBC (initiator side)

By just observing this picture, several properties of the scenario can be observed.
E.g:
- 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 extraction of the description of the possible sequences of interest (model reduction) 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. Further examples are shown in Annex 8.5.

5.4 Observations and considerations

In this section, we summarize the "takeaways" that we have observed and found relevant during this first part of the demonstration activity.

"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 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 simpler way on specific aspects of the system, getting useful insights and confirmations or counterexamples about the expected behaviour 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 abstraction that underlie the model design. Moreover, we should not forget that the designed 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 from which verifiable formal specifications are derived.
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 on which we are interested?". The answer to the question in parts depends on the available verification functionalities provided by the selected formal framework, in part depends on the various steps of abstraction and approximations performed from the initial system requirements, but also depends on the "correctness" of the translation and encoding of the model into the formal specification notation.

"The real goal of our demonstrator"
Once we have associated our initial requirements with one or more UML models, translated them into rigorous formal specifications, and started making rigorous analysis upon them, our realistic goal cannot be the complete "validation" of the initial system requirements or a generic "proof of correctness" of the formal design.
The realistic 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. Moreover, considering the role of the demonstrator inside the whole project, we can see that in our case we are not much interested in answering *all* the questions that can be formally checked on our signalling systems, but rather show the *kind* of question we can check, the *kind* of answer we can obtain, the *difficulty* of the process and the *kind* of feedback returned to the user by this activity.

"System requirements definition and analysis is very different from system implementation "
Notice that the activity of transforming the designer intentions into a system requirements document is an activity that is very different from the activity of taking a system requirement document and developing from it an executable system. In particular, the role that formal and semi-formal methods play is very different in these two different 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 will be focused on the guarantee of 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:
- 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 the 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 eventually cause a failure of the system.

"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 that might have a very strong impact on the expected behaviour of the system. Our demonstrator has shown that this problem can be overcome if:
- We take care of explicitly stating all the otherwise hidden assumptions in the design.
- We possibly restrict the use of UML feature 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.

"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 mandatory for any serious exploitation of formal methods from semiformal UML designs. 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 MBSE framework (PTC, Sparx-EA, IBM-Rapsody, ...) but a *human-oriented, vendor-independent* textual description of the system[8]. We consider the absence of a standard OMG definition for such a human-oriented platform-independent textual representation of the models to be another weakness of the current OMG standardization activity.

"Inadequacy of existing MBSE frameworks to support formal analysis of system requirements"
With the experience of the demonstrator gained so far, the role of the selected Sparx-EA MBSE platform is limited to providing some help in the generation of readable, well-formatted documentation. From the point of view of usability towards purposes of modelling high-level requirements and performing a rigorous analysis or verification of them, this MBSE platform, despite its "animation capabilities", has resulted rather useless. The situation is likely to be the same also 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.

[8] Indeed, 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, and 2) it is not a human oriented format usable to directly communicate in a simple, textual, easily reusable way a model design.

"There are many degrees in which semi-formal and formal methods can be exploited"
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 by the initial attempt to generate a *semi-formal 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 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 play with it. However, it is not necessary to become 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 behaviour at the interfaces, that may greatly improve the confidence on the design with a relatively low effort.

"Usefulness of formal methods diversity"
Another confirmation that we have had from our demonstration activity is that the "diversity of approaches" in formal modelling and verification improves the flexibility of the analysis and the reliability of the results. Many errors in the translation programs have been quickly put in evidence when different behaviours and different statespaces resulted from the translation of the UML model into the different ProB / LNT / UMC notations. And different point views can be exploited 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 to the semi-formal and formal models of the systems a rigorous, clear, well-structured natural language description that communicates in a natural way to the developers the intended internal behavior of the system, the properties that it is supposed to guarantee to the other components, and the assumptions about the other components behavior on which it depends.

"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 - still preliminary - fragment of case study has already 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; indeed we had to resort to the definition of specific scenarios upon which we have made our analysis, e.g. fixing values for the various parameters of the system components. But this analysis does not cover the full range of system configurations.

- Interfaces with wide-range data values: When is system is composed by subsystem 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.

6 Conclusions

The choice of the selected case study, and the choice of the 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.
The selected fragment of the case study chosen in Task 2.1 appears to already have sufficient complexity to have stimulated most of the results that can likely be expected from the full case study, whose analysis is still in progress.
The current picture that starts to arise from the experience of applying the 4SECURail form methods demonstrator process to our case study is the following:

1) UML designs, if not associated with explicit descriptions of all the implementation-dependent aspects (e.g. those related to the communications between state machines), are not rigorous specifications and constitute a *dangerous* ground allowing implementations to adopt different interpretations that may result, in the end, in interoperability and validation problems. On the contrary, a *constrained* use of UML leading to a clear and rigorous operational model of the system can already be of great help in reducing the defects in a system specification. This constrained use of UML can be the base for *simple, mechanical, translations* from the UML designs to various, different formal notations, allowing the observation and the analysis of the system from different points of view.

2) Formal analysis of the UML designs may be of help in discovering consistency problems, incompleteness issues, or aspects that do not reflect the behaviour actually intended by the designer. Formal analysis can be applied with different levels of expertise, efforts, and gains.

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.
[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/
[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, Fŕ ed́ 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" https://www.omg.org/spec/UML/About-UML/
[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.
[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

8 Annexes

8.1 Model transformations

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 [LNTR] models.

The overall schema of the translation from UMC to Prob (*umc2prob*) has already been presented in Section 5.3.3 of Deliverable D2.2 of the project. We remember that the adopted translation schema was based on the construction of a single Prob state machine embodying the behaviour of all the state machines included in the UMC model. Each transition of each UMC state machine was translated into a Prob OPERATION, whose name was given by the corresponding UMC transition label. All the UMC state-machines event pools, modelled as FIFO buffers, are mapped into local variables of the Prob state machine, and all operations are conditioned on, retrieve and push data on them as shown below:

UMC transition

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

Event-B 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;
```

The schema adopted for the translation into the LNT language (*umc2lnt*) is instead quite different. In this case, 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
  ...
  ...
Behaviour
  ...
R4_ICSL_userconnind
NOCOMMSconnecting -> COMMS
 { SAI_CONNECT_confirm /
   RBC_User.IRBC_User_Connect_indication;
   receive_timer := 0;
   connect_timer := max_connect_timer;
   send_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
         RBC_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
```

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: e.g.

- *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).

8.2 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.

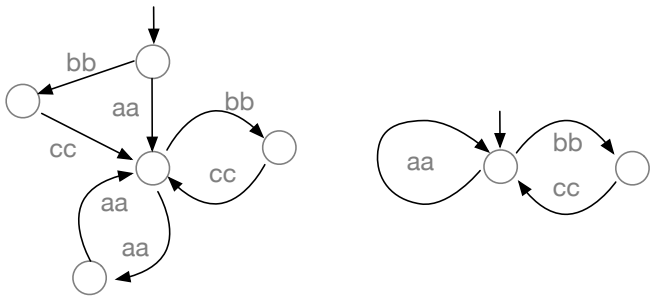
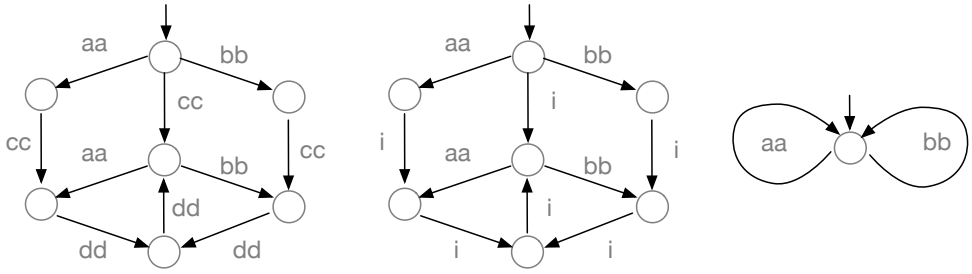


Figure 9 two strongly equivalent LTS

The nice property of *strong* equivalence [STRONG] is that the two behaviours 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[9].

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.



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.

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

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:

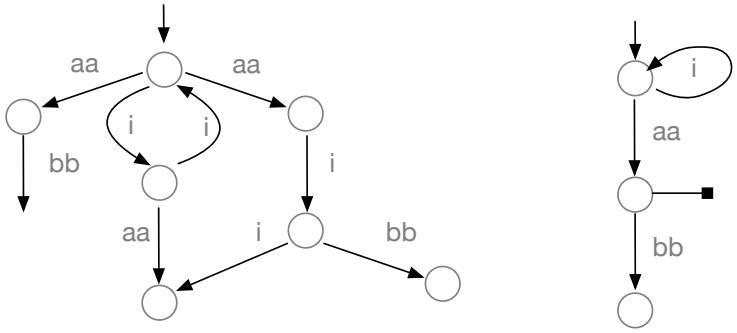


Figure 10: example of complete-div-sensitive-weak trace minimization

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.

8.3 Towards a CSL specification based on the (semi)formal models

As shown in Figure 1, in our case study we have a pair of CSL components, one on the so-called initiator side (I_CSL) and one on the called side (C_CSL).

CSL-layer high level goals

The pair of CSL components have the task to create and maintain an active communication line, using the SAI interface, that can be used by the RBC to handle the RBC to RBC handover protocol.
When the communication line is active, the two CSL forward NRBC data messages received from the local RBC to the SAI (without loss, duplication, reordering, or delay) for final delivery to the other RBC.
When the communication line is active, the two CSL forward NRBC data messages received from the other RBC through the SAI (without loss, duplication, reordering, or delay) to the local RBC.
When the communication line gets lost, the I_CSL takes the initiative to restore it as soon as possible.

Both CSL keeps the RBC informed about the changes in the status (active / non-active) of the communication line.

8.3.1 Specification of the Initiator CSL component

Configuration

The I_CSL has the following parameters:
- a connection timeout;
- a send timeout;
- a receive timeout.

External Interactions:

The **I_CSL** can receive from the Initiator RBC the following messages:
- RBC_User_Data_request (userdata);

and can send to the Initiator RBC the messages:
- RBC_User_Connect_indication;
- RBC_User_Disconnect_indication;
- RBC_User_Data_indication(userdata).

The **I_CSL** can receive from the Initiator SAI component the following messages:
- SAI_Connect_confirm;
- SAI_Disconnect_indication;
- SAI_Data_indication(saidata);
- SAI_Error_report;

and can send to the Initiator SAI the following messages:
- SAI_Connect_request;
- SAI_Disconnect_request;
- SAI_Data_request (saidata).

CSL states

The **I_CSL** can be in two main states:
- COMMS, when the communication is active;
- NOCOMMS, when the communication is inactive.

the NOCOMMS state of the I_CSL contains three substates:
- NOCOMMSwait;
- NOCOMMSready;
- NOCOMMSconnecting.

external assumptions

- I_ISAI must always respond with DISCONNECT_indication to a DISCONNECT_request.
- The frequency of messages being sent from I_RBC component to I_CSL and from I_SAI to

I_CSL is limited by an upper bound.

- All the communications between RBC CSL and SAI are nonblocking and occurring through unbounded FIFO message queues (we have one memory buffer for each component).
- There are no delay, message loss, duplication issues in the communications between these components.

behavioural requirements

R1: At startup, the I_CSL is in NOCOMMSready state.

R2: When in NOCOMMSready state, the I_CSL immediately forwards a SAI_Connect_request to the SAI, moves to NOCOMMSconnecting state, and starts the connection timer.

R3: When in NOCOMMSconnecting state and the connection timer expires, the I_CSL moves to NOCOMMSready state.

R4: When in NOCOMMSconnecting state is received a SAI_Connect_confirm, the I_CSL moves to COMMS state and starts both a send timer and a receive timer.

R5: When in COMMS state the receive timer expires, the I_CSL moves to NOCOMMSwait state and forwards both a SAI_Disconnect_indication to the SAI and a RBC_User_Disconnect_indication to the RBC_User.

R6: When in NOCOMMSwait state is received a SAI_Disconnect_indication, the I_CSL moves to NOCOMMSready state.

R7: When in COMMS state the send timer expires, the I_CSL forwards a SAI_Data_request(saidata) with a life-sign to the SAI.

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

R9: When in COMMS state is received a SAI_Data_indication(saidata) with RBC_User data, the I_CSL forwards an RBC_User_Data_indication(userdata) with such user data to the RBC_User and restarts the receive timer.

R10: When in COMMS state is received a SAI_Data_indication(saidata) with a life-sign, the I_CSL restarts the receive timer.

R11: When in COMMS state is received a SAI_Disconnect_indication, the I_CSL moves to NOCOMMSready state and forwards a RBC_User_Disconnect_indication to the RBC_User.

R12: When a message is received in a condition for which the previous requirements do not specify the required behaviour, the message is discarded.
In particular, this happens when:

- RBC_User_Data_request (userdata) is received and I_CSL is not in the COMMS state.

- SAI_Connect_confirm is received and I_CSL is not in the NOCOMMSconnecting state.
- SAI_Disconnect_indication is received in NOCOMMSready or NOCOMMSconnecting state.
- SAI_Data_indication is received and I_CSL is not in the COMMS state.
- SAI_Error_report is received.

An overview of the I_CSL behaviour UML in state-machines terms is shown in Figure 11: the gray labels associated with the transitions directly match the corresponding behavioural requirement.

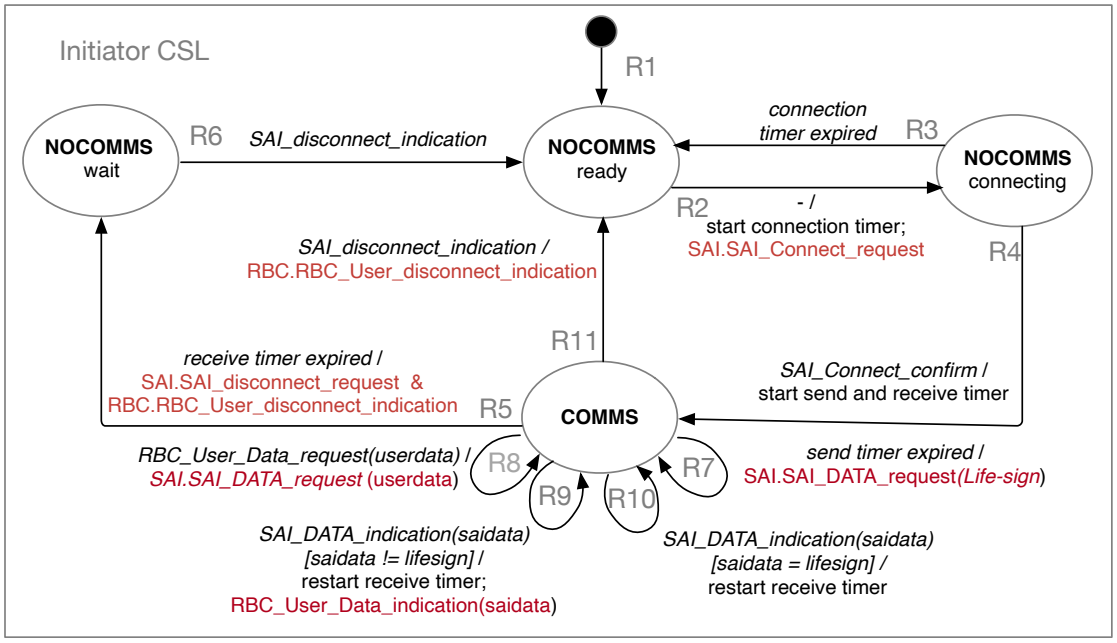


Figure 11: Overview of the I_CSl state machine

CSL external guarantees

- The frequency of messages being sent by I_CSL to I_RBC is limited by an upper bound.
- The frequency of messages being sent by I_CSL to I_SAI is limited by an upper bound.
- I_CSL sends to RBC_Data_indication message only after an RBC_Connect_indication not followed by RBC_Disconnect_indication.
- I_CSL sends to RBC a RBC_Disconnect_indication message only after a RBC_Connect_indication message not already followed by RBC_Disconnect_indication.
- The first message (possibly) send to I_RBC is a RBC_Connect_indication message
- ICSL sends to I_RBC a RBC_Connect_indication message only as first messages or after a RBC_Disconnect_indication not already followed by RBC_Connect_indication.
- I_CSL periodically sends to I_SAI either SAI_Connect_request or SAI_Data_request messages.
- ...

8.3.2 Specification of the Called CSL component

Configuration

The C_CSL has the following parameters:
- a send timeout;
- a receive timeout.

External Interactions:

The **C_CSL** can receive from the Called RBC the following messages:
- RBC_User_Data_request (userdata);

and can send to the Called RBC the following messages:
- RBC_User_Connect_indication;
- RBC_User_Disconnect_indication;
- RBC_User_Data_indication(userdata).

The **C_CSL** can receive from the Called SAI the following messages:
- SAI_Connect_indication;
- SAI_Disconnect_indication;
- SAI_Data_indication(saidata);
- SAI_Error_report.

and can send to the Called SAI the following messages:
- SAI_Connect_request;
- SAI_Disconnect_request;
- SAI_Data_request (saidata).

States

The **C_CSL** can be in two states:
- COMMS, when the communication is active;
- NOCOMMS, when the communication is unactive.

External assumptions

- The frequency of messages being sent from C_RBC component to C_CSL and from C_SAI to C_CSL is limited by an upper bound.
- All the communications between RBC CSL and SAI are nonblocking and occurring through unbounded FIFO message queues (we have one memory buffer for each component).
- There are no delays, message losses, duplication issues in the communications between these components.

Behavioural requirements

R1: At startup, the C_CSL is in NOCOMMS state.

R2: When in NOCOMMS state is received a SAI_Connect_indication, the C_CSL moves to COMMS state and starts both send timer and receive timer.

R3: When in COMMS state is received a RBC_User_Data_request(userdata) with data, the C_CSL forwards a SAI_Data_request(saidata) with such data to the SAI.

R4: When in COMMS state the send timer expires, the C_CSL forwards a SAI_Data_request (saidata) with a life-sign to the SAI.

R5: When in COMMS state is received a SAI_Data_indication(saidata) containing RBC_User data, the C_CSL forwards a RBC_User_Data_indication(userdata) to the RBC and restarts the receive timer.

R6: When in COMMS state is received a SAI_Data_indication(saidata) containing a life-sign, the C_CSL resets the receive timer.

R7: When in COMMS state is received a SAI_Disconnect_indication, the C_CSL moves to NOCOMMS state and forwards a RBC_User_Disconnect_indication to the RBC_User.

R8: When in COMMS state the receive timer expires, the C_CSL moves to NOCOMMS state and forwards a SAI_Disconnect_request to the SAI and a RBC_User_Disconnect_indication to the RBC_User.

R9: When in COMMS state is received a SAI_Connect_indication, the C_CSL restarts both send timer and receive timer.

R10: When a message is received in a condition for which the previous requirement do not specify the required behaviour, the message is discarded.
In particular, this happens when:

- RBC_User_Data_request is received and C_CSL is not in the COMMS state.
- SAI_Disconnect_indication is received in NOCOMMS state.
- SAI_Data_indication is received and I_CSL is not in the COMMS state.
- SAI_Error_report is received.

An overview of the C_CSL behaviour UML in state-machines terms is shown below:

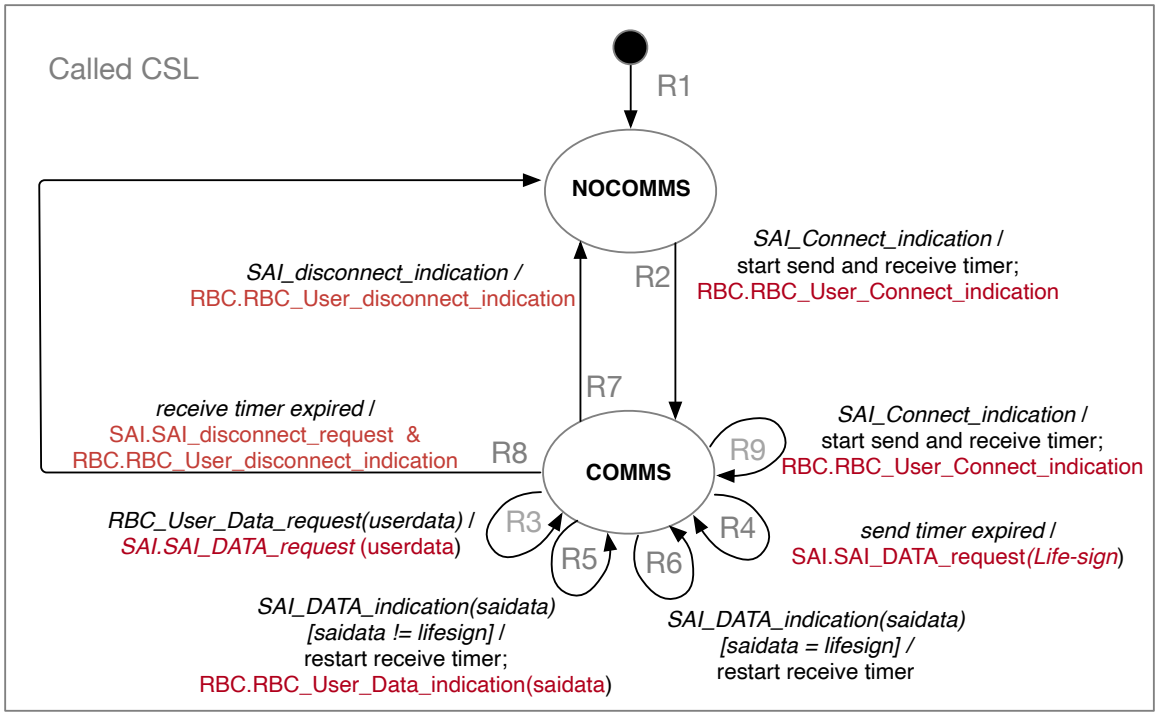


Figure 12: Overview of the C_CSl state machine

External guarantees

- The frequency of messages being sent by C_CSL to C_RBC is limited by an upper bound.
- The frequency of messages being sent by C_CSL to C_SAI is limited by an upper bound.
- C_CSL sends to RBC RBC_Data_indication message only after an RBC_Connect_indication not followed by RBC_Disconnect_indication.
- C_CSL sends to I_RBC a RBC_Disconnect_indication message only after a RBC_Connect_indication not already followed by RBC_Disconnect_indication.
- The first message (possibly) send to I_RBC is a RBC_Connect_indication message
- C_CSL sends to RBC a RBC_Connect_indication message only as first messages or after a RBC_Disconnect_indication not already followed by RBC_Connect_indication.
- ...

8.4 Analysing the internal behaviour of the CSL component

The process of creating the operational semi-formal UML model of the system is an incremental, manual, error-prone activity. The presence of advanced static analysis features in the UML design tool surely helps in reducing most of the trivial manually introduced mistakes. Consistency errors like missing requirements or conflicting requirements can only be detected by more advanced formal reasoning of the system behaviour.
We have the possibility to mechanically translate the semiformal model into a formal one and perform on it all the analysis supported by the target framework.

8.4.1 Defining the scenarios

In order to construct a verifiable system, we have to compose the ICSL component under analysis with other "testing" components that provide the necessary stimuli and handle the returned messages. The composition of the system component with the testing components is called a verification scenario, and they can have a growing degree of complexity.
The simplest scenario under which the I_CSL has been tested is the scenario *ICSLtesting_V27_nodata* in which the RBC_User just accepts incoming ICSL connect/disconnect indications without never sending data messages. On the SAI side, this scenario is based on a "chaos-like" SAI component that sends to ICSL any possible message (error_reports, disconnection indications, lifesign messages, connection confirmations) except RBC_user data. This scenario represents the case in which the two RBC do not have any data to exchange, but the system still has the task to create, check and keep alive the communication line. This scenario is very small, as it contains just 5.224 states but is useful during the first step of prototyping.

The second scenario of interest is the one in which the Initiator RBC periodically (but forever) sends a message to be delivered to the other side (through the SAI component), and the SAI component adds the sending of RBC_User messages to its "chaos-like" behaviour. In this case (*ICSLtesting_V27_continuosdata* scenario) the messages being sent are supposed to be all equal just to avoid the otherwise inevitable state explosion.

Finally, we have generated a third scenario (I*CSLtesting_V27_incrdata*) in which the messages being sent from the RBC and the SAI are all different and contain a numerical value that is incremented at every sending operation. This allows to perform verifications on the possible evolutions of specific messages (e.g. to check the non-occurrence of reordering). This is the most interesting scenario, but to limit the state explosion we need to put a limit also to the number of messages which are exchanged. Increasing the number of messages, however, does not seem to add any greater depth to the analysis of the system. The following table summarizes the sizes of the various scenarios.

ICSLtesting_V27_nodata
receivetimeout=2, sendtimeout=1, connecttimeout=2 States: 5224
receivetimeout=.., sendtimeout=.., connecttimeout=.. States: ...

ICSLtesting_V27_continuosdata
receivetimeout=2, sendtimeout=1, connecttimeout=2 States: 90.751
receivetimeout=.., sendtimeout=.., connecttimeout=.. States: ...

I*CSLtesting_V27_incrdata* (receivetimeout=2, sendtimeout=1, connecttimeout=2)
3 messages each for RBC and SAI states 801.712

5 messages each for RBC and SAI	states	2.253.934
10 messages each for RBC and SAI	states	9.060.669

8.4.2 Standard checks

The first easy checks to be performed are some default tests like the absence of runtime errors (e.g. caused by numerical calculation, operations on lists), absence of events for which no rule specifies the appropriate management, absence of deadlocks.
These checks can be activated:
- In ProB by just selecting the default "Verify -> Model_Check..." command. (see the left side of Figure ZZ).
- In CADP by just generating the statespace and using the command "bcginfo", or by evaluating on-the-fly the formula "[true*] <true> true".
- In UMC with the command "umcstats" or evaluating the formulas "EF FINAL" and "EF {lostevent}".

In case of failure of these tests it is possible to observe the execution trace that leads to the failure (and in the case of Prob and UMC also visualize it as a message sequence chart).
We have observed that already these simple checks allow us to detect most of the errors present in the UML model.

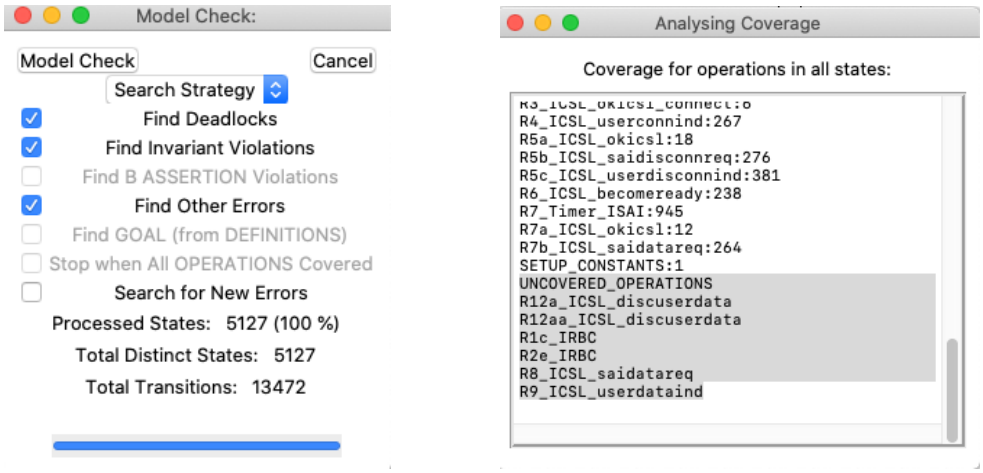


Figure 13: a result of standard ProB checks.

Another rather standard check is the analysis of the coverage of the UML transitions (i.e. the coverage of the Prob operations). 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 13 (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.

8.4.3 Double-check of the behavioural requirements

Let us now consider the behavioural requirements of this I_CSL component, as specified in the revised specification shown in Annex 8.2. We know that the behavioural requirements have been "extracted" from the formal model and should precisely reflect its internal structure. Would it make sense to try to encode these requirements as logical formulas for further verification?
Let us consider, for example, requirement **R11**:

R11: *When in COMMS state is received a SAI_Disconnect_indication, the I_CSL moves to NOCOMMSready state and forwards a RBC_User_Disconnect_indication to the RBC_User.*

This requirement corresponds to the operation *R11_ICSL_userdisconnind* of the UMC model:

```
R11_ICSL_userdisconnind:
  COMMS -> NOCOMMSready
     {ISAI_DISCONNECT_indication /
        RBC_User.IRBC_User_Disconnect_indication;
        receiveTimer := 0;
        sendTimer := 0; }
```

It would seem rather evident that this requirement is going to be satisfied by just performing a double-check in terms of code inspections. And this check is independent of the specific values of the parameters for this component.
Nevertheless, we might still be interested in checking this fact again, e.g. in one of the specific scenarios described in Section 8.3.2, for example in scenario *ICSLtesting_V27_nodata*.

The above requirement **R11** can indeed be encoded as a Prob LTL property like:

```
"G({size(ICSL_buff)>0 & first(ICSL_buff)=ISAI_DISCONNECT_indication & ICSL_STATE=ICSL_COMMS}
    =>
   (e(R11_ICSL_userdisconnind) U ([R11_ICSL_userdisconnind] & X{ICSL_STATE = ICSL_NOCOMMS}))
```

This encoding *cannot* be done in a mechanical way because the ProB encoding is at a more detailed/specific level than the requirement; a detailed knowledge of the formal model is needed.
E.g. the condition "*message received*" should become "*the message is in the first position in the buffer*"; the consequence of this condition is that the *operation* triggering the state transition should be enabled `(e(R11_ICSL_userdisconnind)`, and remain enabled until the operation is finally executed `[R11_ICSL_userdisconnind]`; after that, the state should become the one requested `X{ICSL_STATE= ICSL_NOCOMMS}`.

Another requirement that can easily be verifying to hold, either by code inspection or by formal proof in some scenario is requirement **R8**.

Let us now consider the richer scenario *ICSLtesting_V27_continuosdata*.

In this scenario, we might want to double-check if the requirement **R8** holds (this requirement was trivially satisfied in the previous scenario because there was no data exchange at the RBC level).

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

```
"G ({size(ICSL_buff) > 0 & first(ICSL_buff)=IRBC_User_Data_request & ICSL_STATE=ICSL_COMMS}
   => (e(R8_ICSL_saidatareq) U ([R8_ICSL_saidatareq] & X{ICSL_STATE = ICSL_COMMS})))"
```

The encoding of this formula is similar to the previous one, and its evaluation, despite the larger size of the model, can be completed in a few seconds after the model statespace has been generated.

Verifying the same requirement **R8** in scenario *ICSLtesting_V27_incrdata (3msgs)* is more complex because we have to express also the fact that the RBC data, whichever it is, remains unchanged until the operation `R8_ICSL_saidatareq` is triggered. So far, we have not been able to find a precise encoding for this property, which can instead be expressed in UMC and CADP exploiting logics supporting actions with parametric values or parametric fix-points.

Despite the complexity of the task of encoding system properties in formal terms of temporal logical formulas, a possible reason for trying (as far as possible) this effort might be their inclusion inside the ProB model as explicit LTL assertions.
This would allow, in case of refinements of the initial ProB model into even more detailed state machines, to continue to verify the conformity of the refined model to the initial behavioural requirements.

In the case of ProB, the encoding and verification of internal behavioural requirements might perhaps be easier if instead of modelling a UML-based scenario composed of three state machines we considered a scenario composed by the ICSL state machine alone, removing the buffer-related operations from it, and adding CSP processes as stimulating external environment.
A similar approach is also feasible in the case of CADP.
So far, however, these possibilities have not been explored further.

8.5 Analysing the external behaviour of the ICSL component

Once we are confident that the operational model correctly reflects the intended internal behavioural requirements, we might proceed in verifying that the stated behavioural requirements (i.e. the formal model) imply the stated external guarantees of the system component.

Let us consider again the scenario *ICSLtesting_V27_continuosdata,* which is rather complete but still of a small size.

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 more appear as reachable.

One way to observe the external behaviour 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. The result of the process can be carried out within the CADP framework[10] with the SVL script shown in Figure 14, is given in Figure 15.

```
% 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

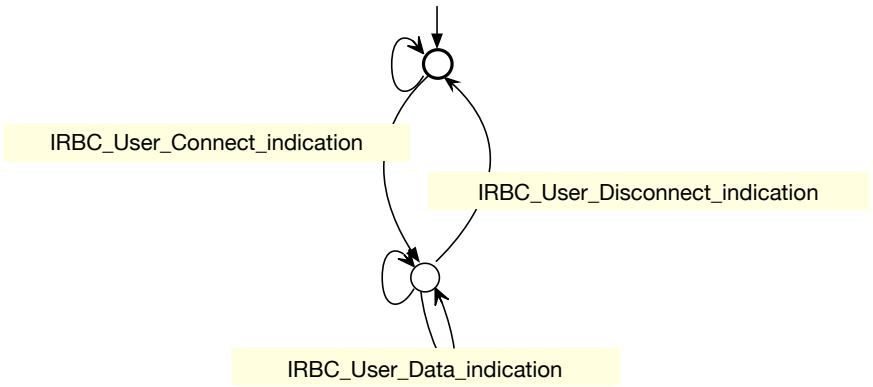


Figure 15: A ICSL->RBC message flow in the scenario *ICSLtesting_V27_continuosdata*

[10] The same result can be obtained using the *umc2aut* and *ltsconvert* tools from the free KandISTI /UMC and mCRL2 frameworks.

We can see how this simple picture shows the satisfaction of several ICSL external guarantees (see Sect 8.1.3) like:

- ICSL sends to I_RBC a RBC_Data_indication message only after a RBC_Connect_indication not followed by RBC_Disconnect_indication.
- ICSL sends to I_RBC a RBC_Disconnect_indication message only after a RBC_Connect_indication not already followed by RBC_Disconnect_indication.
- The first message (possibly) send to I_RBC is a RBCConnect indication message
- ICSL sends to I_RBC a RBC_Connect_indication message only as first messages or after a RBC_Disconnect_indication not already followed by RBC_Connect_indication.

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.

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.

For example, if we want to directly check in this scenario whether is it true that, e.g. from the IRCB side, *after receiving a connect_indication, it is necessary to receive a disconnect_indication before receiving a second connect_indication*, this property can be verified in UMC with the formula:

```
AG [IRBC_User_Connect_indication]
     A[{not IRBC_User_Connect_indication} W {IRBC_User_Disconnect_indication} ]
```

The same property can be verified in ProB with the model checking of the LTL formula:

```
G ([R8_ICSL_userconnind] =>
     X((not [R8_ICSL_userconnind]) W
        ([R16_ICSL_userdisconnind] or [R17b_ICSL_userdisconnind])) )
```

And the same can be verified with CADP using the formula:

```
not <(true)*.
   IRBC_USER_CONNECT_INDICATION.
   (not IRBC_USER_DISCONNECT_INDICATION)*.
   IRBC_USER_CONNECT_INDICATION> true
```

Suppose 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.

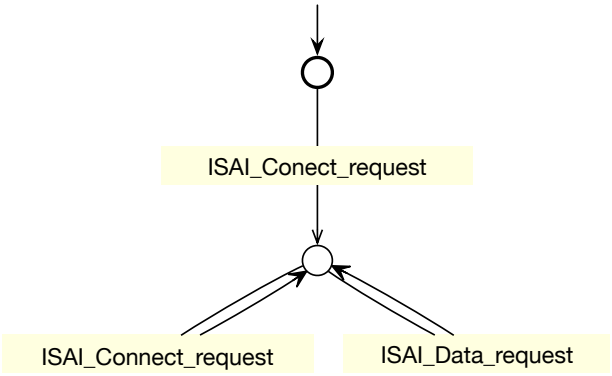


Figure 16: A ICSL->SAI message flow in the scenario *ICSLtesting_V27_continuosdata*

As we can easily see, there is no unlabelled 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])
```

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 misordering 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 lifesign, or connect/disconnect events). Figure 17 shows the generated flow of such messages.

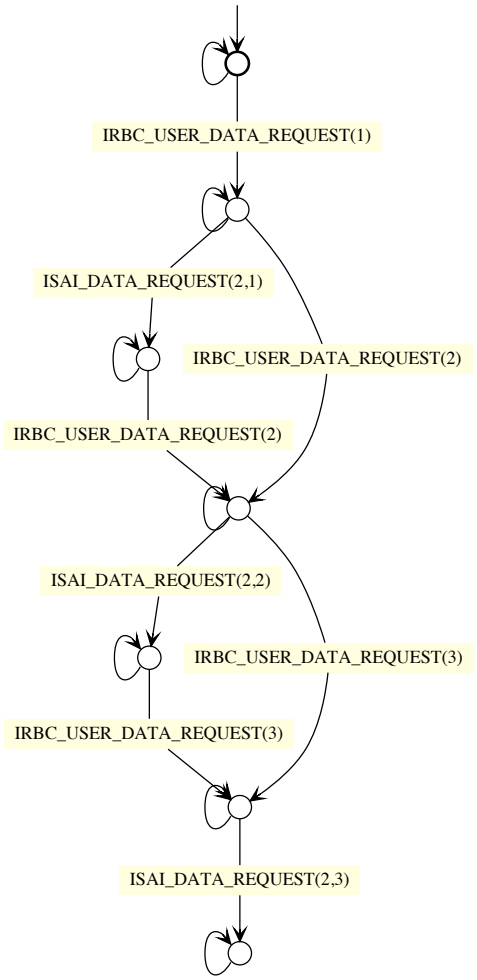


Figure 17: Flow of RBC Data requests messages through the ICSL

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:

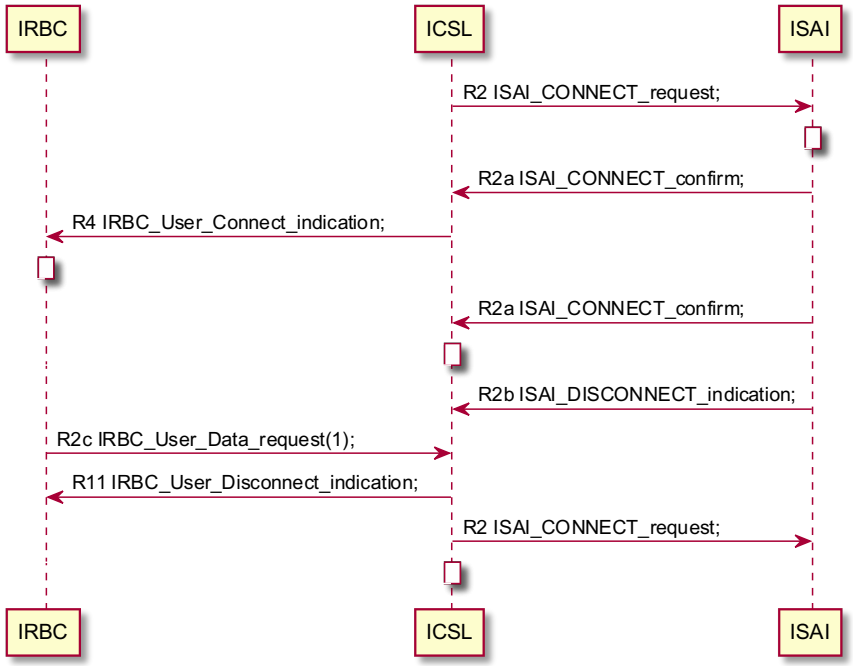
R8: *When in COMMS state is received a RBC_User_Data_request(userdata), I_CSL forwards a SAI_Data_request(userdata) with the same data to the SAI.*

But this is a possible valid behaviour because the current RBC view of the ICSL status might not precisely reflect the reality. 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[11] [12]:

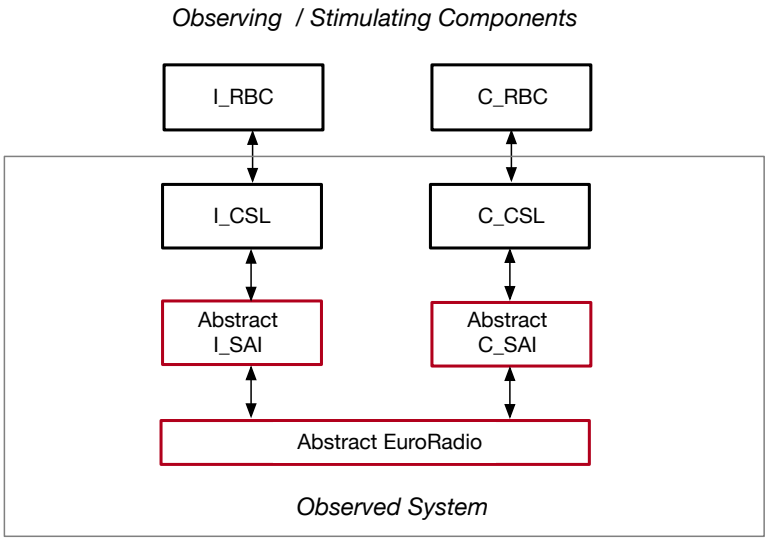


Notice that the RBC_User will surely receive the disconnect indication ... but a little later.
This may help in understanding whether a RBC_User message has been surely sent or not.

8.6 Analysing the external behaviour of the whole CSL layer

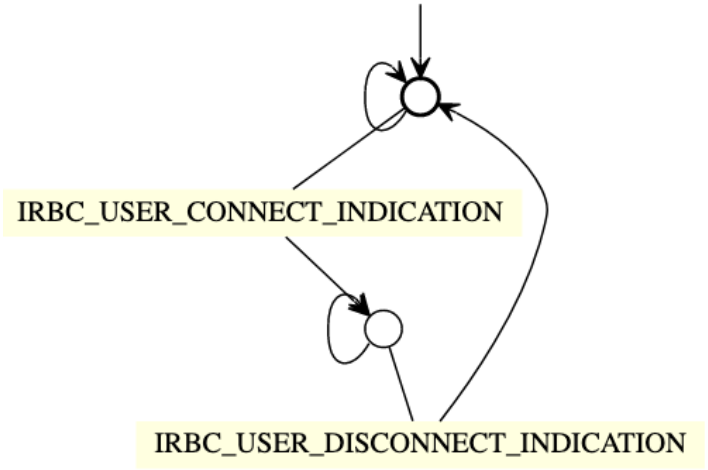
Three scenarios have been defined for analysing the behaviour of the whole CSL layer.

[11] We have removed from the trace the interactions with the Timer object.
[12] Notice that no assumption prevents our "chaos" SAI model to randomly send connection confirmations.



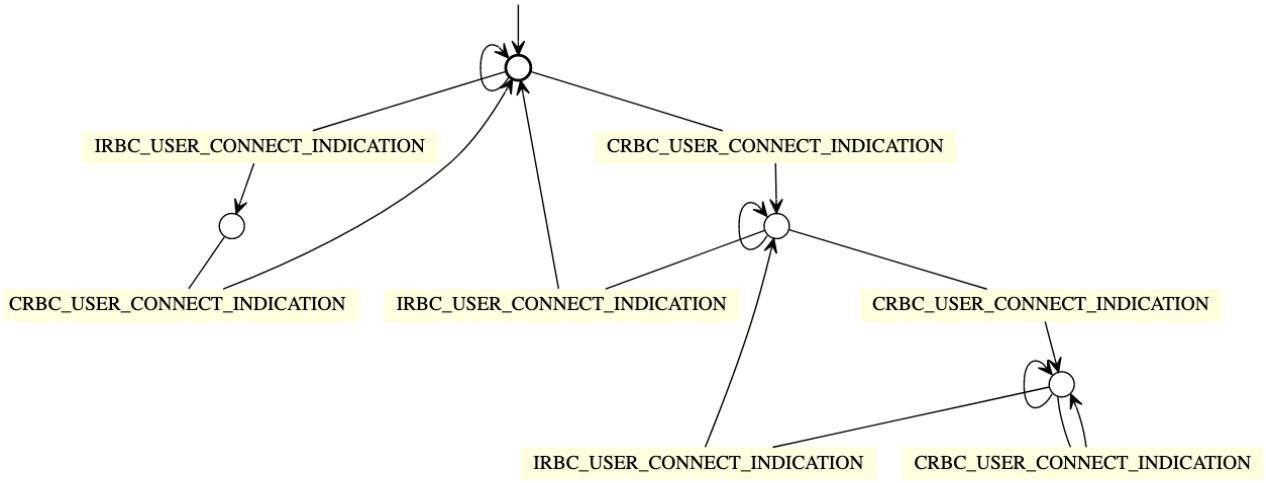
The scenario SC0 has already been mentioned in Section 5.13. In this case the two RBC components just act as observers, receiving the connection/disconnection indications, but without sending any message. This scenario can be used for analysing the relative views that the two sides of the RBC have about the current status of the communication line.

We can observe that the behaviour of the I_RBC - I_CSL interface (i.e. all the possible flows of sequences of messages) is the same already seen when observing the I_CSL component alone (interacting with a chaos-like I_SAI) as illustrated below.



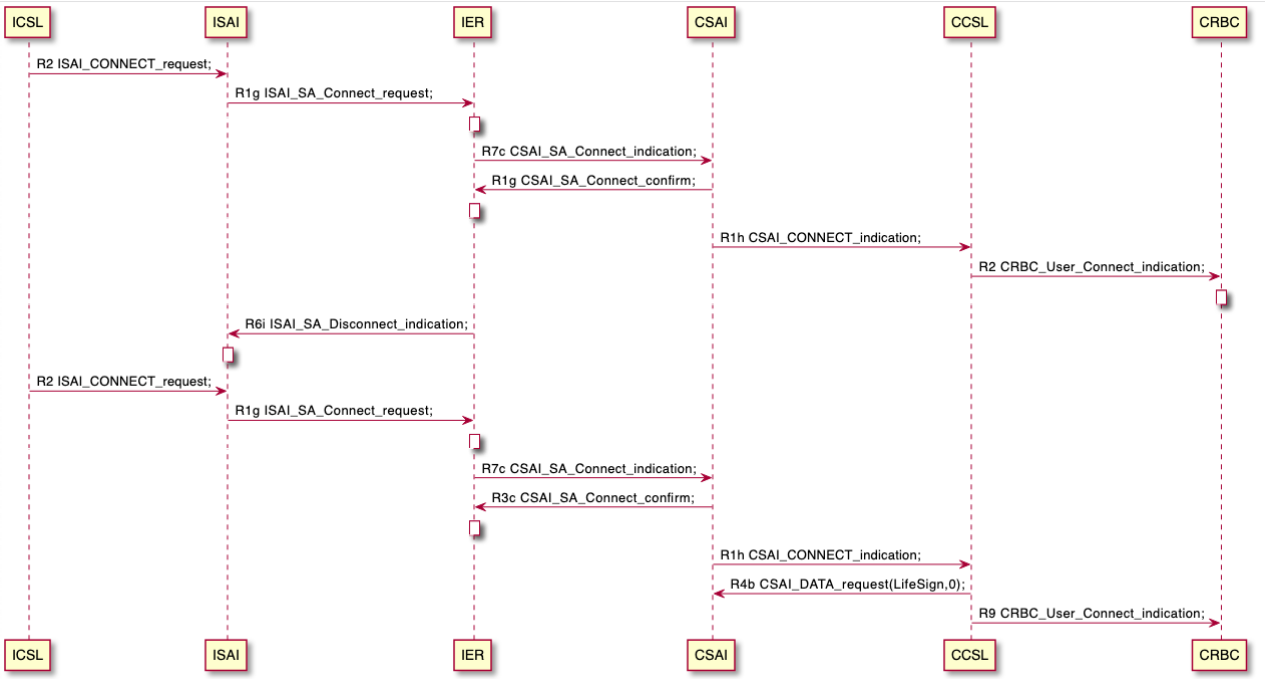
Now that we have a model of the whole system, we can also observe relations between events occurring at different sides of the RBCs, and check the expected properties (or properties to be guaranteed).
For example, we might intuitively expect that both RBC observers may loop between the connected/disconnected view, and if one side remains for a long time in the disconnected view, also the other will eventually move into the disconnected view.

We have been a little surprised to see the following system behaviour instead when observing just the traces of RBC_User_connect_indication messages outgoing from the whole CSL layer:



It seems, in fact, that there are execution paths in which the called RBC always receives connect_indications (i.e. having the view of an **always connected** system) while on the initiator RBC has the view of an **always disconnected** system.

Asking UMC to visualize a path in with two consecutive CRBC_User_Connect_Indication and without any IRBC_User_Connect_indication has revealed the following sequence of events:



I.e. The initiator side continuously tries to make new connections, which have a partial success in the sense that the called side accepts and confirms the request, but the last confirmation is always lost[13]. As a consequence, the initiator retries the connection, which is accepted and notified to the called RBC_User again, still without having ever notified to the initiator RBC any kind of connection indication.

In scenario SC1 the two RBC also try to send a limited number of messages.
If we suppose that the EuroRadio level never loses messages, introduces at most limited delays, we can see that no disconnections ever occur and all messages arrive at their destination and in the correct order[14].
< in progress>

In scenario SC2, one RBC sends one message and waits for a reply from the other side.
This third scenario is useful for having an estimation of the maximum delay, after which if the reply has not been received, we can trust it will no longer arrive (i.e. either the initial message or the reply has gone lost).
< to be done>

[13] In the current abstraction of the SAI levels we have not modelled the initialization phase of the safe connection. This aspect is not essential and the same situation is likely to occur even even if we a add an exchange of initialization messages. This will be checked when further refinements of the model will be made.
[14] notice that at this prototypal abstract level of modelling the EuroRadio level still does not introduce repetitions and reorderings, and the SAI component do not need to mitigate these problems