Deliverable D 2.2
Formal development Demonstrator prototype, 1st Release

Project acronym:	4SECURail
Starting date:	01/12/2019
Duration (in months):	24
Call (part) identifier:	H2020-S2R-OC-IP2-2019-01
Grant agreement no:	881775
Due date of deliverable:	Month 12 (30 November 2020)
Actual submission date:	30-11-2020
Responsible/Author:	CNR / Franco Mazzanti
Dissemination level:	PU
Status:	Issued

Reviewed: yes

Document history		
Revision	*Date*	*Description*
0.1	November 20 2020	First version under review
0.2	November 26 2020	Second version under review
1.0	November 30 2020	Final submitted Version
2.0	December 02 2020	Corrections of wrong cross-references
3.0	January 12 2020	Revision after comments received by the Project Officer

Report contributors		
Name	Beneficiary Short Name	Details of contribution
Franco Mazzanti, Davide Basile	CNR	Overall Structure and Content
Laura Masullo	SIRTI	Comments on Structure and Content
Alessandro Fantechi, Alessio Ferrari	CNR	Comments on Structure and Content
Carlo Vaghi, Elisa Sivori	FIT	Internal Review
Albert Ferrer, Jerónimo Padilla	ARDANUY	Internal Review

Table of Contents

1   Executive Summary

The overall goal of the Workstream 1 (WP2) "*Demonstrator Development for the use of Formal Methods in Railway Environment*", spreading on the activities of Tasks 2.1, 2.2, 2.3 and 2.4 of the 4SECURail project, is:

- the definition of a "Formal Methods Demonstrator process" (shortly *Demonstrator*) for the rigorous construction and analysis of system specifications (from the point of view of Infrastructure Managers),

- the application of the Demonstrator process to a railway signalling system case study, to perform a cost and benefits analysis and the evaluation of the required learning curve for the application of this Demonstrator process.

The previous deliverable D2.1 of Task 2.1: "*Specification of formal development Demonstrator*", has presented the overall structure of the Demonstrator process and has illustrated the selected choices for its architecture, both in terms of methodologies and tools.

This deliverable: "*Formal development Demonstrator prototype, 1st Release*", completes the description of the selected Demonstrator process (identifying the chosen MBSD framework for UML-based modelling and simulation) and describes how the defined Demonstrator process has been exercised in Task 2.1 with an initial fragment of the case study identified in Task 2.2. The initial fragment concerns the modelling and analysis of a Communication Supervision Layer (CSL), specifically dedicated to the control of the communication status between two neighbouring Radio Block Centre (RBC), over the Safe Application Intermediate sub-Layer (SAI) sublevel of the architecture. Modelling and analysis of the CSL have been conducted in three phases:

1) Fast prototyping of the CSL with the UMC **[UMC1][UMC2]** tool allowing easy modelling, animation, and verification of the system structured as set of interacting UML state machines. This step also includes an initial abstract model of the underlying connection layer allowing the two sides of the CSL to communicate.

2) Modelling of the prototyped CSL state machine with the Model-based Software Development **[MBSD]** framework Sparx Enterprise Architect **[SPARX]**. This advanced, industry-ready framework allows us to generate documentation with a graphical representation of the system components, and to perform interactive simulations of the system.

3) Translation of the prototyped CSL state machine definitions and underlying connection layer in the Event-B **[Bmethod]** notation as accepted by the ProB tool **[PROB]**. The resulting system, enriched with the components needed to stimulate the Communication Supervision Layer, is formally analysed using model checking techniques.

The generated models and related audio-visual material are available in **[ZenodoWP2]**.

Both the UML modelling and formal verification steps have allowed identifying ambiguities and imprecisions in the original natural language requirements, to gain more confidence of the overall correctness of the specification, and to acquire a deeper understanding of the CSL behaviour.

In our design of the Demonstrator process, we expressed as particularly desirable (but currently out of reach) the possibility to rely on a single MBSD framework for both design, code-generation, documentation, and formal verification. Our experimentation has shown that this desire is currently out of reach, especially if a semi-formal language such as UML is kept as reference underlying notation. This remains an important direction to be further explored.

An interesting side product of the future activity planned for Task 2.3 might be the generation of a revised version of the systems requirement, still in natural language, but more rigorous and more related to a formal model.

In the remaining phases of the 4SECURail project, this Demonstrator process will be applied in Task 2.3 on the complete case study defined in Task 2.2 and that activity will provide the refined reference for the costs and benefits analysis of Task 2.4.

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
fUML	Foundational Subset for Executable UML Models
IM	Infrastructure Manager
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 the Deliverable D2.2 "*Formal development Demonstrator prototype, 1st Release*" of Task 2.1 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 standardised interfaces). The introduction of formal methods in the process of specifying requirements carries the advantage to reduce ambiguities in the requirements definition; this could even introduce some benefits about the uniformity of products architectures and procurement and maintenance costs.

The idea of IMs is to have modular systems and to define standardised 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 **[MAAP2015] [MAAP2017][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 2.1 is to define and exercise a 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 complete the specification process described in D2.1 and to describe the instantiation of that process with respect to an initial fragment of the signalling system case study defined in D2.2 of Task 2.3.

This activity is aligned with the objectives of TD2.7 (Shift2Rail MAAP) *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 First exercise of the formal development Demonstrator process

This section is the core of this deliverable. We start, in Section 5.1, by recalling the reference framework as initially described in D2.1. The structure of the Demonstrator is fully determined, and the choices made for the various modules are detailed, also providing some background about techniques and methodologies used inside this Demonstrator.

The case study, which is fully reported in D2.3, is briefly described in Section 5.2 together with the initial fragment used by Task 2.1. The modelling phase, comprehending the various tools used inside the Demonstrator, is discussed in Section 5.3. The analysis and verification phase of the case study through the Demonstrator is performed in Section 5.4, whilst in Section 6 we summarise the experienced benefits from the application of the adopted approach.

5.1 The structure of the formal development Demonstrator process

This section briefly recalls some important concepts that underlie the definition of our formal development process, that can be found in full detail in Deliverable 2.1, and completes some aspects that had been introduced in D2.1, i.e., the rationale and choice of the inclusion of a commercial MBSE framework.

5.1.1 The reference framework

The point of view of Infrastructure Managers

We start by recalling the point of view of an IM that must provide a validated specification of the desired equipment not only to single suppliers but to multiple different suppliers that should produce equivalent products, capable of interacting with each other correctly. This is because the railway infrastructure is constituted by a multitude of subsystems (each one potentially developed by a different supplier) that must correctly interact among them, the so-called problem of *interoperability* of *systems of systems*.

A special case is when the produced specification takes the role of "standard specification" supported by international organisations (like the International Union of Railways (UIC) **[UIC]**, the European Union Agency for Railways (ERA) **[ERA]**, or UNISIG **[UNISIG]**, an industrial consortium to develop ERTMS/ETCS technical specifications), defined to create interoperable railways in all Europe (Single European Railway Area, SERA).

Therefore, to ensure interoperability, the IM must provide system requirements specifications that meet quality criteria, for example, requirements should not be ambiguous or in contradiction with each other, and they should be independent from specific implementation choices. Moreover, the developed system requirements specifications should guarantee the expected interoperability among subsystems. The goal of a formal specification is exactly to produce a mathematically precise representation for these system specifications allowing us to formally analyse such requirements.

The role of standard interfaces

Standard interfaces include two standardisation aspects, firstly the need to have interfaces between different equipment that have been agreed upon by different suppliers (i.e., standard). Standard interfaces are promoted and developed by Eulynx **[EULYNX]**, an initiative of a group of railway IM from different European countries. The Eulynx initiative aims to define a modular architecture for signalling systems, including standard interfaces for the individual components such that they can be supplied by different manufactures whilst maintaining the safety integrity levels required by such a critical railway system. This last point can take advantage of the introduction of FM into the process.

The second aspect is the need to use a standardised notation for the specification of standard interfaces. UML is the main standardised modelling language, consisting of an integrated set of graphical diagrams, developed to help system and software developers in specifying, visualizing, constructing, and documenting the artifacts of software systems **[WHATISUML]**.

UML behavioural models (e.g., state machines) can be graphically executed (simulated) to obtain initial feedback on the correctness of the design with respect to the intended requirements. Major drawbacks of this behavioural notation are uncertainties in the semantics, absence of a standard action language, and lots of implementation freedom (cf., e.g., **[FSKR29] [SG30]**).

Within the Demonstrator, UML plays three distinct roles:

- as graphical documentation of system requirements;
- as an engine for the simulation of system models;
- as a base towards translation into other formalisms supported by verification capabilities for overcoming the semantics limitations of UML.

The role of semi-formal methods and Model-Based Software Development

Model-based Software/System Development (MBSD) **[MBSD]** is a methodology for creating software and hardware artifacts that are designed starting from models typically expressed as graphical diagrams. Models support each phase of the development cycle. The methodology is also referred to as Model-based System Engineering, Model-based Development, Model-driven Architecture, and others. Although these terms may have slightly different meanings, hereby we consider all of them synonyms.

The development is guided by the definition of a model of software architecture. Such a model represents a semi-formalization at the abstract level of the system. It is indeed important to have a description of the key features of the system without dealing with the implementation aspects. Being able to manipulate an abstract model before moving to the implementation phase allows the early discovery of errors, by verifying the model with respect to the requirements, e.g., using model checking. Early bug detection will have a minimal impact on the development costs. Finding such bugs at a later stage of development, instead, would result in increasing costs for fixing them. Diagrams may help in identifying and reusing portions of systems, to speed-up the development

phase. Best practices and design patterns are used to guide the modelling phase.

One important aspect of using a semi-formal language is to avoid ambiguities that may lead to errors that would be hard to find in the source code. The models represent different views of the system at distinct levels, as for example, requirements definition, implementation, deployment. This supports a modular design methodology favouring independence between modules, so making them reusable in different contexts.

Once the compliance of the model with the requirements has been ascertained, it is possible to focus on the actual implementation. In the 4SECURail project we are focused on the role of the IM, whose goal is to formalise and validate requirements of standard interfaces. The development phase is delegated to suppliers, who will exploit the provided semi-formal definition of standard interfaces.

MBSD is also useful in case of evolution of the system. The changes will first be made on the model, with a chain of changes that are reflected in the subsequent phases of the development and validation process. This chain is well defined and structured, as an intrinsic consequence of the use of Model-Driven Engineering techniques. The synchronization among different artifacts produced in the various phases of development is called *Round-trip engineering*.

Diagrams are also important as a compound to natural language requirements, to improve the quality of the documentation and communications between the stakeholders involved in the software development.

Two important concepts in an MBSD framework are *separation of concerns* and *correctness-by-construction*. An MBSD framework promotes separation of concerns by distinguishing the separate phases of the development cycle, offering distinct levels of abstraction, and promotes correct-by-construction solutions, where correctness is intended with respect to a set of desired properties. These properties are generally derived from the natural language requirements document or the scenarios to be modelled. Correctness is ascertained on the model solving defined for the problem at hand and is reflected in the implementation that is derived from the model.

Generally, it is possible to use the designed models to automatize the source code implementation phase. Such implementation will then follow established design patterns that are enforced by the model from which they are derived. For example, a State Machine Design will enforce a State Pattern in the implementation. Several languages are supporting MBSD, and the Object Management Group (OMG) promotes UML as a modelling language for MBSD **[OMG-UML]**.

The Demonstrator not only defines an MBSD framework but applies it to the development of the specific case study. The adopted methodology has also been used in complementary projects as X2Rail2 and Eulynx and is one of the modules of the architecture of the Demonstrator. The Demonstrator defines an MBSD framework in which the first step addresses the critical phase of formalising natural language requirements employing (semi)-formal notation, before the software development, as suggest by methodologies widely spread in software engineering **[MBSD]** .

These methodologies advocate the usage of models (e.g., UML Diagrams) for modelling system requirements to produce a first prototype. UML is an ISO standardised notation **[OMG-UML]**, the standard for MBSD. We recall the twofold aspect of standard interfaces for interoperability between manufacturers and for the use of a standard notation. A standard notation helps in minimising ambiguities that could hamper interoperability, and it promotes reusability. Hence,

UML and MBSD are important for our approach that uses a standard notation for developing standard interfaces.

The Demonstrator can also support costs and benefits analysis of MBSD. Indeed, MBSD is an industrially adopted methodology that is not only used for requirements and specifications phases but covers all the phases of system development.

Hence, the impact of the modelling phase is not only restricted to requirements but can spread down to the artifacts and thus helping to validate the Supplier's artifacts.

Diagrams and models are proposed for each phase, following a V software development approach as suggested by CENELEC norms **[CENELEC EN50159]**. Hence, the costs and benefits analysis can also consider such development methodology, a part of which will be exercised by our Demonstrator.

In the Demonstrator architecture, it has been shown that the IM need to formalise the natural language requirements of the standard interfaces between different entities forming an infrastructure. This is important to guarantee interoperability between different suppliers. Indeed, all suppliers must conform to the same standard interface. Slight deviations from the requirements, due to different interpretations, could lead to the impossibility of making different subsystems cooperating.

By formalising standard interfaces through a standard notation, the risk of incompatibility between different subsystems is reduced since the interface's requirements are clear and unambiguous. Models are also important for reasoning about the correctness of a specification. This can be performed by directly applying within the MBSD framework the supported formal verification techniques (if any), or by translating the semi-formal UML model into a formal specification amenable to automatic verification (e.g., model checking, theorem proving). The UML models can also be used to generate test cases through Model-based Testing. Test cases can be handed to suppliers to be used for providing evidence that their subsystems are conformant to the specification provided by the IM. Automatic code generation starting from such models will also ensure that the implementation is derived directly from the specification, employing traceability. Hence, several facilities are available to ensure that suppliers comply with the specifications provided by the IM.

The role of formal methods

The translation of semi-formal UML diagrams into another formalism equipped with semantics that is described using rigorous mathematical notation allows mathematical proofs of correctness of the specification according to the quality criteria that must be met by the specification.

Such mathematical demonstrations will provide further evidence of the correctness of the specification against the desired quality criteria and will improve the confidence in the actual interoperability of the designed systems.

These proofs can be obtained by using, for example, theorem proving or model checking. Indeed, these can be considered the two approaches to system verification that are mostly used, also in railway-related contexts **[Bmethod][ICSE2020]** .

Concerning model simulation and validation functionalities provided by semi-formal design

methods, the great advantage of FM is that they allow to verify properties on all possible system behaviours, whilst simulation is used to observe only some behaviours, those manually stimulated by the user. To do that, formal verification may require the specification to also include a model of the environment/user interacting with the (specified) system.

The X2Rail2 complementarity

Both 4SECURail project and complementary project X2Rail2 **[X2RAIL2]** identify the responsibility of IM in providing standard/rigorous/verifiable specifications of the standard interfaces.

As described in D2.1, the 4SECURail approach is based on the use case described in Section 5.4.1 ("Development of Systems with Standardised interfaces") of D5.1 of X2Rail2 **[X2R2D51]**, which describes the use of formal and semi-formal methods for the specification and development of standardised systems.
For that use case, the 4SECURail activity is focused on the part related to the IM, i.e., to the phase of construction of high-quality system requirements specifications. Moreover, while X2Rail2 focuses its efforts on the use of formal and semi-formal methods for the analysis of a single system component, in 4SECURail we want to tackle the important but difficult aspect of analysing also interoperability issues in the case of Systems of Systems.

In agreement with what has been done in X2Rail2 about the use case of FM, our Demonstrator will also follow the choices made by the Eulynx project **[EULYNX]** regarding the underlying notation for the system specification, i.e., the reference to the UML/SysML **[OMG-UML]** **[OMG-SysML]** standards.

Both X2Rail2 and ASTRail **[ASTRAIL]** projects provide a taxonomy of recommended tools for formal verification and model-based development. The choices adopted by our Demonstrator, which are ProB **[PROB]** as tool for model checking and Sparx Enterprise Architect **[SPARX]** as a tool for simulation and model-based development, are among the choices experienced and suggested by these projects.

5.1.2 The choice of the MBSD Sparx Enterprise Architect platform

This is one aspect that in the previous Deliverable D2.1 ("Specification of the Formal Methods Demonstrator") has been left open because more experimentation was deemed necessary before reaching a conclusion.

There is a wide range of tools that support MBSD. Some of them are open-source and thus more fashionable for research purposes. Others are commercially available, offering the desired features for industrial adoption, as e.g., customer support and licensing systems. Two top tools for MBSD are Cameo System Modeler by NoMagic **[3DS]** and Enterprise Architect by Sparx **[SPARX]**. Another tool is PTC by WindChill **[PTC-Windchill]** (adopted by Eulynx) and many others are

available.

An initial task was devoted to test and to select one of the different tools as the most suitable for the Demonstrator and to collect data as, e.g., costs of licenses, customer support, training. We focused on those containing features desirable by industrial suppliers. We discuss three of them now.

- **NoMagic Cameo Systems Modeler**: Cameo Systems Modeler **[3DS]** is a UML/SysML modelling tool that supports MBSD with Round-trip Engineering, and it has been adopted among the others by Bombardier Transportation, Siemens Mobility, Alstom, Deutsche Bahn, CAF Signalling. One of the peculiarities is that it enforces UML2 rules for syntax (notation) and semantics. It also provides solid support for requirements traceability; automated documentation generation and it offers plugins to integrate with external Requirements Management and Simulation tools.

- **Sparx Enterprise Architect:** Sparx Enterprise Architect (Sparx EA) **[SPARX]** is a competitor of Cameo Systems Modeler; it is an MBSD tool supporting Round-Trip Engineering, and it has been used, among the others, by the Belgian National Railway Company SNCB **[SPARXCaseStudy]**. It supports requirements traceability, automated documentation generation and simulation. To its competitors, Sparx EA has a more competitive price. Sparx EA is designated as a "Best Value" among competitive UML2 modelling tools according to **[UMLReview]**, and is among the top tools according to **[UMLSurv]**.

- **PTC WindChill Modeler:** we finally mention PTC WindChill Modeler **[PTC-Windchill]** as another tool supporting Model-based Development and Round-trip Engineering, adopted, among the others, by Alstom SA **[PTC-Alstom]** . It provides functionalities for document generation, requirements traceability, and simulation.

Why Sparx Enterprise Architect?

The industrial tools promoting MBSD provide lots of functionalities that span across all the development phases of a system or software, supporting solutions concerning, e.g., database management, collaborative working, process analysis, version management and many others.

However, we are focused on the point of view of the IM, whose goal is not to fully develop a system, but to describe the requirements using (semi-)formal notation. In other words, we are interested in facilities for modelling state machine diagrams, and simulating their behaviour, to help interaction with stakeholders and validate the requirements. A strict unambiguous subset of state machine diagram notation provided by UML can be equipped with a specific algebraic meaning and thus state-of-the-art formal verification techniques can be applied to it.

We now list some of the desirable properties that lead to the adoption of Sparx EA in the Demonstrator.

Firstly, a desirable MBSD tool for the Demonstrator must be affordable, given the resources of the project and the fact that such tools tend to be expensive depending on the specific configuration. As already mentioned, the cost of the license for Sparx EA, if compared to the cost for the same

service offered by competitors (e.g., PTC), is lower by an order of magnitude (as per November 2020). Thus, Sparx EA provides the best compromise between cost and quality and has been recognised as one of the top tools for MBSD in recent surveys **[UMLReview][UMLSurv]**.

Apart from the clear cost/benefit advantage, a desirable MBSD tool must also provide an appealing, clear graphical display of UML State Machine diagrams, and must provide facilities for simulating such diagrams and documenting them. This is because in the Demonstrator the infrastructure manager will specify the standard interfaces, document them, and validate them using simulation.

Moreover, standard interfaces specifically require that different models of the various systems must be composable and the interactive simulation must span across the various simulatable components, thus demonstrating their interoperability. Below we will discuss how Sparx EA provides a light interactive simulation of different interoperable machines, if compared to, for example, Cameo modelling tools.

We briefly summarise our experience in the usage of Cameo modelling tools.

As already mentioned, Cameo adheres to SysML/UML notation. The crucial task of modelling the interaction among different devices, potentially provided by different suppliers, required usage of additional diagrams and notation (i.e., activity diagrams containing opaque behaviours, block diagrams and internal block diagrams, ports connected to signals in turns connected to triggers). This is needed to express the primitive operation of sending an event to another device, to model interactions between components and thus validate their interoperability.

The adoption in Cameo Modelling Tools of the Action Language for Foundational UML (ALF) as the scripting language for expressing effects of events occurrence made such specification harder to adapt to the artifact models coming from the other tools included in our Demonstrator (i.e., UMC, ProB). All these extra layers of diagrams would hamper the initial goal of having visual diagrams easy to communicate between various stakeholders, and whose translation into a formalism amenable to formal verification could be straightforward and easily validated.

On the other hand, Sparx EA provides a specific artifact, called *Executable State Machine,* which is specifically used for simulating the composition of different state machines. Such state machines can interact through a simple instruction for sending an event. As such, Executable State Machines do not require overly superfluous notation and provide all the ingredients for an easy translation to/from a formal specification amenable to verification, and for graphically displaying, in a succinct way, the informal specification, as well as simulating it. Apart from simulating a composition of state machines, it is possible to interact with each state machine using the standard simulation engine of Sparx EA.

5.1.3 The structure of the Demonstrator

The Demonstrator structure and its rationale have been defined in D2.1. In this section, we briefly recall the planned structure and its underlying rationale.

5.1.3.1   The desired and the possible

The desirable structure of the Demonstrator would be an industry-ready tool, offering all facilities desired by industries such as direct support, licensing management, team development, available training and so on (we refer to ASTRail deliverables for a full account) **[ASTRAIL-D41][ASTRAIL-D43]**.

Such tool should allow on one hand to design a model of the specification using standard notation such as UML, and on the other hand to verify, using FM, the designed model. This should come without any translation effort. The final implementation should be automatically derived from the validated model.

However, it is not possible to fully achieve the desired structure. The main factor hindering the desired solution is that UML, although being the standard for MBSD, in its latest incarnation still has an informal and incomplete semantics of its behavioural models. The semantics is incomplete because specific semantics choices are left to the producers of tools supporting UML. The reason for that is to make the standard more adaptable to the various tools.

It is informal because the latest UML standard document still describes semantics in natural language terms and not in rigorous mathematical notation**[OMG-fUML1][OMG-PSSM]**.

The main practical drawback is that the very same UML behavioural model could behave differently in different tools. The other important drawback is that to be formally verifiable a model must be designed using a formalism equipped with a formal semantics described with rigorous and validated mathematical foundations.

On a positive note, the UML notation can be considered a valuable tool for animating graphical diagrams, to give intuitions of how a specification behaves to non-experts. UML tools can be used to provide compound documentation to be attached to natural language requirements such as a standard interface documents. Once a specific UML tool is chosen, the undefined semantics choices can be determined, and the resulting diagram can be translated into a formalism amenable to the application of FM. Finally, UML and MBSD represent the bridge between formal verification of specifications and their software development, by providing facilities for automatic generation of code from a given specification.

The structure of the Demonstrator thus consists of different tools for different purposes. UMC **[UMC1][UMC2]** is the chosen lightweight tool to provide fast prototyping of UML State Machine Diagrams, with a formal semantics overapproximating and subsuming possible concrete semantics of different industry-ready UML tools. This last aspect is crucial in ensuring interoperability between models potentially having slight semantics differences. UMC is not an industrial tool but an on-going research project that has been developed and maintained inside the Formal Methods and Tools lab at ISTI CNR Pisa in the last decades.

The UML State Machine models made in UMC are mechanically translated to Sparx Enterprise Architect UML models, to perform simulation and documentation using a state-of-the-art UML industry ready tool, as recommended by X2Rail2 and Eulynx.

On the other hand, UMC models are also translated into ProB, which is a model checking tool applied in several railway signalling projects.

We underline that the Demonstrator helps bridging formal specification and verification (through

ProB and UMC) with software implementation (using Sparx Enterprise Architect). Thus, the IM leaves to the suppliers no unspecified aspects about how to handle the standard interfaces to deliver the final product. Figure 1 below summarises the Demonstrator process.
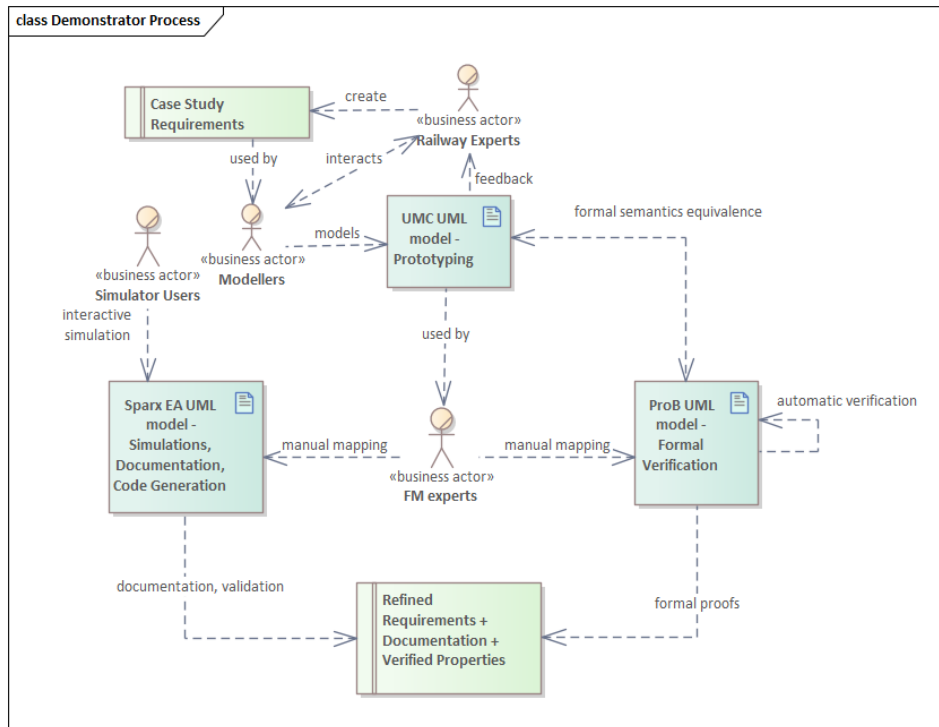
Figure 1 The Demonstrator process

5.1.3.2   UMC for prototyping

Given the list of natural language requirements describing the selected case study, our first step is to transform these requirements into a set of interacting UML state machines providing a semi-formal model of the system.

This step has been done using the UMC framework, developed in the last 20 years internally at CNR and used in many other projects. The reason for this choice is that UMC is a tool explicitly oriented to the *fast prototyping* of systems constituted by interacting state machines. UMC allows the user to:
- specify a UML state machine design using a simple textual notation,
- visualise the corresponding graphical representation,
- interactively animate the system evolutions,
- formally verify (using on-the-fly model checking) properties of the system behaviour. Detailed explanations are given when a property is found not to hold, also in terms of simple UML sequence diagrams.

Notice that UMC is not an industry-ready tool: e.g., its available documentation is limited, and no commercial support is provided. The role of this CNR framework is the support of experimentations and teaching of advanced formal verification techniques.

For this reason, our Demonstrator will make use of UMC only in the initial prototyping phase, while relying on other industry-ready, commercially supported frameworks for the semi-formal and formal modelling and verification.

The double textual/graphical representation of state machines fits well the need of simplifying the task of transforming the UML model into the textual Event-B notation accepted by the ProB tool, as well the need of having an immediate graphical feedback of the state machine design for its subsequent inclusion in Sparx EA.

Command-line executable binaries for UMC are freely available upon request, while a graphical interface to the framework is freely accessible on the CNR-ISTI-FMT site **[UMC1][UMC2]**.

5.1.3.3   Sparx Enterprise Architect for Model-based Development

Being that UMC is not industry ready, and only destined to prototyping phase, the official role of UML tool in the Demonstrator has been assigned to Sparx Enterprise Architect.

As already described in Section 5.1.2 with this platform it is possible (among other functionalities):

- to design graphically appealing UML state machines,
- to animate their behaviour (according to the Sparx EA execution model),
- to automatically generate the corresponding documentation (an example of which is shown in Appendix E - Sparx Model Report – CSL),
- to handle traceability, that is, the mapping of the model with respect to the initial set of requirements,
- to generate specific software components of the system from its models.

The Sparx EA execution engine (used for simulations) is deterministic and gives its own specific solutions to all the aspects intentionally unspecified in the UML standard. From this point of view the UML semantics adopted by the UMC and ProB formal framework is more general than the one observable through a specific Sparx EA simulation, in the sense that it is not locked to a specific vendor solution. This aspect will be discussed in more detail in Section 5.4.1. Sparx EA is a commercial product, and the license is permanent with renewable customer support each year. Professional training is available on-demand.

5.1.3.4   ProB for detailed simulations, analysis, verifications

ProB **[PROB]** is an animator and model checker for the B-Method. It allows animation of Event-B specifications and can be used to systematically check a specification for a range of errors.

Some of the reasons for the successful experience of its use in the ASTRail project, which have suggested to reuse it on 4SECURail as well, are the following:

- it is a free, open-source product whose code is distributed under the EPL v1.0 license

[PROB-licensing][EPL];
- it is actively maintained, and commercial support is available from Formal Mind **[FORMALMIND]** ;
- it runs on Linux, Windows, and MacOS environments;
- it has several nice, very usable graphical interfaces, but it can also be used from the command line;
- it is well integrated with the B/Event-B ecosystem (Rodin, AtelierB, iUML, B-Toolkit) **[Bmethod]**;
- it allows construction, animation and visualisation of nondeterministic systems;
- it allows formal verification through different techniques like constraint solving, trace refinement checking, and model checking.

Concerning the planned uses of ProB inside 4SECURail project, this framework, however, suffers a few limitations since it is born and grown around a specific specification notation (Event-B). In particular:
- It does not allow the explicit modelling of multiple mutually interacting state machines. The only way to achieve that is to merge all the separate machines into a global one.
- Event-B state machines are different from UML/SysML state machines. At the current state of the art several proposals for translations from UML to ProB state machines have been made but, to the best of our knowledge, no industrially usable product currently supports that mapping.
- Model checking is not likely to scale when the system is composed of many interacting asynchronous state machines.

The ProB encoding of the UML design can be obtained quite directly from textual UMC representation of the system, and the coherence of the two specification notations can be formally verified.

ProB natively provides feedback in the presence of error or as an explanation of properties using execution traces. To provide a graphical representation of such feedback, a simple tool for depicting these execution traces in terms of Sequence Diagrams is provided.
ProB is a freely available tool distributed under EPL v1.0 license. As stated above, commercial support is available from FormalMind.

5.2 The selected case study

The case study chosen to test the 4SECURail Demonstrator is a subset of the RBC/RBC hand over protocol, as specified by Deliverable 2.3 of 4SECURail project.
In the ERTMS/ETCS train control system, a Radio Block Centre (RBC) is responsible for managing trains under its area of supervision. A handover procedure is needed to manage the interchange of train control supervision between two neighbour RBCs: when a train is approaching the end of the area supervised by one handing over RBC, an exchange of information with the accepting RBC

takes place to manage the transaction of responsibilities. Since the two neighbouring RBCs may have been manufactured by different providers, the RBC/RBC interface is a typical product where the products (RBC) of different suppliers must be interoperable.

Deliverable 2.3 integrates the ETCS specifications contained in SUBSET-039 – FIS for the RBC/RBC Handover **[SUB-039]** and SUBSET-098 – RBC/RBC Safe Communication Interface **[SUB-098]** with additional requirements. The adapted definition of the subsystem is limited to higher application levels and safety levels (SAI sub-level of SUBSET-098). Thus, the case study isolates and identifies two layers:

- Communication Supervision Layer (CSL): responsible for commanding opening/closing of a communication line between RBC and keep the connection alive through Life Signs. Its functional requirements are covered by UNISIG SUBSET-039 and UNISIG SUBSET-098.
- Safety Application Intermediate sub-layer (SAI): below the CSL, it implements the protection mechanisms against deletion, replication, resequencing, and delay threats as identified by CENELEC EN50159 **[CENELEC EN50159]**. In this case, its functional requirements are covered by UNISIG SUBSET-098. Most requirements related to the safety of the communication are allocated in the SAI.

Above the CSL, the RBC User layer includes all application functions (e.g., evaluation of Movement Authorities MAs, communication with on-board units, actual management of RBC-RBC handover transactions) and the generation/reception of information to communicate, while protocol Layers are dedicated to formatting and exchanging such information with communication partners. The specification of RBC_User functions is not included in the requirements of the case study. Moreover, lower levels below SAI (EuroRadio and Communication Functional Module of SUBSET-098) are also not part of the requirements of the case study.

Thus, the Demonstrator will be applied to the CSL and SAI levels, whilst RBC_User, EuroRadio and Communication Functional Module are treated as external environment.

Figure 2 shows the overall structure of the system. Notice that of the two communicating sides, one side is configured as "initiator" of safe connections while the other is configured as "called side".

CSL/SAI Initiator side   CSL/SAI Called side

RBC_User_1   RBC_User_2

I_CSL   C_CSL

I_SAI   C_SAI

Euradio/CFM levels
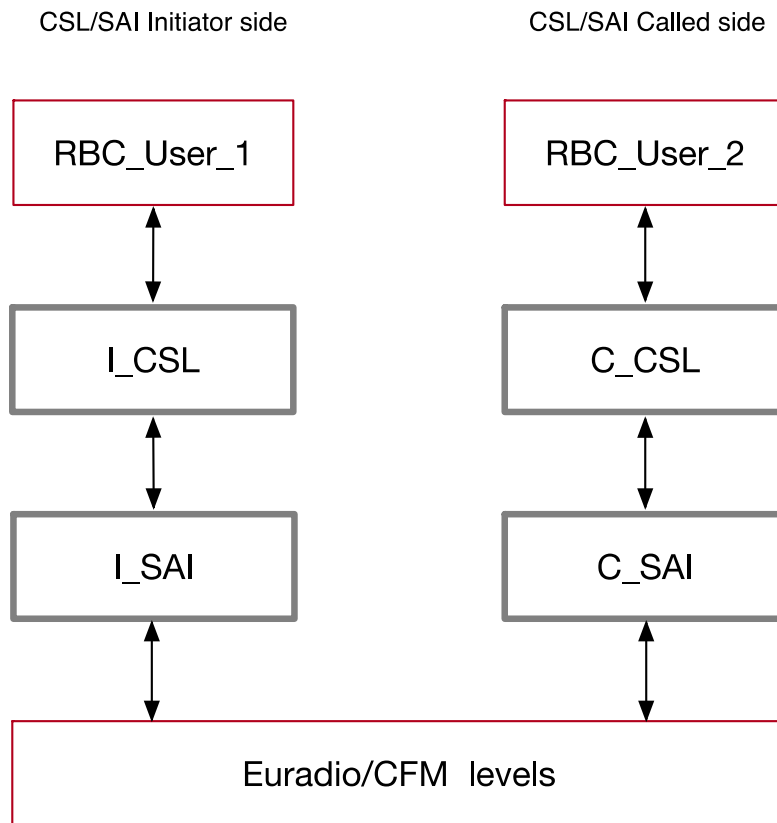
Figure 2: Overview of the case study structure

5.2.1 Task 2.1 initial case study fragment

Within this first experimentation of the Demonstrator prototype in the context of Task 2.1, we are interested in analysing the behaviour resulting from the interactions of the two CSL sides (initiator and called). Therefore, we designed an initial abstract version comprehending both the underlying SAI and EuroRadio layers allowing the two CSL to interact. Similarly, we added to this composite system two abstract RBC_User sides that stimulate and receive data from the underlying CSL subsystems with the exchange of RBC_User data.

As said in D2.3, it is outside the goal of the project to fully model the RBC_User behaviour to give a complete design and analysis of the RBC-RBC Handover protocol. Thus, such RBC_User will produce inputs and outputs following pre-determined operational scenarios. The actual modelling and formal analysis of underlying SAI levels will be an objective of Task 2.3.

The resulting formal model that we consider in this deliverable has therefore the structure:
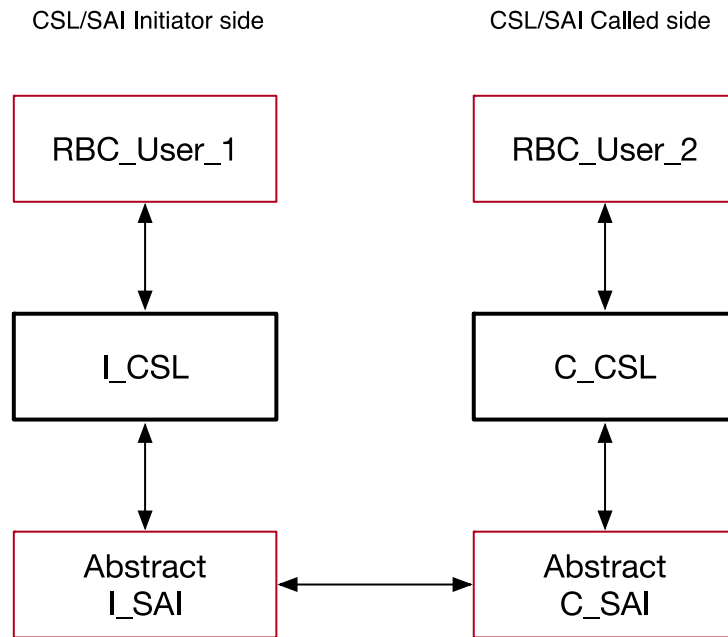
Figure 3 : Case Study Fragment

At the current stage of the project a formal model of the SAI-EuroRadio level has not yet been done (this will be a goal of Task 2.3). Thus, the used SAI abstraction that is part of the requirements of the case study must be a prototypical one. This prototype model will be replaced by the actual SAI model in the final release of the Demonstrator.

The current prototypal SAI model allows us to perform verifications of the system that provide useful early feedback on the CSL design.

For simulation in Sparx EA the SAI component, as well as the RBC_User, are not specified. Indeed, inputs coming from SAI and RBC_User are provided by the human user acting as the environment stimulating the CSL component. On the converse, for formal verification with model checking the human is out of the loop, thus the need to provide abstract models of RBC_User and SAI.

For the higher RBC_User levels we have considered the modelling of two kinds of scenarios:

1. A basic scenario in which the two RBC_Users do not play any active role: they just receive notifications of the creation and destruction of communication sessions between them. In this case, the entire system activity is centred on the CSL connections, disconnections, and management of the lifesigns. We briefly recall the behaviour of the two CSL modules:
   - If disconnected, the initiator CSL (I_CSL) tries (each period identified by a connect timer) to establish a new connection.
   - If disconnected, the called CSL (C_CSL) waits for connection indication from the initiator side.

- If connected, when a local send_timeout expires each CSL sends a lifesign message to the other CSL.
- If connected, when a local receive timeout expires the CSL (initiator or called) closes the connection, notifying the RBC_USER and the other CSL.
- If connected, when a CSL receives a disconnection indication from the lower SAI level it becomes disconnected notifying the RBC_User of the event.
- The lower levels (SAI/EuroRadio) in their current abstract modelling behave as if each message had its own sequence number, and they autonomously close the connection when the difference between the sequence number of the two most recently received messages is at least N (N is a parameter). Moreover, in this abstract SAI model, received messages can be nondeterministically marked as "invalid", while in the real system this will occur only when the "travelling time" (arrival time - departure time) exceeds a certain threshold.

2. A second scenario is based on the previous one, with the addition that the RBC_User on the initiator side tries to send a message to the other RBC_User, and waits for another message as reply.

3. A third scenario, similar to the previous one, but with inverted roles (the RBC_User on the caller side is the one that starts sending a message and waiting for a reply).

All these scenarios can furthermore be configured with different values for the timers used for requesting a connection (max_connect_timer), sending a lifesign (max_send_timer), closing a connection (max_receive_timer), and the value of N for the SAI abstractions.

5.3 An overview of the modelling phase of the case study

In this (sub-)section, we provide some details on the actual modelling of the case study in this Demonstrator. We refer to the corresponding Appendix for a complete presentation and the Zenodo-repository **[ZenodoWP2]** for the observation of the full models and audio-visual material on our modelling and analysis steps.

5.3.1 UMC modelling

The main characteristic of UMC is that a simple textual notation is used to specify the state transitions of a UML state machine. In Figure 4 one of these transitions is depicted.

```
-- when connecting handle connection confirmation
--
R8_ICSL_IRBC_rbcuserconnectindication:
  NOCOMMS -> COMMS
      { SAI_CONNECT_confirm   /
          RBC_User.RBC_User_Connect_indication;
          connect_timer := max_connect_timer;
          receive_timer := 0;
          send_timer := 0; }
```

Figure 4 : Example of UMC rule

Each transition definition is defined by:
- an optional transition name (R8_ICS_IRBC_rbcuserconnectindication in Figure 4),
- the source and target states of the transition (NOCOMMS, COMMS in Figure 4),
- A block { } containing:
    - the *triggering* event of the transition (SAI_CONNECT_confirm in Figure 4), possibly with parameters and guards,
    - the sequence of actions to be performed as an *effect* of the transition (the sending of the RBC_User_Connect_indication signal to the RBC_User component and the assignment of the connect, receive, and send variables in Figure 4).

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

Figure 5 shows the automatically generated (thanks to Graphviz **[GRA]**) graphical representation of a state machine (I_CSL). Its layout is not interactively editable but can be saved in the .dot format for offline manual adjustments.

With UMC it is possible to check if/how a given transition is eventually fired, if/when a certain signal is sent, if/when a certain variable is modified, or a certain state reached.
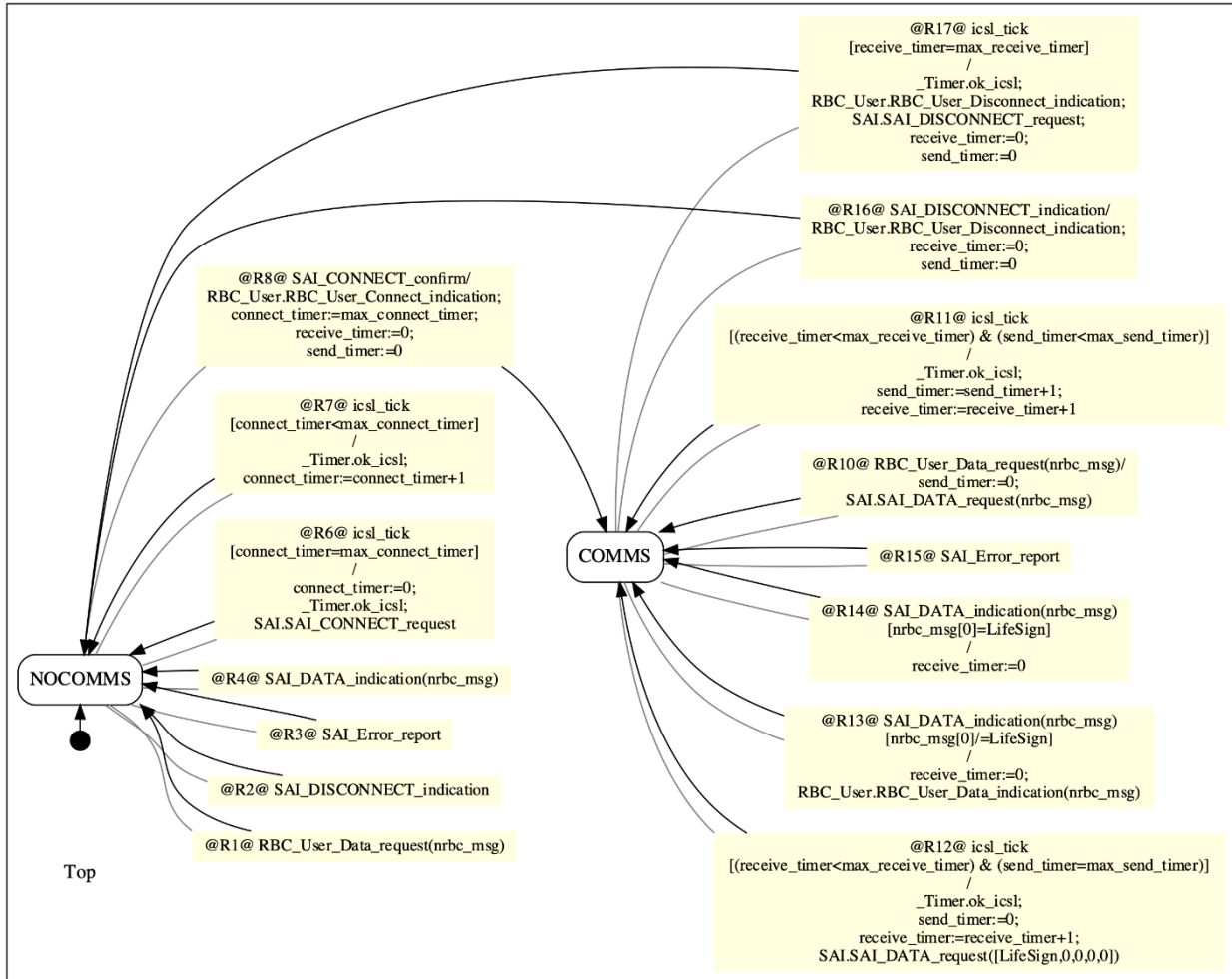
Figure 5 : The initiator CSL state machine

For example, we can ask for an explanation about when the transition shown in Figure 4 is fired (i.e., when it happens that the signal SAI_CONNECT_confirm causes the sending of the RBC_User_connect_indication), and the answer can be observed in term of a sequence diagram, as shown in Figure 6 (the graphical layout is automatically generated by UMC using the PlantUML online services **[PlantUML]**).
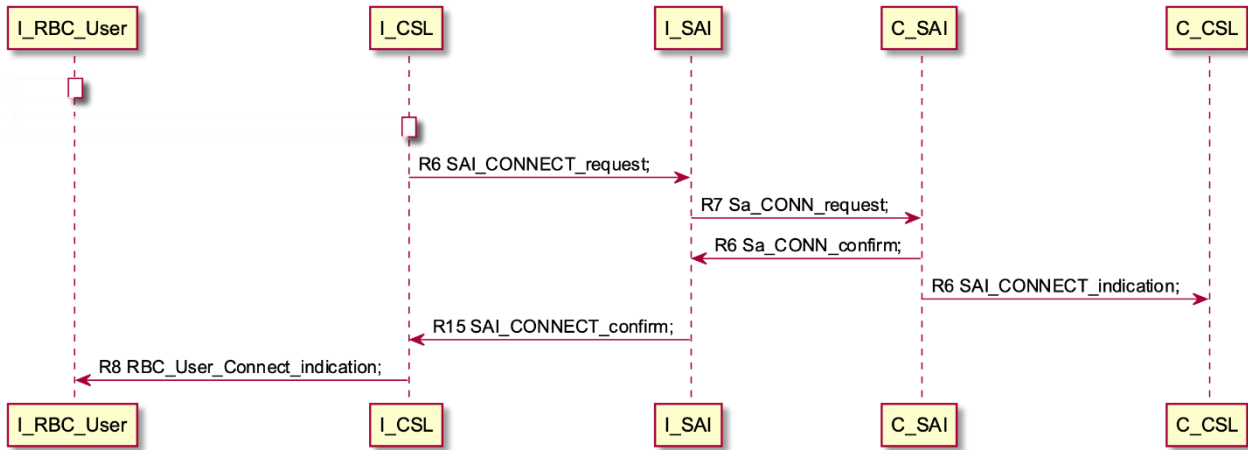
Figure 6 : A sequence diagram generated with UMC

The complete set of UMC models and some related audio-visual material can be retrieved from the public Zenodo repository **[ZenodoWP2]**. The UMC encoding of the I_CSL component is shown in Appendix C - UMC encoding of I_CSL.

5.3.2 Sparx Enterprise Architect modelling

In the Sparx Enterprise Architect, state machines are specified using a WYSIWYG (What You See Is What You Get) graphic user interface. In this way the user has intuitive control of the graphical display of diagrams. Figure 7 depicts the state machine for the CSL derived from the UMC models. Details about the mapping from UMC to Sparx EA are in Appendix A – More details on Sparx Modelling.

Figure 7 : The Communication Supervision Layer State Machine

In a UML framework as Sparx Enterprise Architect, one cannot define state machines only. Indeed, a state machine has to be assigned to a class in a class diagram, so to model the behaviour of that class. Moreover, to compose state machines and make them interact, an executable state machine connecting various class instantiations must also be defined.

Details about the class diagram (in Figure 8) and executable state machine (in Figure 9) are in Appendix A – More details on Sparx Modelling.
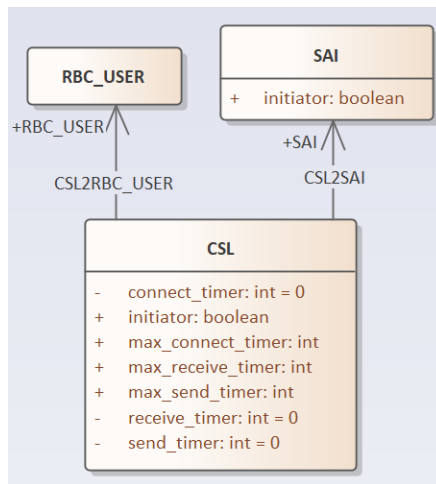
Figure 8 : The class diagram
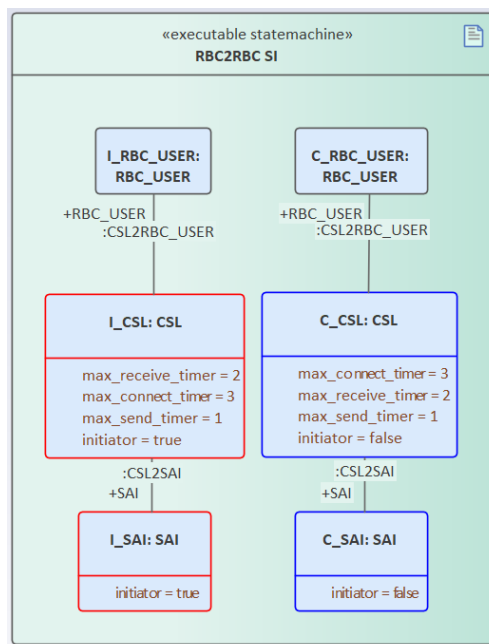
Figure 9 : The executable state machine

Once designed, a system composed of several interacting state machines can be simulated interactively, by sending triggers, to observe its behavior. The executable state machine is used

for generating code, and the simulation gives an interactive graphical animation of the system being debugged, as depicted in Figure 10 (a snapshot taken from a 13-inch screen).

Details about the behavior of executable state machines, comparisons with UMC, and description of the simulation functionalities are in Appendix A – More details on Sparx Modelling.
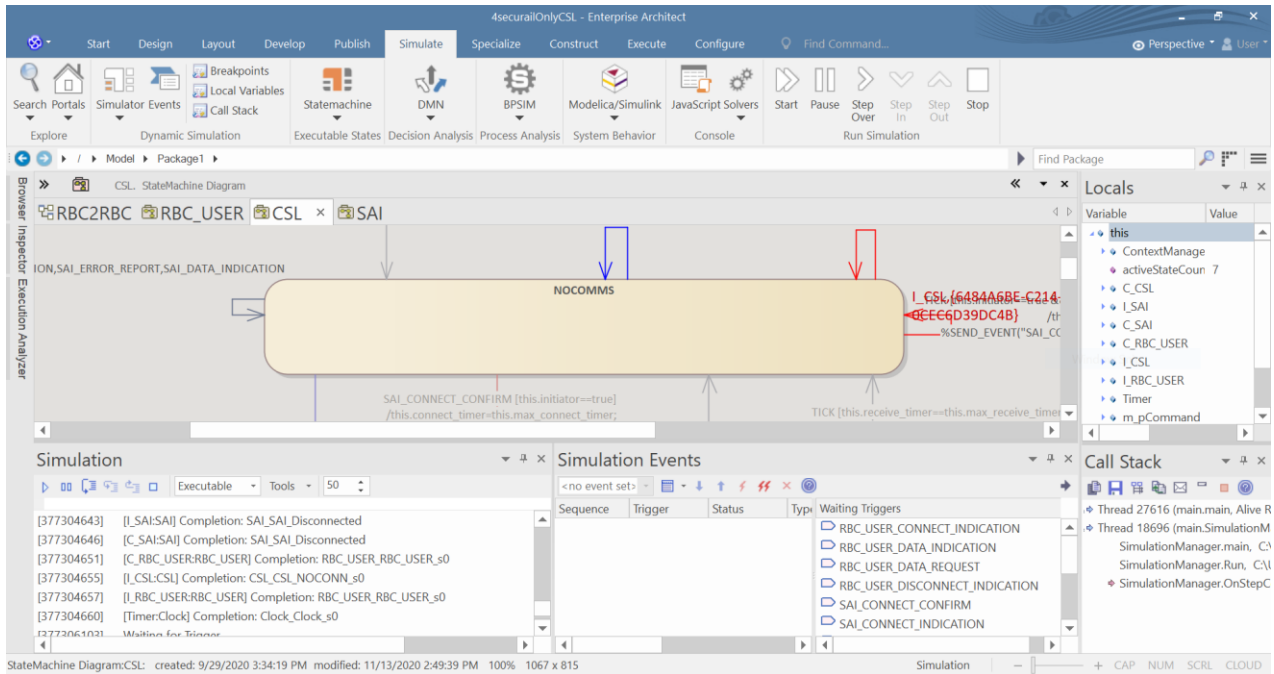
Figure 10 : Simulation workspace in Sparx EA

Another important feature for the industrial use of the tool is the automatic generation of the system documentation.

In **[ZenodoWP2]** the documentation automatically generated (as .docx document) from the developed model is present. The portion of documentation about the CSL is in the Appendix E - Sparx Model Report – CSL. It is interesting to note that such documentation can be further annotated with comments for each element present in the system, using "notes" that may include important explanations. These notes are rendered as comments in the actual code generated from the model. A more detailed description of the Sparx EA modelling steps is presented in Appendix A – More details on Sparx Modelling. The Sparx Enterprise Architect model together with the generated documentation, generated code and a video presentation demonstrating Sparx EA at work on the 4SECURail case study is available at **[ZenodoWP2]**.

5.3.3 ProB modelling

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

B state machine. This has several implications:

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

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

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

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

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

```
       UMC transition                           Event-B Operation

R8_ICSL_IRBC_rbcuserconnectindication:      R8_ICSL_IRBC_rbcuserconnectindication =
NOCOMMS -> COMMS                               PRE
 { SAI_CONNECT_confirm /                         ICSL_STATUS = NOCOMMS &
   RBC_User.RBC_User_Connect_indication;         icsl_buff /= [] &
   connect_timer := max_connect_timer;            first(icsl_buff) =  SAI_CONNECT_confirm
   receive_timer := 0;                         THEN
   send_timer := 0; }                            ICSL_STATUS := COMMS;
                                                 irbc_buff := irbc_buff <- RBC_User_Connect_indication;
                                                 icsl_connect_timer := icsl_max_connect_timer;
                                                 icsl_receive_timer := 0;
                                                 icsl_send_timer := 0;
                                                 icsl_buff := tail(icsl_buff)
                                               END;
```

Clearly the mapping between UMC and ProB appears to be particularly direct because we are restricting ourselves to use only basic state machine features. For example, deferred events, completion transitions, parallel states, entry-do-exit activities, pseudo-states and other modelling facilities of UML State Machines are not used.

This choice was one of our design principles for:

- allowing the design of state machines with a clear meaning not particularly tied to UML technicalities
- allowing an easy transformation into other notations, not related to UML, for formal analysis and verification.

The generated models and related audio-visual material are available in **[ZenodoWP2]**.

Once the ProB model has been generated, all its possible executions can be interactively simulated, and the whole state space can be exhaustively checked for absence of errors, deadlocks, or invariant violations. Furthermore, properties in the form of Linear Temporal Logics (LTL) or Computation Tree Logics (CTL) can be verified and counterexamples saved and visualized.

It is important to have an intuitive graphical representation of counterexamples (or in general execution traces) in terms of the signals exchanged among the various components. Thus, we have developed a *prototype* translator that converts a ProB textual execution trace into a sequence diagram (thanks again to the PlantUML platform **[PlantUML]**). For example, the confirmation of the actual reachability of transition R8_ICSL (the same of Figure 4) can be model-checked, and the resulting trace visualized as shown in Figure 11:
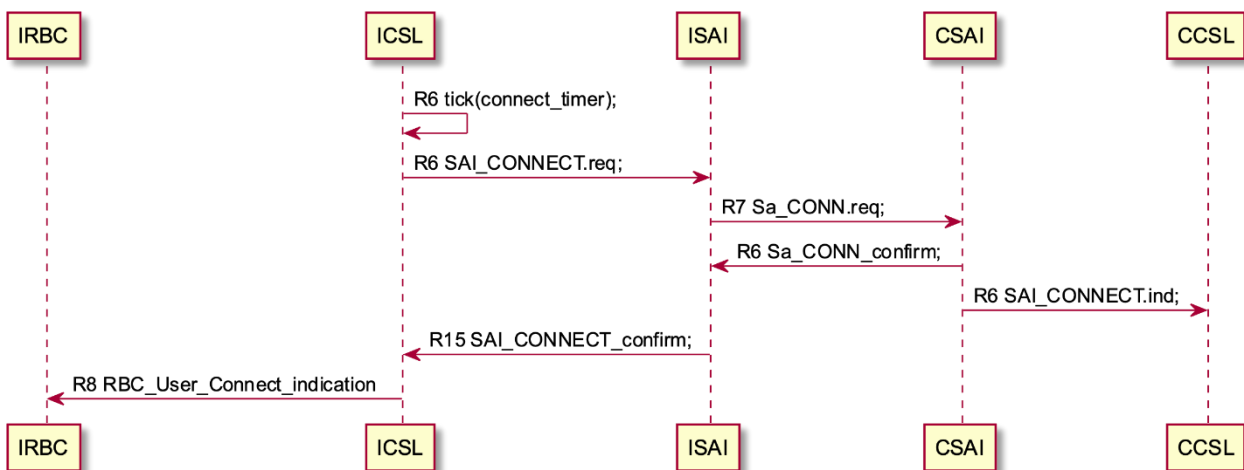
Figure 11 : Sequence diagram of a ProB execution trace

The ProB encoding of the I_CSL component is shown in Appendix D - ProB encoding of I_CSL.

5.4 Formal Analysis of the initial case study fragment

In this section we present an overview of the formal analysis which has been done on the CSL components of the case study.

In Section 5.4.1 we discuss the implications of having selected the UML as a standard notation for the design of the system, which in our case is constituted of six mutually asynchronously interacting state machines.

In the subsequent Section 5.4.2 and Section 5.4.3 we present, with the help of some examples, the impact of our semi-formal and formal modelling and verification steps towards

- a deeper understanding of the designed system,
- a potential improvement of the initial natural language requirements,
- a greater confidence in the robustness of the requirements.

We emphasise that the goal of our activity is to demonstrate the process of developing system specifications using semi-formal/formal methods and to identify costs and benefits involved in it, rather than an exhaustive validation of a given system specification.

5.4.1 The impact of using the UML as a "standard" notation for the description of the composition of State Machines

Each of the system components shown in Figure 3 has been modelled by its own UML state-machine as described in Section 5.3. It becomes essential, however, to clarify the role and impact of using UML as a "standard" notation for describing this kind of composition of state machines.

When reasoning in terms of a *single* state machine, the UML semantics of Run-To-Completion steps is precisely defined by the OMG standard (see e.g., fUML **[OMG-fUML1] [OMG-PSSM]**, Alf **[OMG-Alf] [OMG-Alf-Spec]**), although not in mathematical terms but using natural language, and at least when appropriate restrictions are made to avoid more complex and still unclear situations **[SG30]**. "Precisely", however, does not mean "complete".

The OMG UML **[OMG-UML]** definition leaves intentionally unspecified several aspects of the behaviour of a state machine. For example:
- which transition is fired when multiple transitions are in conflict, that is they have the same priority and are enabled by the dispatching of the current event,
- which outgoing path from a choice pseudo node is selected when multiple alternatives are enabled,
- in which order parallel transitions are fired.

Each model-based development tool that is compliant to the UML standard (like Sparx Enterprise Architect) can make its implementation choices and define its (possibly deterministic) specific execution model.

If we want to adopt a semantics of a UML state machine design that is as far as possible independent from any implementation choice, we can model the above aspects as nondeterministic aspects.

In our case, due to the precise behaviour expected from the CSL components, these nondeterministic aspects (i.e., conflicting transitions) are not present in the CSL state machine models.

However, due to the current abstraction of both SAI end EuroRadio layers, we have in our system some UML state machines that behave in a nondeterministic way (recall that is true for UMC and ProB, but not for Sparx EA where the human user acts as the environment). This nondeterminism is indeed introduced either from the UML modelling of some physical component (the communication mean that can lose, delay, reorder messages) or from the abstraction of the actual behaviour of other components (e.g., the abstraction of the actual presence of sequence numbers, sending time of a message, arrival time of a message). Some of these aspects will be refined in the next release of the Demonstrator.

Moreover, the overall semantics of interacting state machines is intentionally left undefined by the OMG standards to allow flexibility and adaptivity of the UML notation to different contexts.

From this point of view the most important aspects that are left undefined by the UML standard are:

- The characteristic of inter-machine communications (e.g., delay, loss of messages, reordering of messages during communications).
- How the "event pool" associated with each state machine is managed.
- How the various state-machines composing the system are scheduled and executed in parallel.
- Whether or not a single Run-To-Completion step of a state machine can be interrupted or overlapped with other Run-To-Completion steps of other state machines.

It is not possible to verify the behaviour of interacting state machines in its complete generality. As an example, we can imagine pools that select events to trigger a Run-To-Completion step in a random way, pools that keep the event as a FIFO queue (this is the UML default choice), pools that handle the arrived events according to some internally defined priority mechanism, pools that are bounded in size and that discard messages when full, and so on.

Among the many possibilities, we believe that the most interesting and useful choice is to rely on the UML default choice, unless someone has the goal of specifying the semantics of a particular class of systems which is known to reflect a different assumption.

Therefore, any attempt to associate a meaning to a composition of UML state machines design should as first step clarify the assumptions which underlie this design.

In our case, the specific assumptions underlying our UML design for what concerns the UMC modelling and ProB encoding of the system are that:

Assumption 1. During interstate machine communications, the sending of an event from one side corresponds to the receiving of the event in the event pool of the other side; therefore, communication events are not delayed, lost, or reordered.

Assumption 2. The event pool associated with each state machine is an (unbounded) FIFO queue.

Assumption 3. The Run-To-Completion steps of a state machine are executed atomically also in the context of System-of-Systems.

Assumption 4. The scheduling of the various state machines is considered free and unconstrained.

Some remarks are needed to clarify the above assumptions.

- *Assumption 1* reflects the case where communications occur via shared memory (e.g., by writing into a buffer). If this is not the case (as in our model the ISAI-EuroRadio-CSAI interactions), we must explicitly model the existence of a "communication layer" which introduced delay, loss, or reordering of messages.

- *Assumption 2* for modelling the event pool with FIFO queues directly reflects the default policy suggested by the UML standard. If the system being modelled is based on different assumptions, then this fact should be made clear, and consequently explicitly modelled/specified within the formal framework (e.g., UMC allows us to specify that the event pool should behave as a random set of events, but this aspect is not used in our models).

- *Assumption 3* is a simplification that reflects well the case of the atomic execution of operations that are not interrupted by the scheduling policy adopted within the architecture that we are trying to model. If the underlying scheduling policy does not reflect this assumption, then we should explicitly split our non-intended-atomic transition into a sequence of atomic ones.

- *Assumption 4*, which in its generality models any possible interleaving of state machines, implies that the resulting system might be unfair, with certain machines evolving while others not progressing at all. This implies that additional components might have to be added to the system to constrain how the various machines can proceed in parallel. In our case we have modelled all the system components as continuously executing a cycle, and added a "Timer" component acting as explicit scheduler that allows all the cycles to be executed in parallel, in all possible orders, but reflecting the property that no machine executes more cycles than another. Alternative, more specific assumptions might have been made: e.g., assigning a (possibly dynamic) priority to the various executing state machines, or scheduling state machines according to some notion of global time and duration of the Run-To-Completion. Our choice, which is the most general, is also the one that underlies the UMC design and the simplest one to encode in terms of ProB machine.

In conclusion, our reference model is not a perfect model of the actual real-time system, but it is a convenient abstraction that allows to reason on the possible properties of the actual system.

5.4.2 Semi-formal analysis

A first consideration is that already the initial step of building an operational model of the system (as UML state machines) has revealed duplications, implicit assumptions, ambiguities and unclarities in the natural language system requirements specified by Deliverable 2.3 of 4SECURail project.

Let us consider, for example, REQ_001:

REQ_001	If configured as initiator, when switched on (communication in state NOCOMMS), the CSL is responsible to send to underlying Layers the command for the establishment of a safe connection with the partner RBC, and to command re-establishment of safe connection when it is considered lost (communication in state NOCOMMS).

This requirement is found to be overlapping with REQ_012, which says:

REQ_012	If configured as initiator, at start-up, and when the loss of safe connection is detected, the CSL shall send safe connection init order to SFM (SAI_CONNECT.request).

Moreover, it is not clear if the SAI_CONNECT.request should be sent immediately as soon as initiator CSL is switched on, or immediately as soon as it returns to the NOCOMMS state, or if some delay is allowed (is not clear whether any allowed maximum delay exists at all).
We argue that it is not desirable to write the system requirements at such a level of detail (stating explicitly that some small delay is allowed, but no more than a certain amount). However, an explicit note in the form of INFO would not do any harm. In our CSL model the SAI_CONNECT.request is sent at the beginning of the next cycle.

Continuing to analyse REQ_001, we note that it requires that, when the communication is in state NOCOMMS, the CSL should send the SAI_CONNECT request. However, after having sent that request, the CSL still remains in state NOCOMMS and should not continue to send again the same request (this should be done only after the expiration of the connection timeout, as specified by REQ_002).

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

In this case REQ_012, while requesting the same behaviour, is more explicit in suggesting that the

SAI_CONNECT request should be sent (apparently only) initially, and when "loss of safe connection is detected". This is further clarified in REQ_006 and REQ_007:

REQ_006	Loss of safe connection shall be detected by the CSL reading reports from the underlying SFM (SAI_DISCONNECT.indication).
REQ_007	If a report from underlying Layers is received that safe connection is lost, the CSL shall consider the communication in state NOCOMMS.

But there is another requirement (REQ_009) that mandates a transition back to state NOCOMMS even without mentioning any "loss of safe connection" (which can still be alive):

REQ_009	After reception of report from the SAI that the clock offset procedure or EC initialisation has been completed, the condition where no valid messages are received within a configurable time shall be recognised by the CSL. This is achieved through a configurable receive timer (started at the reception of report from SAI on completion of initialisations and reset at the reception of any message); if no message (User or life sign) is received within such configurable receive time interval, the communication shall be considered in state NOCOMMS.

Summing up, from all these requirements it seems clear that when a CSL receives a SAI_DISCONNECT.indication or when the receive timer expires, the CSL should switch to state NOCOMMS and, if it configured as initiator, it should send the SAI_CONNECT.request.

The next question one may ask is: what if the SAI_DISCONNECT.indication is received when already in the NOCOMMS state? It is reasonable to imagine that in this case the report is not "read" but just "discarded", without any resending of SAI_CONNECT.request (until the expiration of connection timeout).

To complete the picture, we should also consider the cases in which the state NOCOMMS is left: and re-entered. This is specified by REQ_011, REQ_013 and REQ_014:

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

REQ_013	If configured as initiator, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.confirm).
REQ_014	If configured as called, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.indication).

We can see that the above natural language presentation of the system requirements, whilst

describing what the system should do, requires a certain effort in merging the various fragments and aligning the various overlapping of requirements.

We argue that, if a state machine model of the CSL would be available, probably it would have been more natural to state the required behaviour in a concise, clear way, like:

- when the initiator CSL enters in state NOCOMMS (either at start-up or when coming from state COMMS) it shall send a SAI_CONNECT_request to SFM and start a connection-timer.
- when the initiator CSL is in state NOCOMMS, if the connection-timer expires then it shall resend a SAI_CONNECT.request to SFM.
- when the initiator CSL is in state NOCOMMS and receives SAI_CONNECT.confirm it shall move to COMMS.
- when the called CSL is in state NOCOMMS and receives SAI_CONNECT.indication it shall move to COMMS.
- when the (initiator or called) CSL is in state COMMS, if receive-timer expires or a SAI_DISCONNECT.indication is received, then it shall switch to NOCOMMS.

5.4.3 Formal analysis

While certain weaknesses of the natural language requirements can be spotted already in the UML design phase, other subtle weaknesses can be detected more easily only if an automatic model exploration/model checking is done.

Model checking, for example, may reveal missing requirements. This can be put in evidence by the fact that the state machine deadlocks or discards events (i.e., a signal arrives in a certain component state, but there is no explicit rule handling it).

During the initial design of new components (e.g., the RBC_Users modelling a scenario) it is quite easy to initially overlook certain possible system evolutions.

For example, with respect to the above quoted REQ_012, we can observe that the "detection of loss of connection" (i.e., the SAI_DISCONNECT.indication) can occur when the RBC is already in state NOCOMMS. In this case the indication should be discarded and no SAI_CONNECT.request should be issued until the appropriate connection timeout is expired.

An example of how this situation might occur can be seen by observing the generated Sequence Diagram, obtained by checking a formula (either in UMC or Prob) proving that this situation (modelled by rule R2_ICSL) might occur.

As shown in Figure 12 the generated Sequence Diagram shows that this event might occur if both the ISAI and the ICSL autonomously decide (at almost the same time) to move to the disconnected state, the SAI because it has received a Sa_DISCONNECT.indication, and the RBC because its timeout is expired. Both components in this case send a disconnection notification to the other, which is received when already in the disconnected state.
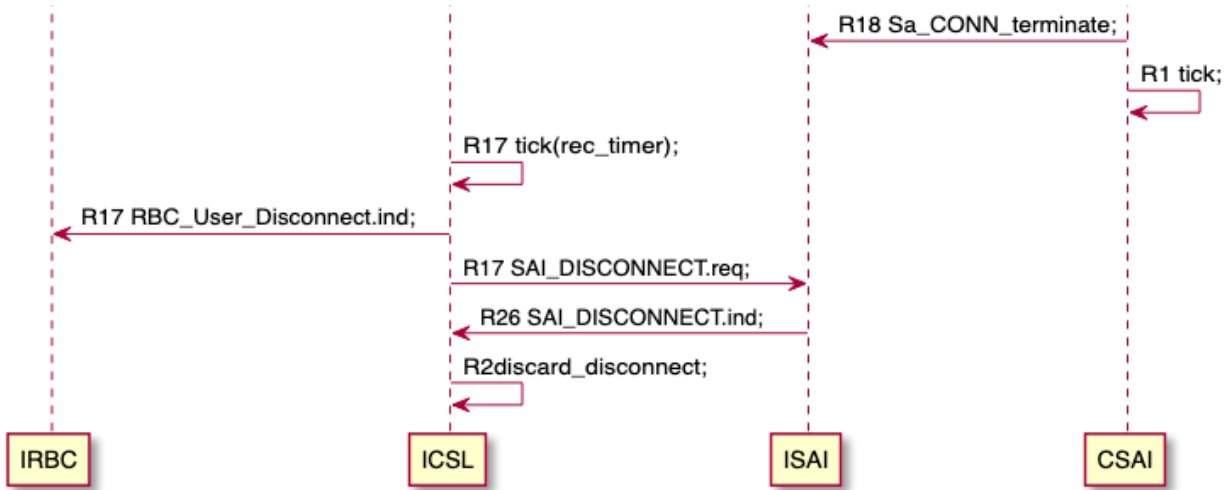
Figure 12 : ICSL and ISAI exchange disconnect messages

We have shown so far just one example of the deeper insights and benefits that can be obtained by an early formal analysis of a UML design: deeper insights that might lead to the construction of more complete and clear system requirements specifications.

In Appendix B - Annotated CSL Requirements and Mapping we show the initial version of the D2.3 requirements related to the CSL subsystem, annotated with observations resulting from this initial phase of modelling and analysis of the CSL specification. This Appendix also shows the direct and inverse mapping between the natural language requirements and UMC-ProB rules formally modelling their impact on the system behaviour. On the other hand, the mapping from Sparx EA transitions (or rules) to UMC-ProB rules is in Appendix E - Sparx Model Report – CSL.

Using the selected scenarios several generic properties of the formal model can be verified, often at just the price of a single click:

- The modelled systems have no deadlocks (i.e., the two CSL components are always live).
- All transitions present in the CSL state machine diagrams are eventually fired, (i.e., there are no dead transitions in the CSL state machine diagrams).
- No signals/event are ever lost (i.e., there are no missing transitions in the CSL state machine diagrams).

Figure 13 shows the ProB default interface for model checking the system, and Figure 14 shows the coverage report that can be analysed after the verification.

The actual reachability of each UML transition in each scenario (i.e., if a transition will be executed) can be explicitly checked through model checking, and a Sequence Diagram can be generated that illustrate a viable way in which the transition is reached.
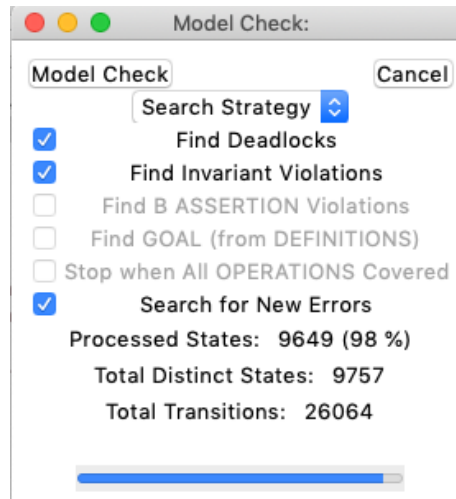
Figure 13 : Model checking interface of ProB
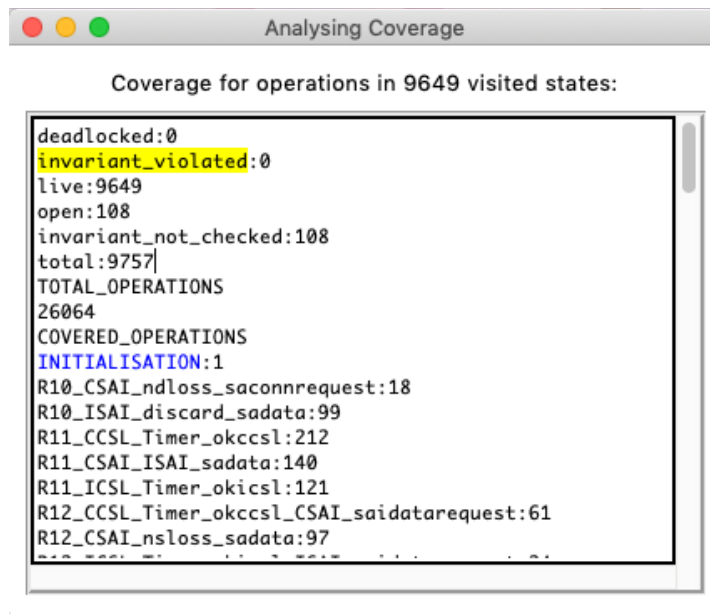
Figure 14 : Operations coverage summary

An interesting point that we should address is: how can we be sure that the formal model is a good representation of our requirements?

Different answers to the above question may be given: by tracing requirements to the formal model and vice versa, from the analysis of the read-write table[1] that can be automatically generated by ProB, and from a manual inspection of the model and requirements.

---

[1] A map that associates each rule with the variables used (reading or writing) by that rule.

If requirements are mapped to properties expressed in the form of temporal logic, formal methods (in particular, model checking) can be used to automatically check that the model satisfies the properties, that is, the model is conformant to the requirements.

Let us consider the fragment of REQ_009 (shown previously) stating that if the receive-timer expires the CSL should move into NOCOMMS state.

We can verify the Linear Temporal Logic formula[2] stating that it never happens that in the initiator CSL (the property for the called CSL is equivalent) the receive-timer is greater than the maximum allowed value. This property will be checked exhaustively over all possible system executions (i.e., by generating the so-called state-space).

We can also verify more complex conditions. For example, it is possible to check whether, assuming that the CSL is connected and no data has been received from the partner RBC within maximum allowed time, at the next time slot either a data or disconnect command is received from SAI, or the CSL will close the connection[3].

By expressing through temporal logic formulae all behaviour characterising functional requirements of the CSL, it is possible to automatically check them on an arbitrary model of the CSL to check if it is conformant to the requirements. This type of formal verification is a step towards more automatization with respect to full manual validation and is also equipped with formal guarantees that all executions of the model will respect the desired properties.

This way of checking the model conformance surely is of help in complementing other validation practices described above.

Another interesting question might be: how can we ensure that the adopted tools faithfully model the expected behaviour of the provided UML design (under the chosen assumptions in terms of UML state machine interactions)? I.e., is our ProB approach to the modelling of UML designs correct?

In this Demonstrator we used two formal tools that explore all executions of the (composition of) UML state machine models. These tools are ProB and UMC. We can automatically verify that their generated state-space coincides[4]. This proves that the ProB approach towards the modelling of interacting state machines reflects the UML assumptions that underlie the UMC behaviour, and that the translation of the UMC code into the ProB code is correct (i.e. we have a mathematically grounded proof that the behaviour of the two models is equivalent). Still, even though both ProB

---

[2] G (({icsl_receive_timer <= icsl_max_receive_timer}))

[3] G (({icsl_receive_timer = icsl_max_receive_timer}) =>
    (({ICSL_STATUS = COMMS}) &
    ((not ([R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest]))
    U
    (((([R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest])
    or ([R16_ICSL_IRBC_rbcuserdisconnectindication]) => X({ICSL_STATUS = NOCOMMS}))
    or (((([R13_ICSL_IRBC_rbcuserdataindication]) or ([R14_ICSL_handle_lifesign]))
     => ((({icsl_receive_timer = 0}) & ({ICSL_STATUS = NOCOMMS}))
))))))

[4] This has been achieved by asking the two tools to save their state-space as labelled transition systems, transforming these state-spaces into the .bcg format of the CADP **[CADP]** framework and comparing them with the bcg_cmp tool.

and UMC models are proved equivalent, we may ask whether this is the case also for Sparx model. Indeed, even if Sparx and UMC support the same UML design, their semantics is allowed to be different by the UML Standard. Since a semi-formal tool as Sparx does not offer the above type of formal verification, the semantics compliance between Sparx and UMC has been manually validated, to check whether it satisfies the initial intuition on the model behaviour.

Reasoning at a higher level on the composed CSL-SAI behaviour

In this project the functional CSL-SAI requirements have been extracted from well-established UNISIG standards so we can reasonably trust that the specified requirements satisfy some global system expectations.

If, from the point of view of the system developer, producing a system which complies to the requirements is in principle all that is needed, from the point of view of the Infrastructure Manager it is of paramount importance to confirm that all the possible system behaviours satisfy certain higher-level requirements, which constitute the base for successful interoperation of the whole system.

In the case of the SAI subcomponent, these higher-level requirements essentially coincide with the safety requirements of the SAI component, while in the case of CSL these higher-level requirements are not explicitly stated.

FM are a possible and reliable way to support this task. In our case, let us consider the full system composed of the EuroRadio level, the two SAI levels, and the two CSL levels.
We might ask:
> *which properties of our full system can the two RBC_User trust (or should not trust) to design their handover protocol (or whatever other functionality) based over the CSL layer?*

Notice that we are not making hypotheses on the possible behaviour (or requirements) of RBC_Users, but we are just trying to better understand what in the end is the behaviour of our CSL+SAI+Euroradio system.

Some answers to this question can be given quite immediately. E.g.
- the composed system <u>does not guarantee</u> that a successful connection can be achieved (and this is acceptable, e.g., if the wire connecting the two sides is broken),
- the composed system <u>does not guarantee</u> that a User message sent from one side is actually delivered to the other side (and this is acceptable, because the message might be lost),
- the composed system <u>guarantees</u> that if one RBC sends two messages to the other RBC, the two messages, if received, are received within a certain timeout and in the correct order (actually the correct ordering is supposed to be guaranteed by the SAI component, and the CSL layer does not introduce reordering).

Additional answers can be given with some simple simulation of the system. E.g.

- the full system <u>does not guarantee</u> that two RBC always have the same view on the status of the safe connection (for a limited time their views can be different, and not matching the actual SAI status).

Other answers might be less obvious to check, especially if they depend on the specific configuration parameters with which the system components have been instantiated. In this case the model checking of the system may be of help in providing some clarity.

For example, we might ask:
"*Does the full system guarantee that, once successfully connected, an exchange of messages between two sides[5] will always eventually have success or a disconnection will occur?*"

We can see, by an exhaustive automatic formal analysis of the system[6], that the answer is negative. For example, it is sufficient that the message is unfortunately lost each time it is sent. If the SAI component is configured with parameter N set to 2[7], if the subsequent lifesign is successfully delivered then no disconnection would occur and no reply would ever arrive.

We can therefore analyse what happens if we set the SAI parameter N to 1. Also, in this case the above property is <u>not guaranteed</u>, because the SAI of destination may receive the message, find it "invalid" (i.e., arrived too late) and discard it, without delivering it to the higher CSL and RBC levels. Again, we would have no reply nor disconnections.

If we assume instead that no invalid messages are found, N=1 and connections do always have success, then we can see that finally the above property is satisfied.

This kind of analysis of the overall system behaviour is interesting as it may put in evidence useful insights into the system behaviour and suggest indications for the system configuration and use.

E.g., in our case we might for example observe that:
- Too frequent sending of lifesigns might be dangerous as they can hide the loss of more important NRBC messages.
- Generally, it is possible to automatically analyse the behaviour of the system at the varying of parameters such as N or send_timer, to perform a sensitivity analysis on properties as the one described above,
- Since the RBC_User, when sending an RBC message, in our case study cannot rely on an explicit feedback on the fact that the message has been actually delivered or not delivered

---

[5] I.e. one side send an initial message and the other message sends a reply
[6] e.g. by checking the LTL formula "F (([R9_IRBC_aborted]) or ([R10_IRBC_done])) on the Scenario 2 described in Section 5.2
[7] The parameter N of SAI specified the maximum number of successive messages which be accepted to be lost without terminating the safe connection.

or delivered as "invalid"[8], it might become necessary, also at the higher layers levels, to reason again in terms of timeouts for detecting that something has gone wrong at the lower levels of communications and therefore resending the messages which have had no reply.

- Also, the strategy of sending RBC messages in chains of N repetitions might not solve the problem, because in the unfortunate case that all the RBC messages are marked as "invalid" and the exchanged life-signs were instead successfully delivered, we would continue to have no disconnections and no successful delivery of messages.

The above considerations should be considered just examples of the insights that formal modelling and verification can provide, with no intention to provide official feedback or comment of the system requirements specification being analysed, nor to suggest requirements on the outer RBC_User levels which are clearly out of scope.

During this analysis only the CSL layer has been modelled completely while some aspects of the SAI have been abstracted away by modelling them in a simpler (and nondeterministic) way. In the next release of the Demonstrator a deeper and more rigorous modelling and analysis of the SAI will be present.

---

[8] Subset-98 [SUB-098] leaves to the implementation the error handling procedure.

6 Conclusions

Context, goals, approach, and results of the activity

The activity described in this Deliverable is part of the activity of Task 2.1, whose objective is the definition of a formal methods Demonstrator process (D2.1) to be applied (D2.2, D2.5) to a selected signalling case study (D2.3) to demonstrate the costs and benefits of the use of formal methods (D2.4, D2.6) in the requirements definition phase.
This Deliverable (D2.2) is related to the second part of T2.1 and has the goal of demonstrating and validating with a trial case study from D2.3 the formal-methods based specification-design process defined in D2.1.

The formal (model checking) and semi-formal (MBSD) process adopted in our Demonstrator process starts with the design in terms of UML state machines of the systems being specified. This semi-formal design is then translated to more rigorous formal notations that can be used for exhaustive formal analysis.

An analysis of the kind of difficulties and efforts needed for passing from a set of natural language requirements to a formal model and a set of formally verifiable properties is an activity which needs further investigation and that will be more thoroughly addressed in the future deliverables of the project.

The choice of the initial UML-based design has been done in agreement with what has been done in EULYNX and X2RAIL2, where UML/SysML is used as a standardised notation for the semi-formal design of system specifications.
The tools which have been used so far (UMC, ProB, Sparx EA) have been applied with a limited effort and cost, exploiting features among those provided that were necessary for our initial experimentation. Becoming an expert in the use of these tools and mastering all their many functionalities is outside the goals of this project.

Our experience has shown that the specification of a single UML state machine, when used with certain constraints, may well fit the need to give standardized definitions of system components that are easily translatable in other formal notations.

In the remaining phases of the 4SECURail project, this demonstrator process will be applied in Task 2.3 on the complete case study defined in Task 2.2 and that activity will allow to verify the meaningfulness of costs and benefit *categories* at the base of the CBA.

Potential future extensions and recommendations

In our design of the Demonstrator we expressed, as particularly desirable (but currently out of

reach) the possibility to rely on a single MBSD framework for both design, code-generation, documentation, and formal verification. Our experimentation has shown that this desire is currently out of reach, especially if a semi-formal language such as the UML is kept as reference underlying notation. This remains an important direction to be further explored.

With this experimentation, the FM Demonstrator process defined in D2.1 has proved to be in line with the goals required by Task 2.3. This, however, does not exclude that in the project continuation additional FM will be taken into consideration due to the specific characteristics of the signalling system case study to be analysed in Task 2.3.

An interesting side product of the future activity planned for Task 2.3 might be the generation of a revised version of the systems requirement, still in natural language, but more rigorous and more related to a formal model.

As a general recommendation for future projects and research activities towards the adoption of FM in industry we observe that it might be worth to promote the development of the standard UML profiles (i.e., where no semantics choices are left free) oriented to the specification of Systems of Systems, that standardize the characteristics of inter-state-machine interactions. This is one of the weakest points for the successful use of the UML as a standard notation for Systems of Systems.

It might be also worth promoting the design of a basic UML profile for state machines, which simplifies their behaviour and reduces the degree of their undefined aspects, to propose and make easier standardised translations from UML state machines to other formal notations.

7 References

[3DS] 3DS Catia nomagic "Cameo Systems Modeler"
 https://www.nomagic.com/products/cameo-systems-modeler

[ASTRAIL] ASTRail project. https://projects.shift2rail.org/s2r_ip2_n.aspx?p=ASTRAIL

[ASTRAIL-D41] ASTRail Deliverable D4.1 "Report on Analysis and on Ranking of Formal Methods"
http://astrail.eu/download.aspx?id=bb46b81b-a5bf-4036-9018-cc6e7d91e2c2

[ASTRAIL-D43] ASTRail Deliverable D4.3 "Validation Report"
http://astrail.eu/download.aspx?id=d7ae1ebf-52b4-4bde-b25e-ae251fd906df

[Bmethod] Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F. and Voisin, L., 2020, September. The first twenty-five years of industrial use of the B-method. In International Conference on Formal Methods for Industrial Critical Systems (pp. 189-209). Springer, Cham.

[CADP] Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R. and Sighireanu, M., 1996, July. CADP a protocol validation and verification toolbox. In International Conference on Computer Aided Verification (pp. 437-440). Springer, Berlin, Heidelberg.

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

[EPL] http://www.eclipse.org/org/documents/epl-v10.html

[ERA] European Union Agency for Railways https://www.era.europa.eu/

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

[EVB] Event-B.org Website http://www.event-b.org/

[FORMALMIND] https://www.formalmind.com/

[FSKR29] Fecher H., Schönborn J., Kyas M., de Roever WP. (2005) 29 "New Unclarities in the Semantics of UML 2.0 State Machines" In: Lau KK., Banach R. (eds) Formal Methods and Software Engineering. ICFEM 2005. Lecture Notes in Computer Science, vol 3785. Springer, Berlin, Heidelberg https://doi.org/10.1007/11576280_5

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

[ICSE2020] Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H. and Fantechi, A., 2020, June. Comparing formal tools for system design: a judgment study. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 62-74).

[MAAP2015] Shift2Rail Multi-Annual Action Plan 2015 https://shift2rail.org/wp-content/uploads/2013/07/S2R-JU-GB_Decision-N-15-2015-MAAP.pdf

[MAAP2017] Shift2Rail Multi-Annual Action Plan – Executive View - Part A (2017) https://shift2rail.org/wp-content/uploads/2018/04/Maap_2018_FINAL_2.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

[MBSD] Krogstie, J., 2012. Model-based development and evolution of information systems: A Quality Approach. Springer Science & Business Media.

[OMG-Alf] OMG "Action Language for Foundational UML (Alf)" https://www.omg.org/spec/ALF/1.1

[OMG-Alf-Spec] OMG "Alf Specification" https://www.omg.org/spec/ALF/1.1/PDF

[OMG-fUML1] OMG "Semantics of a Foundational Subset for Executable UML Models (fUML)"

https://www.omg.org/spec/FUML/1.4

[OMG-PSSM] OMG "Precise Semantics of UML State Machines (PSSM)", May 2019 https://www.omg.org/spec/PSSM/1.0

[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/

[PROB-licensing] https://www3.hhu.de/stups/prob/index.php/ProB_for_Event-B

[PTC-Alstom] https://www.omg.org/mda/mda_files/PTC_Alstom.pdf

[PTC-Windchill] PTC "Windchill Modeler SySim" https://www.ptc.com/en/products/plm-plm-products/windchill/modeler/sysim

[SG30] Anthony J.H. Simos, Ian Graham, "30 Things that go wrong in object modelling with UML 1.3." In: Behavioral Specifications of Businesses and Systems. The Springer International Series in Engineering and Computer Science, vol 523. Springer, Boston, MA https://doi.org/10.1007/978-1-4615-5229-1_17
http://staffwww.dcs.shef.ac.uk/people/A.Simons/research/papers/uml30things.pdf

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

[SPARXCaseStudy] "SparxSystems CE: Belgian Railway relies on Enterprise Architect", https://community.sparxsystems.com/case-studies/1189-snfc4ea

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

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

[UIC] European Union Agency for Railways https://uic.org/

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

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

[UMLReview] https://umltoolreviews.com/review-sparx-ea-uml/

[UMLSurv] Ozkaya, M. and Erata, F., 2020. A survey on the practical use of UML for different software architecture viewpoints. Information and Software Technology, 121, p.106275.

[UNISIG] UNISIG is an industrial consortium factsheet http://www.ertms.net/wp-content/uploads/2014/09/ERTMS_Factsheet_8_UNISIG.pdf

[WHATISUML] Visual paradigm, "What is Unified Modeling Language (UML)"? https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

[X2R2D51] X2Rail project, Deliverable D5.1 "Formal Methods (Taxonomy and Survey), Proposed Methods and Applications" https://projects.shift2rail.org/download.aspx?id=b4cf6a3d-f1f2-4dd3-ae01-2bada34596b8

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

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

8 Appendix A – More details on Sparx Modelling

In this section the Sparx modelling phase is detailed.

8.1 Simulation of Executable State Machines in Sparx EA

In Sparx EA, Executable State Machines are artifacts used to compose several separate state machines and make them interact using simple send primitives. They are used to perform simulations of interacting state machines.
 To do that, source code is automatically generated from such models, that is then executed/debugged. It is possible to generate source code in JavaScript, Java, C, C++ and C#.
The source code also contains the implementation of the behavioural engine of State Diagrams, as for example the pool of events for each state machine, the dispatching method and so on.

Templates for code generation are provided and can be customized to specific needs. Inspecting the engine of State diagrams allows crucially to disambiguate the semantics choices left free by UML and fUML State Diagrams ISO standard **[OMG-fUML1][OMG-PSSM]**. In particular, the order of event dispatching, the strategy for solving conflicting transitions, the scheduling for dispatching among state machines.

Once the code is generated, it is possible to debug it visually, using facilities that connect the source code with the graphical diagrams, that are then animated and seamlessly integrated with the simulation engine of Sparx. The animation emphasises the current state of each entity simulated, and a simulation output console prints all relevant events that occur, thus constructing a trace of simulation.
Under the hood a program (in Java in our case) is being debugged, thus it is also possible to use breakpoints into the source code to perform standard debugging, inspecting the call stack to see which method is currently executed, as well as the current thread. More importantly, the values of variables of each state machine can be inspected at each step of simulation.

It is possible to interact with the state machine graphically, by clicking on one of the triggers that are available in the Simulation events window. Once a trigger is sent, the state machines animate and change the state according to that specific event.
It is also possible to interact with the state machines via console, typing instructions like send event to machine, used to send an event to a specific machine. Such instructions can also be put into a script to be automatized.
The simulation has a speed that can be set. If the speed is set to zero, the simulation will pause after one Run-To-Completion step of all state machines. If the speed is greater than zero, the simulation will pause only when an external trigger from the user is needed, otherwise it will continue its execution at the specified speed.

Breakpoints can also be used as markers for start and stop the recording of simulation. From this simulation recording, it is possible to extract a sequence diagram showing how the various objects

invoke each other's method.

By focussing on a single state machine, it is also possible to use the standard simulation engine provided by Sparx Enterprise Architect. In this case, the standard simulation will not generate executable code but will interpret the state machine. Simulations can be interpreted or manual. In the manual simulation, the user decides which transitions to fire. Interpreted simulation interacts using simulation events.

Concerning how suppliers can exploit this framework, thanks to Sparx Enterprise Architect the informal specification becomes an executable artifact that can be directly imported by the suppliers into their development process. This will facilitate the task of validating the correctness of the suppliers' implementations with respect to the IM' specifications. Indeed, suppliers shall link the provided specification to lower layers of implementation using tracing, so embedding the specification into their whole development process and combining it with specifications of other components/layers of architecture. Model-based Testing can also be used for testing whether the specific implementation adheres to the requirements expressed by the specification. The specification itself has already been formally verified by the IM against informal requirements. Other functionalities that can be exploited regard the automatic generation of documentation, and requirements traceability into the model. All these functionalities are available in Sparx Enterprise Architect.

8.2 From UMC model to Sparx Executable State Machine

UMC supports a subset of UML State Machine Diagrams, polished from syntactic sugar notation, and each construct can be mapped one-to-one to a construct in Sparx EA, as explained below.

It becomes possible to tie up rigorous state-of-the-art formal verification of state machine diagrams, as currently provided by academic and industrial tools, with all facilities provided by an industrial MBSD tool like Sparx EA.

The mapping from UMC model to Sparx Executable State Machine is almost straightforward. The following adopted restrictions on the model are exploited to keep the notation light and as much independent as possible from UML technicalities. Note that many of the following constructs are syntactical sugar that can be expressed using the adopted lighter notation:

- no *entry*, *exit*, *do* behaviour is present in the states of the model,
- interaction happens only using signals, and no operation calls are used,
- only one-to-one interactions are used, i.e., no signals broadcast,
- conflicts in enabled transitions are removed from the portion of UMC model that is to be mapped to Sparx, and are only used for modelling the external environment,
- no timing behaviour is present, elapsing of time is explicated using a TICK event,
- no internal and local transitions are used,
- no hierarchical states are used,
- no history, fork, join and other syntactic sugar nodes are used.

The following rules are applied to mechanically translate a UMC specification into an Executable Machine Specification.

1. Each class in UMC corresponds to a class in Sparx EA.
2. Attributes of a class in UMC are mapped to attributes of the corresponding class in Sparx EA.
3. Each Object in UMC, with its variables' instantiation, is mapped into a Property of an Executable State Machine (i.e., an instantiation of class), to where the values of the attributes can be instantiated.
4. Both UMC and Sparx EA classes have a relation "has-a" with other classes, in such a way that every object has a reference to other objects to whom it is interacting with.
5. Each class in UMC is specified as a state machine. Similarly, in Sparx EA a classifier behaviour will be assigned to each class in the form of a state machine.
6. States of a machine in UMC are in one-to-one correspondence with states of a machine in Sparx EA, comprehending also composed states.
7. Transitions of a machine in UMC are in one-to-one correspondence with transitions of a machine in Sparx EA.
8. Signals that are attributes of each class in UMC are mapped to global trigger events in the Sparx model, accessible by each state machine. These events are of type Signal and the specification is a signal `sig` that has one attribute `arg` that will be used for value passing. Indeed, in this first Demonstrator release it only suffices to pass one argument.
9. Triggers, guards, and effects of each transition are in one-to-one correspondence in both formalisms, with the only exception of sending signals and value passing, as explained below.
10. In UMC the sending of a signal with parameters is performed using the instruction

    Object.Signal`(value)`

    where `Object` is the object argument, `Signal` is the signal invoked in that object, and `value` is the value to be passed as an argument.

    In an Executable State Machine, Properties are connected by *connectors* typed with the relation "has-a" coming from the class diagram. Each end of the connector identifies the partner of the communication.

    A send operation is performed with the macro

    ```
    %SEND_EVENT("TRIGGER.sig(value)",CONTEXT_REF(RECIPIENT))%;
    ```

where `TRIGGER` corresponds to Signal in UMC, and `RECIPIENT` corresponds to Object in UMC. `RECIPIENT` is the identifier provided in the corresponding connector end of the Executable State Machine.

In case values of signals must be accessed inside the guard or effect of a transition, in UMC this can be done with the instruction

Signal[index]

where `Signal` is again the received signal, and `index` is a natural number indexing the argument. If, for example, only one argument is passed then `Signal[0]` will read that argument (at index 0).

In Sparx EA this is done with the instruction

signal.parameterValues.get(`"arg"`)

where `arg` is the name of the attribute of the signal. We also note that this syntax is specific to Java code generation as used in the Demonstrator.

Starting from the model mapped from UMC, some further syntactic transformation will be performed in the Sparx EA model, without affecting the semantics correspondence with the UMC model.
A further Boolean attribute will be used as a configuration parameter. This parameter is unfolded in two different classes in UMC, one class corresponding to Boolean value `true` and the other class corresponding to `false`.
The Sparx documentation reports, for each transition, the corresponding transition of the UMC model.

Semantics of Sparx Executable State Machines and UMC models

The semantics aspects left unspecified in UML State Machine ISO standard (e.g., events occurrence order) that are present in the model, are validated to be the behaviourally equivalent in both Sparx EA and UMC models (e.g., events occur in a first-come first-served fashion). Other unspecified aspects of UML State Diagrams are simply avoided in the Sparx EA model (e.g., internal events are always deterministic), whilst are only restricted to non-determinism in the environment in UMC (e.g., uncertainty in wireless communications), thus making the semantics of the state machine diagrams unambiguous and behaviourally equivalent in both Sparx EA and UMC for what concerns the software artifacts.

Indeed, in Sparx EA it is possible to inspect and review the code generated by an Executable State Machine, and in particular the UML engine, to disambiguate the semantics choices left open by the UML standard. The composition of state machines in EA allows each state machine to have its own pool of events, as specified by the UML standard. The code generated though is single threaded. As such, there is no concurrency between state machines, and their scheduling is fixed. Each state machine must complete its Run-To-Completion cycle before another machine can start its own, and the dispatching order among various state machines is fixed.

On the other hand, UMC overapproximate this behaviour by allowing all orders of scheduling. It also interleaves non-atomic Run-To-Completions of different state machines. As such, the semantics of UMC *includes* the semantics of Sparx Executable State Machines, as well as all semantics obtained by changing the scheduling order.

If a safety property (nothing bad ever happens) will be verified to hold in the UMC model (and its mapping to ProB), then the property will also hold for the Sparx model.

We note that, the solution adopted by Sparx is not standard and each UML compliant tool can implement its own scheduling/dispatching policies, its own choice strategy. Different state machines could concurrently execute their own Run-To-Completion cycle, or if a scheduler is imposed, then the order of scheduling is left free by the standard.

Concerning how events are ordered in each pool of events, both UMC and Sparx EA use lists for implementing the pool of events and the dispatching of events, with a first-in-first-out policy.

Concerning strategies to solve conflicting transitions, no conflicts in enabled transitions are present in the model that will be mapped in Sparx EA (the CSL in this deliverable), thus there is no need to specify the choice strategy for conflicting transitions. Moreover, such choice is left open in UMC, thus overapproximating again all possible behaviours obtained by fixing a specific choice strategy.

The effects of each transition contain source code, Java code in our case. The effects will be limited to only use code for performing arithmetic operations on variables, sending signals and reading values as described above.

These restrictions on Sparx models are necessary to disambiguate the semi-formal semantics and proceed in the external formal verification using model checking.

8.3 UML Diagrams of the Case Study

In this first release of the Demonstrator, we only mapped in Sparx EA the Communication Supervision Layer modelled in UMC, whilst the Safe Application Intermediate sub-layer will be mapped in the final release and are now present as stubs.

Instead, the other levels that will be present in UMC specification, that are the RBC-USER and Euro radio, will not be mapped in Sparx EA as they are not part of the requirements of the case study.

Indeed, whilst in Sparx EA interactive simulation is performed, with the human user acting as environment, in model checking tools the environment (i.e., RBC-USER, Euro radio) must also be modelled to automatize the verification. We note that for Euro radio model we intend the various threats in wireless communications that the SAI will protect against, but not the protection against

masquerading and corruption, that is assumed to hold.

8.4 The Class Diagram

In UML and object-oriented modelling, class diagrams are used to depict the static structure of a system, showing its *classes*, their *attributes, operations,* and the *relations* with other classes.

As stated above, in this first release, only the CSL will be modelled whilst RBC-User and SAI are present as stubs. As such, each of these entities is represented as a class.

As stated previously, we will identify the *initiator* RBC and the *called* RBC using a parametric configuration, using a Boolean attribute *initiator*, that is present in both SAI and CSL class, as those are the classes affected by the attribute.

The requirements of the case study also model the presence of the following time *thresholds:*

- The maximum time allowed before the initiator CSL reissues a connection request to the SAI. This attribute is called `max_connect_timer`, whilst another attribute `connect_timer` is used to accumulate the time passed since the last connection request.

- The maximum time allowed before the CSL sends a *Life sign* to the other CSL, because no messages have been sent so far. This threshold is called `max_send_timer`, and the corresponding accumulator attribute is called `send_timer` and is reset every time a message is sent by the CSL.

- The maximum time in which a message must be received by the CSL, called `max_receive_timer`, and the corresponding accumulator is called `receive_timer` and is reset every time a message is received from the underlying SAI.

The accumulator variables are depicted as *private* (prefixed by -), because they do not take part in the overall configuration. The other attributes are *public* (prefixed by +).

Finally, the CSL class has an *association relation* of type *"has a"* **[OMG-UML]** with its corresponding SAI and RBC_USER, because messages will be sent to them by the CSL and thus it has a reference to them. Both relations are depicted as arrows, and are labelled with the name of the relation, and identifying also the target of the arrow with a label.

We stress again that in the current release, neither the SAI nor the RBC_USER will ever send signals to the CSL, because such signals will be sent interactively by the human user so mimicking the environment.

The class diagram is depicted in Figure 8.

8.5 The Executable State Machine

The Executable State Machine artifact (see Figure 9) contains *Properties,* that are instances of classes in the class diagram. There are two instantiations of each class: one for the *initiator* RBC and one for the *called* RBC. Each property initialises the public attributes of its class. The relevant

attribute is *initiator,* whilst the other depends on the specific simulation at hand. In particular:

- The initiator RBC is represented by three properties called `I_RBC_USER` of class `RBC_USER`, `I_CSL` of class `CSL`, and `I_SAI` of class SAI . Both `I_CSL` and I _SAI have attribute i ni t i at or set to `true` (to enhance readability, depicted in blue).

- The called RBC is represented by three properties called `C_RBC_USER` of class `RBC_USER`, `C_CSL` of class `CSL`, and `C_SAI` of class SAI . Both `C_CSL` and C_SAI have attribute i ni t i at or set to `false` (to enhance readability, depicted in blue).

The various properties of the Executable State Machine are connected using *Connectors.* Such connectors will be typed using the relations presented in the class diagram, and the type is visible in the Executable State Machine artifact, that is depicted below.

It is possible to see that each CSL will refer to its RBC_USER using the context reference "RBC_USER" as depicted in the end of the corresponding connector.

Similarly, each CSL will refer to its SAI using the context reference "SAI" as depicted in the end of the corresponding connector.

8.6 The State Machines

A *classifier behaviour* is used to assign a state machine to each class, representing the behaviour of that class. We start by describing the Communication Supervision Layer state machine. This state machine has been obtained through the mapping described previously.

Firstly, transitions depicted in red are belonging to the initiator, whilst those depicted in blue are belonging to the called.

This is indeed enforced by a guard `[this.initiator==true]` for the initiator and `[this.initiator==false]` for the called.

Each `SEND_EVENT` primitive declares the context reference as declared in the class diagram and executable state machine.

Other than that, the state machine is a straightforward mapping from the UMC state machine described in the document and is depicted in Figure 7.

The state machines for the RBC_USER and SAI are only used for receiving the messages sent by the CSL and are used as a stub for the context reference (see Figure 15 and Figure 16). To avoid depiction of arrows internal transitions are used, but there is no semantics difference in using external transition since there is no entry/exit/do behaviour in each state. Moreover, we recall that such state machines are not part of the specification and are just used as stubs to perform interactive simulation with the CSL.
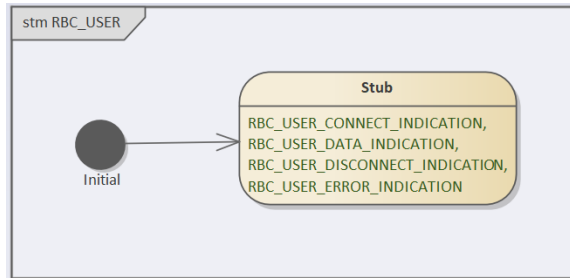
Figure 15 : RBC_USER State Machine Stub
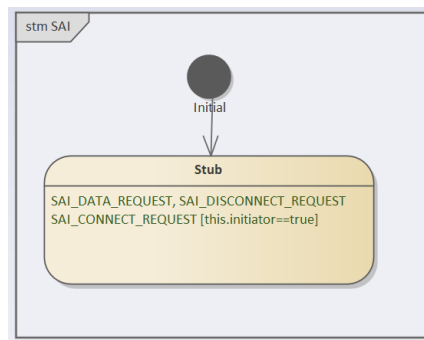
Figure 16 : SAI State Machine Stub

9 Appendix B - Annotated CSL Requirements and Mapping

This Appendix provides a mapping from the requirements of the case study to the State diagram model that has been developed.

For each requirement, the rule (i.e., the transition) modelling such requirement is reported. Additionally, we report remarks further clarifying the requirements. Finally, for each requirement we report issues that have been found through the (semi)-formal modelling phase. The identification of such issues during the first phase of requirements modelling represents an improvement over manual check of each requirement, thus showing benefits of modelling phase. The later these issues are discovered, the more the cost needed to amend them.

Once the model has been made, the evaluation has been structured according to the criteria below. Note that some issues are easy to find even without a modelling phase (e.g., multiple requirements) whilst others can really benefit from the construction of a model (e.g., ambiguity issues).

- *Ambiguity*: A requirement is considered ambiguous if several interpretations can be given to it, giving rise to different models with different behaviours, hindering interoperability.
- *Inconsistency*: A requirement may be in contradiction with another requirement or it could provide misleading information.
- *Incomplete*: The modelling phase may discover corner cases where the behaviour to be modelled is not completely covered by requirements.
- *Redundancy*: A requirement is considered redundant if it contains information already described by other requirements. Such redundancy increases the complexity of the requirement document, making the model less manageable and increasing the possibility of errors.
- *Irrelevant*: A requirement is irrelevant if it specifies a part of the system that is not supposed to be covered by this set of requirements. An irrelevant requirement unnecessarily augments the complexity of the requirements document.
- *Multiple*: A requirement may contain a conjunction of different requirements that should be listed individually.

REQ_001	If configured as initiator, when switched on (communication in state NOCOMMS), the CSL is responsible to send to underlying Layers the command for the establishment of a safe connection with the partner RBC, and to command re-establishment of safe connection when it is considered lost (communication in state NOCOMMS).

	Rules: **R6_ICSL**

	Issues:
	Redundancy: this rule is a duplication of REQ_012.
	Ambiguity: It is not clear if the SAI_CONNECT.request should be sent *immediately* as soon at initiator CSL is switched on, and *immediately* as soon as it returns to the NOCOMMS state, or if some delay is allowed (is there any allowed maximum delay?).
	Multiple

	Remarks:
	In the model the request is sent at the beginning of the next cycle.
	The CSL remains in state NOCOMMS until a safe connection has been established. While in state NOCOMMS, if a SAI_CONNECT.request has already been sent, it is not sent again until the connection timeout is expired.

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

	Rules: **R6_ICSL**, **R7_ICSL**, **R8_ICSL**
	Issues:
	Multiple

	Remarks: A connection is considered established when the CSL receives from SAI a SAI_CONNECT.indication.
	The "command for the establishment of the connection" corresponds to the sending of a SAI_CONNECT.request.

REQ_003	If configured as called, the CSL shall wait for report from underlying Layers that safe connection is established.

	Rules: **R7_CCSL**
	Issues: no issues have been found.

	Remarks: This requirement generalises what stated by REQ_014.
	With respect to REQ_014 this requirement is supposed to cover also the case a new connection confirmation when already in the COMMS state.
	A connection is considered established when the CSL receives from SAI a SAI_CONNECT.indication (REQ_024).
	What to do when SAI_CONNECT.indication is received is specified by REQ_008, REQ_009, and REQ_011.

REQ_004	The CSL shall discard any message either from User functions or from partner CSL before a confirmation of successful clock offset estimation (TTS option) or EC initialisation has been received from SAI sublayer.

	Rules: **R1_ICSL**, **R4_ICSL**, **R1_CCSL**, **R4_CCSL**
	Issues:
	Irrelevant: there is no need to refer to the specific SAI options for initialising the safe connection (i.e., TTS or EC).

	Remarks: The CSL shall discard any message either from User functions or from partner CSL when in state NOCOMMS.
	The completion of the safe connection initialisation, i.e., the establishing a safe connection is signalled by a SAI_CONNECT.indication (REQ_024).
	The loss of the safe connection (and return to the NOCOMMS state) is described by REQ_007 and REQ_015.

REQ_005	The CSL shall forward a received User message to RBC User functions only if all checks specified in supervision functions (REQ_006-REQ_011) are passed.
	--
	Rules: **R13_ICSL, R13_CCSL**
	--
	Issues:
	Ambiguity: there is no disambiguation between User messages received from partner RBC User or received from own RBC User.
	Irrelevant: Checks in supervision functions are about establishing or closing a connection, and this is a requirement about SAI to send such SAI_DATA indication only when all checks specified in supervision functions 10.2 are passed.
	Redundant: This requirement is redundant with REQ_017.
	--
	Remarks:
	When in state COMMS, the CSL forwards all the received User message to RBC User functions (when receiving from SAI the corresponding SAI_DATA indication).

REQ_006	Loss of safe connection shall be detected by the CSL reading reports from the underlying SFM (SAI_DISCONNECT.indication).
	--
	Rules: **R16_ICSL, R16_CCSL, R1_ICSL, R1_CCSL**
	--
	Issues:
	Inconsistency: Loss of safe connection shall also be detected without reading reports of underlying SFM, i.e., by timeout.
	--
	Remarks: The SAI_DISCONNECT.indication are considered the CSL only in state COMMS. In state NOCOMMS such reports are discarded.

REQ_007	If a report from underlying Layers is received that safe connection is lost, the CSL shall consider the communication in state NOCOMMS.
	Rules: **R16_ICSL, R16_CCSL**
	--
	Issues: no issues have been found
	--
	Remarks: The CSL when receiving a SAI_DISCONNECT.indication moves from COMMS to NOCOMMS.

	Further effects are related to the timers mentioned in REQ_008, REQ_009.

REQ_008	TTS option: after reception of report from SAI that the clock offset procedure has been completed, the CSL shall ensure that a message is sent to the partner RBC at the expiration of a configurable transmit time interval (reset at the sending of any message). If no User message needs to be sent, CSL is responsible to send a life sign message (see Figure 4);
	EC option: After reception of report from SAI that the EC initialisation procedure has been completed, the sending of messages is scheduled cyclically every (configurable) TC. If no request to send messages from User application is pending, a life sign is sent by CSL. If requests are pending, only one message per cycle is sent.

	Rules: **R8_ICSL**, **R10_ICSL**, **R11_ICSL**, **R12_ICSL**, **R16_ICSL**, **R17_ICSL**, **R8_CCSL**, **R9_CCSL**, **R10_CCSL**, **R11_CCSL**, **R12_CCSL**, **R16_CCSL**, **R17_CCSL**

	Issues:
	Irrelevant: The reference to SAI criteria (TTS or EC) user for initialising the connection is not relevant for the CSL.
	Ambiguity: For the case EC, it is not specified which of the pending messages is selected for being sent.
	Multiple

	Remarks:
	Life signs and user messages are sent only when CSL is in state COMMS.
	The send time interval is set and reset by CSL each time a new safe connection has been established, (i.e., a SAI_CONNECT.indication is received - REQ_024).
	After the CSL sends a life-sign message to SAI, it is responsibility of SAI to immediately forward it (TTS option) or to bufferise it and sending it at the appropriate EC cycle, but this is part of the requirements on the SAI behaviour.
	The send timer is reset and stopped when moving from COMMS to NOCOMMS.

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

	Rules: **R8_ICSL, R11_ICSL, R12_ICSL, R13_ICSL, R14_ICSL, R16_ICSL, R17_ICSL, R8_CCSL, R9_CCSL, R11_CCSL, R12_CCSL, R13_CCSL, R14_CCSL, R16_CCSL, R17_CCSL**

	Issues:
	Irrelevant: The reference to SAI criteria (TTS or EC) user for initialising the connection is not relevant for CSL.
	Multiple

	Remarks: The receive time interval is set and reset by CSL each time a new safe connection has been established, (i.e., a SAI_CONNECT.indication is received - REQ_024).
	Further effects of the expiration of the receive timer are stated in REQ_015.
	The receive timer is reset and stopped when moving from COMMS to NOCOMMS.

REQ_010	When communication is in state NOCOMMS, the CSL shall not accept/forward messages neither from its own RBC User functions nor from partner RBC; when switching to NOCOMMS, if the safe connection is still active, the CSL shall send a termination order (SAI_DISCONNECT.request).
	Note: when informed that the communication is in state NOCOMMS, the User functions will terminate all transactions.

	Rules: **R1_ICSL, R4_ICSL, R17_ICSL, R1_CCSL, R4_CCSL, R17_CCSL**

	Issues:
	Irrelevant: The note attached to REQ_010 does not specify any CSL behavior and is considered as an INFO.
	Redundant: The same behavior stated by this requirement is stated in REQ_015.
	Inconsistency: With respect to the wordings "if the safe connection is still active," the CSL does not need to have direct access to the current status of the SAI safe connection. The SAI_DISCONNECT.request must be sent only when moving to NOCOMM because of the expiration of the receive timer. In the other case, when CSL moves to NOCOMM because of DISCONNECT notification received from SAI, there is no need to send back to SAI the SAI_DISCONNECT.request.
	Multiple

	Remarks:
	Notice that, in principle, in the case of called CSL, at the SAI level the safe connection might have been already returned to active even if no indication has already arrived.

REQ_011	CSL can switch the communication from state NOCOMMs to state COMMS only when underlying Layers confirm the re-establishment of a safe connection. Note: communication in state COMMS is communicated to User functions, that will be able to restart management of transactions. --- Rules: **R8_ICSL**, **R8_CCSL** --- Issues: **Ambiguity**: The first part of the Note (notification to RBC_User) is covered REQ_019. **Irrelevant**: The second part of the node (management of transactions) is not related to the CSL behaviour. **Multiple** --- Remarks: The confirmation of the establishment of the safe connection occurs when a SAI_CONNECT.confirm (initiator case) or a SAI_CONNECT.indication (called case) is received. Further effects of this event are specified in REQ_008 and REQ_009, REQ_019.

REQ_012	If configured as initiator, at start-up, and when loss of safe connection is detected, the CSL shall send safe connection init order to SFM (SAI_CONNECT.request). --- Rules: **R6_ICSL** --- Issues: **Redundant**: The required behaviour is a duplication of what already required by REQ_001. --- Remarks: The "safe connection init order" of this rule (also called "command for the establishment of the connection" in REQ_001 and REQ_002) corresponds to the sending of a SAI_CONNECT.request.

REQ_013	If configured as initiator, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.confirm). --- Rules: **R6_ICSL**, **R8_ICSL**

	--- Issues: **Multiple** Remarks: What to do when a SAI_CONNECT.confirm is received is specified by REQ_011, REQ_008, REQ_009, REQ_011 and REQ_019.
REQ_014	If configured as called, at start-up, and when loss of safe connection is detected, the CSL shall wait for reception of safe connection established confirmation from SFM (SAI_CONNECT.indication). --- Rules: **R7_CCSL**, **R8_CCSL** --- Issues: **Redundant**: The required behaviour is a duplication of what already required by REQ_003. **Multiple** --- Remarks: This requirement repeats what said in REQ_003, in the particular cases of start-up and loss of safe connection. What to do when SAI_CONNECT.indication is received is specified by REQ_008, REQ_009, REQ_011 and REQ_019.
REQ_015	In case loss of communication is detected due to no valid messages received within a configurable time, the CSL shall send a safe connection termination order to SFM (SAI_DISCONNECT.request). --- Rules: **R17_ICSL**, **R17_CCSL** --- Issues: **Redundant**: The same behaviour is requested with more complex wording in REQ_010. --- Remarks: Further effects of this event are stated in REQ_009.
REQ_016	While the safe communication is active (state COMMS), the CSL is responsible of sending User messages received from RBC User functions to partner RBC.

	Rules: **R10_ICSL, R10_CCSL**

	Issues:
	Irrelevant: The forwarding occurs when the CSL is in state COMMS, with no relation with the current SAI safe connection state.

REQ_017	While the safe communication is active (state COMMS), the CSL is responsible of checking User messages received from partner RBC and forwarding (if checks are passed, see 7.2) to RBC User functions.

	Rules: **R13_ICSL** , **R13_CCSL**

	Issues:
	Redundant: This is a duplication of REQ_005.
	Irrelevant: The forwarding occurs when the CSL is in state COMMS, with no relation with the current SAI safe connection state.
	Multiple

REQ_018	The CSL is responsible of reading reports from SFM.

	Rules: **R2_ICSL, R3_ICSL, R8_ICS, R15_ICSL, R16_ICSL,** **R2_CCSL, R3_CCSL, R8_CCSL, R9_CCSL, R15_CCS, R16_CCSL**

	Issues:
	Redundant: The meaning and required effect of reading reports from SAI is defined by the other requirements.
	Incomplete: No requirements state the effect of reading SAI Error Reports.

	Remarks:
	In state NOCOMMS all SAI reports are discarded with the exception of SAI_CONNECT.indication (called CSL) or SAI_CONNECT_confirmation (calling CSL).

REQ_019	The CSL is responsible of sending reports to RBC User functions about state of communication (COMMS/NOCOMMS)

	Rules: **R8_ICSL, R16_ICSL, R17_ICSL, R8_CCSL, R9_CCSL, R16_CCSL, R17_CCSL**

	Issues:
	Redundant: This requirement partly overlaps with the note inside REQ_011.
	Inconsistency: Reports are not sent to the user when in state NOCOMMS.

	Remarks: Reports to RBC User functions about state of communication are sent each time CSL moves from COMMS to NOCOMMS and from NOCOMMS to COMMS.
	In the case of called CSL, report on new connection is sent even without state change (COMMS).

The following requirements do not mandate any additional CSL behaviour, but they map logical events to specific interface signals (e.g., SAI_DATA.request).
This mapping is partially provided by the previous requirements.
They should have the role of info and summary of the exchanged messages, notifications, orders, reports, between the system components.

REQ_020	CSL shall receive from User functions the messages to be forwarded to peer RBC User when in state COMMS.

	Rules: not applicable

	Issues:
	Redundant with REQ_016.
REQ_021	CSL shall forward to User functions the forwarding of messages received from communication partner.

	Rules: not applicable

	Issues:
	Redundant: with REQ_017.
REQ_022	CSL shall send to User functions the reports on loss of communication (missing life sign - state NOCOMMS).

	Rules: not applicable

	Issues:
	Redundant: this requirement is a sub-case of REQ_019.
REQ_023	CSL shall send to User functions the reports on state of safe connection state change (COMMS/NOCOMMS).

	Rules: not applicable

	Issues: **Redundant:** this requirement is a sub-case of REQ_019.
REQ_024	• SAI_CONNECT.request shall be used by initiator CSL to command the establishment of a safe connection • SAI_CONNECT.indication shall be used by called SAI to notify to the CSL the connection establishment request • SAI_CONNECT.response shall be used by called CSL to accept the connection request. • SAI_CONNECT.confirm shall be used by the initiator SAI entity to inform the CSL about the successful establishment of the safe connection. -- Rules: not applicable -- Issues **Redundant:** first bullet with REQ_012, second bullet with REQ_014, fourth bullet with REQ_013. **Irrelevant:** the third bullet is not useful for the modelling phase. **Multiple**
REQ_025	• SAI_DATA.request shall be used by CSL to transmit data to the peer entity. • SAI_DATA.indication shall be used to indicate to the CSL that data have been received successfully from the peer entity. -- Rules: **R10_ICSL,R12_ICSL,R13_ICSL, R10_CCSL,R12_CCSL, R13_CCSL,R14_CCSL** -- Issues **Multiple**
REQ_026	• SAI_DISCONNECT.request shall be used by the CSL to enforce a release of the safe connection. • SAI_DISCONNECT.indication shall be used to inform the CSL about a safe connection release. -- Rules: **R16_ICSL,R16_CCSL** -- Issues **Redundant:** first bullet with REQ_015. **Ambiguity:** other requirements mention loss of safe connection. **Multiple**
REQ_027	SAI Error Report shall be sent from SAI to CSL in case of errors detection by SAI (deletion, resequencing, delay, repetition). -- Rules: not applicable --

	Issues: **Irrelevant:** This requirement is related to SAI.

REVERSE MAPPING FROM MODEL RULES TO REQUIREMENTS

R1_ICSL	REQ_004, REQ_006, REQ_010,
R2_ICSL	REQ_018,
R3_ICSL	REQ_018,
R4_ICSL	REQ_004, REQ_010,
R6_ICSL	REQ_001, REQ_002, REQ_013,
R7_ICSL	REQ_002,
R8_ICSL	REQ_002, REQ_008, REQ_009, REQ_011, REQ_013, REQ_018, REQ_019,
R10_ICSL	REQ_008, REQ_016, REQ_025
R11_ICSL	REQ_008, REQ_009,
R12_ICSL	REQ_008, REQ_009, REQ_012, REQ_025
R13_ICSL	REQ_005, REQ_009, REQ_017, REQ_025
R14_ICSL	REQ_009, REQ_025
R15_ICSL	REQ_018,
R16_ICSL	REQ_006, REQ_007, REQ_008, REQ_009, REQ_018, REQ_019, REQ_026
R17_ICSL	REQ_008, REQ_009, REQ_010, REQ_015, REQ_019,

R1_CCSL	REQ_004, REQ_006, REQ_010,
R2_CCSL	REQ_018,
R3_CCSL	REQ_018,
R4_CCSL	REQ_004, REQ_010,
R7_CCSL	REQ_003, REQ_014,
R8_CCSL	REQ_008, REQ_009, REQ_014, REQ_018, REQ_019,
R9_CCSL	REQ_008, REQ_009, REQ_011, REQ_018, REQ_019,
R10_CCSL	REQ_008, REQ_016, REQ_025
R11_CCSL	REQ_008, REQ_009,
R12_CCSL	REQ_008, REQ_009, REQ_025
R13_CCSL	REQ_005, REQ_009, REQ_017, REQ_025
R14_CCSL	REQ_009, REQ_025
R15_CCSL	REQ_018,
R16_CCSL	REQ_006, REQ_007, REQ_008, REQ_009, REQ_018, REQ_019, REQ_026
R17_CCSL	REQ_008, REQ_009, REQ_010, REQ_015, REQ_019,

10 Appendix C - UMC encoding of I_CSL

```
-----------------------------------------------
Class I_CSL is
-----------------------------------------------
Signals
    SAI_CONNECT_confirm;
    SAI_DISCONNECT_indication;
    SAI_Error_report;
    SAI_DATA_indication(nrbc_msg);
    RBC_User_Data_request(nrbc_msg);
    --
    icsl_tick;
    -- SAI CONNECTIONS ARE AUTONUMOUS, not requsted by RBC_User

    --      OUTGOING signals
    -- SAI_CONNECT_request;
    -- SAI_DISCONNECT_request;
    -- SAI_DATA_request(nrbc_msg);
    -- RBC_User_Connect_indication;
    -- RBC_User_Disconnect_indication;
    -- RBC_User_Data_indication(nrbc_msg);

Vars
    RBC_User:obj;
    SAI:obj;
    receive_timer := 0;
    send_timer := 0;
    -------------
    max_receive_timer := 2; -- CONFIGURATION PARAM
    max_send_timer := 1;    -- CONFIGURATION PARAM
    max_connect_timer := 3; -- CONFIGURATION PARAM
    connect_timer := 3;     -- SAME VALUE of max_connect_timer
    connecting: bool := False;
    -------------

Behaviour

------------------------------------------------------------
-- when disconnetted ignore NRBC_MSGs and SAI_Notifications,
------------------------------------------------------------

R1_ICSL_discard_userdata:
    NOCOMMS -> NOCOMMS
        {RBC_User_Data_request(nrbc_msg) }

R2_ICSL_discard_disconnectindication:
    NOCOMMS -> NOCOMMS
        {SAI_DISCONNECT_indication }

R3_ICSL_discard_errorreport:
    NOCOMMS -> NOCOMMS
        {SAI_Error_report}

R4_ICSL_discard_dataindication:
    NOCOMMS -> NOCOMMS
        {SAI_DATA_indication(nrbc_msg)}


------------------------------------------------------------
-- establishing connections
------------------------------------------------------------

-- when disconnected issue new SAI connection request
--
```

```
R6_ICSL_Timer_okicsl_ISAI_connect request:
    NOCOMMS -> NOCOMMS
       { icsl_tick [connect_timer = max_connect_timer] /
           connect_timer := 0;
           _Timer.ok_icsl;
           SAI.SAI_CONNECT_request; }


-- when connecting handle clock cycles
--   and ignore all other events but connection confirmations
R7_ICSL_Timer_okicsl:
   NOCOMMS -> NOCOMMS
       {icsl_tick [connect_timer < max_connect_timer] /
           _Timer.ok_icsl;
           connect_timer := connect_timer +1}

-- when connecting handle connection confirmation
--
R8_ICSL_IRBC_rbcuser connect indication:
   NOCOMMS -> COMMS
       { SAI_CONNECT_confirm   /
           RBC_User.RBC_User_Connect_indication;
           connect_timer := max_connect_timer;
           receive_timer := 0;
           send_timer := 0; }

   --------------------------------
   -- handling active connections
   --------------------------------

-- when connected forward RBC_User NRBC_MSGs
--
R10_ICSL_ISAI_saidatarequest:
   COMMS -> COMMS
       { RBC_User_Data_request(nrbc_msg) /
           send_timer := 0;
           SAI.SAI_DATA_request(nrbc_msg); }


-- when connected, in no send or receive timers are expired, adjust timers
--
R11_ICSL_Timer_okicsl:
   COMMS -> COMMS
       { icsl_tick [(receive_timer < max_receive_timer)
               and (send_timer < max_send_timer)] /
           _Timer.ok_icsl;
           send_timer := send_timer +1;
           receive_timer := receive_timer+1; }

-- when connected, if the receive timer is not expired but send timer is expired,
-- send lifesign
R12_ICSL_Timer_okicsl_ISAI_saidatarequest:
   COMMS -> COMMS
       { icsl_tick [(receive_timer < max_receive_timer)
               and (send_timer = max_send_timer)] /
           _Timer.ok_icsl;
           send_timer := 0;
           receive_timer := receive_timer+1;
           SAI.SAI_DATA_request([LifeSign,0,0,0,0]) }


   -- when connected, is msg received from sai,forward to user and
   --   reset receive timer;
R13_ICSL_IRBC_rbcuserdataindication:
   COMMS -> COMMS
       {SAI_DATA_indication(nrbc_msg)
           [nrbc_msg[0]  /= LifeSign] /
           receive_timer := 0;
           RBC_User.RBC_User_Data_indication(nrbc_msg);
       }
```

```
-- when connected, if lifesign received from sai, reset receive timer;
--
R14_ICSL_handle_lifesign:
   COMMS -> COMMS
      {SAI_DATA_indication(nrbc_msg)
         [nrbc_msg[0] = LifeSign] /
         receive_timer := 0;
      }

-- when connected, ignore SAI_Error_report
--
R15_ICSL_IRBC_rbcusererrorindication:
   COMMS -> COMMS
      {SAI_Error_report }

-- when connected, disconnect and forward notification when notified by SAI
-- (and clear timers)
--
R16_ICSL_IRBC_rbcuserdisconnectindication:
   COMMS -> NOCOMMS
      {SAI_DISCONNECT_indication /
         RBC_User.RBC_User_Disconnect_indication;
         receive_timer := 0;
         send_timer := 0 }


-- when connected, if receive timer expires,disconnect, notify RBC_User and
--   request SAI termination
--
R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest:
   COMMS -> NOCOMMS
      { icsl_tick [receive_timer = max_receive_timer] /
         _Timer.ok_icsl;
         RBC_User.RBC_User_Disconnect_indication;
         SAI.SAI_DISCONNECT_request;
         receive_timer := 0;
         send_timer := 0; }

end I_CSL;
```

The automatically generated graphical layout for this component is shown in Figure 5.

11 Appendix D - ProB encoding of I_CSL

```
MACHINE HANDOVER

SETS

   CSL_SIGNALS =
    {SAI_CONNECT_confirm,      // only ISAI
     SAI_CONNECT_indication,   // only CSAI
     SAI_DISCONNECT_indication,
     SAI_Error_report,
     SAI_DATA_indication,      // +arg
     //
     RBC_User_Data_request,    // +arg
     icsl_tick,
     ccsl_tick};

   ...
   CSL_STATES =
    {NOCOMMS,
     COMMS};
```

```
...

CONSTANTS
  ...
  // I_CSL
  LifeSign,
  icsl_max_receive_timer,
  icsl_max_send_timer,
  icsl_max_connect_timer,

  ...

PROPERTIES
  LifeSign = 1 &
  ...
  // I_CSL
  icsl_max_receive_timer = 3 &
  icsl_max_send_timer = 1 &
  icsl_max_connect_timer = 4 &
  ...

VARIABLES
  // I_CSL
  icsl_buff,
  icsl_databuff,
  ICSL_STATUS,
  icsl_receive_timer,
  icsl_send_timer,
  icsl_connect_timer,
  ...

INVARIANT
  // I_CSL
  icsl_buff: seq(CSL_SIGNALS) &
  icsl_databuff: seq(0..5) &
  ICSL_STATUS: CSL_STATES &
  icsl_receive_timer: 0..10 &
  icsl_send_timer: 0..10 &
  icsl_connect_timer: 0..20 &
  ...

INITIALISATION
  // I_CSL
  icsl_buff := [];
  icsl_databuff := [];
  ICSL_STATUS := NOCOMMS;
  icsl_receive_timer := 0;
  icsl_send_timer := 0;
  icsl_connect_timer := icsl_max_connect_timer;
  ...

OPERATIONS

//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
//  ICSL
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

// ----------------------------------------------------------
// when disconnetted ignore NRBC_MSGs and SAI_Notifications,
// ----------------------------------------------------------

R1_ICSL_discard_userdata =
PRE
  ICSL_STATUS = NOCOMMS &
  icsl_buff /= [] &
  first(icsl_buff) = RBC_User_Data_request
THEN
  ICSL_STATUS := NOCOMMS;
```

```
   //
   icsl_buff := tail(icsl_buff);
   icsl_databuff := tail(icsl_databuff)
END;

R2_ICSL_discard_disconnectindication =
PRE
   ICSL_STATUS = NOCOMMS &
   icsl_buff /= [] &
   first(icsl_buff) = SAI_DISCONNECT_indication
THEN
   ICSL_STATUS := NOCOMMS;
   //
   icsl_buff := tail(icsl_buff)
END;

R3_ICSL_discard_errorreport =
PRE
   ICSL_STATUS = NOCOMMS &
   icsl_buff /= [] &
   first(icsl_buff) = SAI_Error_report
THEN
   ICSL_STATUS := NOCOMMS;
   //
   icsl_buff := tail(icsl_buff)
END;

R4_ICSL_discard_dataindication =
PRE
   ICSL_STATUS = NOCOMMS &
   icsl_buff /= [] &
   first(icsl_buff) = SAI_DATA_indication
THEN
   ICSL_STATUS := NOCOMMS;
   //
   icsl_buff := tail(icsl_buff);
   icsl_databuff := tail(icsl_databuff)
END;

// ----------------------------------------------------------
// establishing connections
// ----------------------------------------------------------

// when disconnected issue new SAI connection request
//
R6_ICSL_Timer_okicsl_ISAI_connectrequest =
PRE
   ICSL_STATUS = NOCOMMS &
   icsl_connect_timer = icsl_max_connect_timer &
   icsl_buff /= [] &
   first(icsl_buff) = icsl_tick
THEN
   ICSL_STATUS := NOCOMMS;
   //
   icsl_connect_timer := 0;
   timer_buff := timer_buff <- ok_icsl;
   isai_buff := isai_buff <- SAI_CONNECT_request;
   //
   icsl_buff := tail(icsl_buff)
END;

// when connecting handle clock cycles
//   and ignore all other events but connection confirmations
R7_ICSL_Timer_okicsl =
PRE
   ICSL_STATUS = NOCOMMS &
   icsl_buff /= [] &
   icsl_connect_timer < icsl_max_connect_timer &
   first(icsl_buff) = icsl_tick &
   icsl_connect_timer < icsl_max_connect_timer
THEN
```

```
    ICSL_STATUS := NOCOMMS;
    //
    timer_buff := timer_buff <- ok_icsl;
    icsl_connect_timer := icsl_connect_timer + 1;
    //
    icsl_buff := tail(icsl_buff)
END;


// when connecting handle connection confirmation
//
R8_ICSL_IRBC_rbcuserconnectindication =
PRE
    ICSL_STATUS = NOCOMMS &
    icsl_buff /= [] &
    first(icsl_buff) = SAI_CONNECT_confirm
THEN
    ICSL_STATUS :=COMMS;
    //
    icsl_connect_timer := icsl_max_connect_timer;
    icsl_receive_timer := 0;
    icsl_send_timer := 0;
    irbc_buff := irbc_buff <- RBC_User_Connect_indication;
    //
    icsl_buff := tail(icsl_buff)
END;


//-------------------------------
//-- handling active connections
//-------------------------------

// -- when connected forward User NRBC_MSGs
// --
R10_ICSL_ISAI_saidatarequest =
PRE
    ICSL_STATUS =COMMS &
    icsl_buff /= [] &
    first(icsl_buff) = RBC_User_Data_request
THEN
    ICSL_STATUS :=COMMS;
    //
    icsl_send_timer := 0;
    isai_buff := isai_buff <- SAI_DATA_request;
    isai_databuff := isai_databuff <- first(icsl_databuff);
    //
    icsl_buff := tail(icsl_buff);
    icsl_databuff := tail(icsl_databuff)
END;

// -- when connected, in no send or receive timers are expired, adjust timers
//
R11_ICSL_Timer_okicsl =
PRE
    ICSL_STATUS =COMMS &
    icsl_buff /= [] &
    first(icsl_buff) = icsl_tick &
    icsl_receive_timer < icsl_max_receive_timer &
    icsl_send_timer < icsl_max_send_timer
THEN
    ICSL_STATUS :=COMMS;
    //
    timer_buff := timer_buff <- ok_icsl;
    icsl_send_timer := icsl_send_timer + 1;
    icsl_receive_timer := icsl_receive_timer+1;
    //
    icsl_buff := tail(icsl_buff)
END;

// when connected, if the receive timer is not expired but send timer is expired,
//   send lifesign
```

R12_ICSL_Timer_okicsl_ISAI_saidatarequest =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = icsl_tick &
  icsl_receive_timer < icsl_max_receive_timer &
  icsl_send_timer = icsl_max_send_timer
THEN
  ICSL_STATUS :=COMMS;
  //
  timer_buff := timer_buff <- ok_icsl;
  icsl_send_timer := 0;
  icsl_receive_timer := icsl_receive_timer+1;
  isai_buff := isai_buff <- SAI_DATA_request;
  isai_databuff := isai_databuff <- LifeSign;
  //
  icsl_buff := tail(icsl_buff)
END;

// -- when connected, is msg received from sai,forward to user and
// -- reset receive timer;
//
R13_ICSL_IRBC_rbcuserdataindication =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = SAI_DATA_indication &
  first(icsl_databuff) /= LifeSign
THEN
  ICSL_STATUS :=COMMS;
  //
  icsl_receive_timer := 0;
  irbc_buff := irbc_buff <- RBC_User_Data_indication;
  irbc_databuff := irbc_databuff <- first(icsl_databuff);
  //
  icsl_buff := tail(icsl_buff);
  icsl_databuff := tail(icsl_databuff)
END;

R14_ICSL_handle_lifesign =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = SAI_DATA_indication &
  first(icsl_databuff) = LifeSign
THEN
  ICSL_STATUS :=COMMS;
  //
  icsl_receive_timer := 0;
  //
  icsl_buff := tail(icsl_buff);
  icsl_databuff := tail(icsl_databuff)
END;

// when connected, forward SAI msg notifications
//
R15_ICSL_IRBC_rbcusererrorindication =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = SAI_Error_report
THEN
  ICSL_STATUS :=COMMS;
  //
  icsl_buff := tail(icsl_buff)
END;

// when connected, disconnect and forward notification when notified by SAI
//

```
R16_ICSL_IRBC_rbcuserdisconnectindication =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = SAI_DISCONNECT_indication
THEN
  ICSL_STATUS := NOCOMMS;
  //
  irbc_buff := irbc_buff <- RBC_User_Disconnect_indication;
  icsl_receive_timer := 0;
  icsl_send_timer := 0;
  //
  icsl_buff := tail(icsl_buff)
END;

// when connected, if receive timer expires,disconnect, notify User and
// request SAI termination
//
R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest =
PRE
  ICSL_STATUS =COMMS &
  icsl_buff /= [] &
  first(icsl_buff) = icsl_tick &
  icsl_receive_timer = icsl_max_receive_timer
THEN
  ICSL_STATUS := NOCOMMS;
  //
  timer_buff := timer_buff <- ok_icsl;
  irbc_buff := irbc_buff <- RBC_User_Disconnect_indication;
  isai_buff := isai_buff <- SAI_DISCONNECT_request;
  icsl_receive_timer := 0;
  icsl_send_timer := 0;
  //
  icsl_buff := tail(icsl_buff)
END;

  ...
  ...

END
```

12 Appendix E - Sparx Model Report – CSL

CSL

Class in package 'Package1'

The Communication Supervision Layer class.

ELEMENTS OWNED BY CSL
📄 CSL : StateMachine

ATTRIBUTES
🔷 connect_timer : int Private
Accumulating time since last SAI_CONNECT_REQUEST.
[Is static True. Containment is Not Specified.]
🔷 initiator : boolean Public
The attribute initiator is used to distinguish the initiator SAI from the called SAI
[Is static True. Containment is Not Specified.]
🔷 max_connect_timer : int Public
This attribute represents the maximum amount of time that CSL waits after the last connection request issued to the SAI, before issuing a new one.
[Is static True. Containment is Not Specified.]
🔷 max_receive_timer : int Public
The maximum amount of time the CSL waits for a message from SAI before closing the connection.
[Is static True. Containment is Not Specified.]
🔷 max_send_timer : int Public
The maximum amount of time in which the CSL sends a message to underlying layer SAI. When this threshold is reached, the CSL will issue a LifeSign message.
[Is static True. Containment is Not Specified.]
🔷 receive_timer : int Private = 0
Accumulating time since last message received from SAI.
[Is static True. Containment is Not Specified.]
🔷 send_timer : int Private = 0

Accumulating time since last message.

[Is static True. Containment is Not Specified.]

ASSOCIATIONS

✏ Association (direction: Source -> Destination) CSL2RBC_USER

Each CSL has a reference to its own RBC_USER for sending signals

Source: Public (Class) CSL	Target: Public RBC_USER (Class) RBC_USER

✏ Association (direction: Source -> Destination) CSL2SAI

Each CSL has a reference to its own SAI for sending signals

Source: Public (Class) CSL	Target: Public SAI (Class) SAI

CSL

StateMachine owned by 'CSL', in package 'Package1'

The CSL state machine

ELEMENTS OWNED BY CSL

🗏 COMMS : State

🗏 NOCOMMS : State

🗏 Initial : Initial State

CSL diagram

StateMachine diagram in package 'Package1'

The CSL state machine
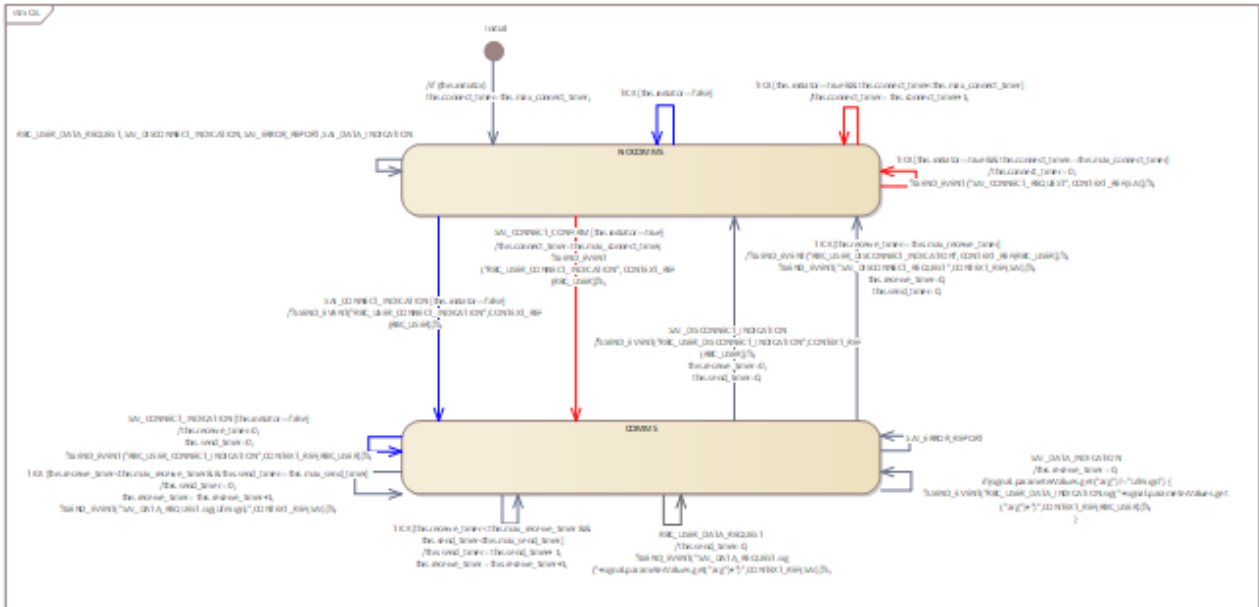
3. CSL

COMMS

State owned by 'CSL', in package 'Package1'

The CSL state machine COMMS state, where the connection is established

OUTGOING BEHAVIORAL RELATIONSHIPS

◀▪ Transition from COMMS to NOCOMMS
   Effect: %SEND_EVENT("RBC_USER_DISCONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;
   this.receive_timer=0;
   this.send_timer=0;

   Triggers:
     SAI_DISCONNECT_INDICATION Signal

Mapping to UMC rules:
R16_ICSL_IRBC_rbcuserdisconnectindication
R16_CCSL_IRBC_rbcuserdisconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◀▪ Transition from COMMS to COMMS
   Effect: this.send_timer=0;
   %SEND_EVENT("SAI_DATA_REQUEST.sig("+signal.parameterValues.get("arg")+")",CONTEXT_REF(SAI))%;

Triggers:
   RBC_USER_DATA_REQUEST Signal sig

Mapping to UMC rules:
R10_ICSL_ISAI_saidatarequest
R10_CCSL_CSAI_saidatarequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◄▪ Transition from COMMS to COMMS
   Effect: this.receive_timer = 0;
   if (signal.parameterValues.get("arg") !="LifeSign") {
      %SEND_EVENT("RBC_USER_DATA_INDICATION.sig("+signal.parameterValues.get("arg")+")",CONTEXT_REF(RBC_USER))%;
   }

   Triggers:
     SAI_DATA_INDICATION Signal sig

Mapping to UMC rules:
R13_ICSL_IRBC_rbcuserdataindication
R14_ICSL_handle_lifesign
R13_CCSL_CRBC_rbcuserdataindication
R14_CCSL_handle_lifesign

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◄▪ Transition from COMMS to COMMS
   Guard: this.receive_timer<this.max_receive_timer && this.send_timer==this.max_send_timer
   Effect: this.send_timer = 0;
   this.receive_timer = this.receive_timer+1;

   %SEND_EVENT("SAI_DATA_REQUEST.sig(LifeSign)",CONTEXT_REF(SAI))%;

   Triggers:
     TICK Signal

Mapping to UMC rules:
R12_ICSL_Timer_okicsl_ISAI_saidatarequest
R12_CCSL_Timer_okccsl_CSAI_saidatarequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◄▪ Transition from COMMS to NOCOMMS
   Guard: this.receive_timer==this.max_receive_timer
   Effect: %SEND_EVENT("RBC_USER_DISCONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;
   %SEND_EVENT("SAI_DISCONNECT_REQUEST",CONTEXT_REF(SAI))%;
   this.receive_timer=0;
   this.send_timer=0;

   Triggers:
     TICK Signal

Mapping to UMC rules:

R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest
R17_CCSL_Timer_okccsl_CRBC_rbcuserdisconnectindication_CSAI_saidisconnectrequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◀▪ Transition from COMMS to COMMS
    Guard: this.initiator==false
    Effect: this.receive_timer=0;
    this.send_timer=0;
    %SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

    Triggers:
      SAI_CONNECT_INDICATION Signal

Mapping to UMC rules:
R9_CCSL_CRBC_rbcuserconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◀▪ Transition from COMMS to COMMS

    Triggers:
      SAI_ERROR_REPORT Signal

Mapping to UMC rules:
R15_ICSL_IRBC_rbcusererrorindication
R15_CCSL_CRBC_rbcusererrorindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

◀▪ Transition from COMMS to COMMS
    Guard: this.receive_timer < this.max_receive_timer && this.send_timer<this.max_send_timer
    Effect: this.send_timer = this.send_timer+ 1;
    this.receive_timer = this.receive_timer+1;

    Triggers:
      TICK Signal

Mapping to UMC rules:
R11_ICSL_Timer_okicsl
R11_CCSL_Timer_okccsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

INCOMING BEHAVIORAL RELATIONSHIPS

⇥ Transition from COMMS to COMMS
    Effect: this.send_timer=0;
    %SEND_EVENT("SAI_DATA_REQUEST.sig("+signal.parameterValues.get("arg")+")",CONTEXT_REF(SAI))%;

Triggers:
    RBC_USER_DATA_REQUEST Signal sig

Mapping to UMC rules:
R10_ICSL_ISAI_saidatarequest
R10_CCSL_CSAI_saidatarequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from COMMS to COMMS
    Effect: this.receive_timer = 0;
    if (signal.parameterValues.get("arg") !="LifeSign") {
        %SEND_EVENT("RBC_USER_DATA_INDICATION.sig("+signal.parameterValues.get("arg")+")",CONTEXT_REF(RBC_USER))%;
    }

    Triggers:
        SAI_DATA_INDICATION Signal sig

Mapping to UMC rules:
R13_ICSL_IRBC_rbcuserdataindication
R14_ICSL_handle_lifesign
R13_CCSL_CRBC_rbcuserdataindication
R14_CCSL_handle_lifesign

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from COMMS to COMMS
    Guard: this.receive_timer<this.max_receive_timer && this.send_timer==this.max_send_timer
    Effect: this.send_timer = 0;
    this.receive_timer = this.receive_timer+1;

    %SEND_EVENT("SAI_DATA_REQUEST.sig(LifeSign)",CONTEXT_REF(SAI))%;

    Triggers:
        TICK Signal

Mapping to UMC rules:
R12_ICSL_Timer_okicsl_ISAI_saidatarequest
R12_CCSL_Timer_okccsl_CSAI_saidatarequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from COMMS to COMMS
    Guard: this.initiator==false
    Effect: this.receive_timer=0;
    this.send_timer=0;
    %SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

    Triggers:
        SAI_CONNECT_INDICATION Signal

Mapping to UMC rules:
R9_CCSL_CRBC_rbcuserconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from NOCOMMS to COMMS
   Guard: this.initiator==true
   Effect: this.connect_timer=this.max_connect_timer;

   %SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

   Triggers:
    SAI_CONNECT_CONFIRM Signal

Mapping to UMC rules:
R8_ICSL_IRBC_rbcuserconnectindication
Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from NOCOMMS to COMMS
   Guard: this.initiator==false
   Effect: %SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

   Triggers:
    SAI_CONNECT_INDICATION Signal

Mapping to UMC rules:
R8_CCSL_CRBC_rbcuserconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from COMMS to COMMS

   Triggers:
    SAI_ERROR_REPORT Signal

Mapping to UMC rules:
R15_ICSL_IRBC_rbcusererrorindication
R15_CCSL_CRBC_rbcusererrorindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from COMMS to COMMS
   Guard: this.receive_timer < this.max_receive_timer && this.send_timer<this.max_send_timer
   Effect: this.send_timer = this.send_timer+ 1;
this.receive_timer = this.receive_timer+1;

   Triggers:
    TICK Signal

Mapping to UMC rules:
R11_ICSL_Timer_okicsl
R11_CCSL_Timer_okccsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

NOCOMMS

State owned by 'CSL', in package 'Package1'

The CSL state machine NOCOMMS state, where the connection is not established

OUTGOING BEHAVIORAL RELATIONSHIPS

Transition from NOCOMMS to COMMS
Guard: this.initiator==true
Effect: this.connect_timer=this.max_connect_timer;

%SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

Triggers:
SAI_CONNECT_CONFIRM Signal

Mapping to UMC rules:
R8_ICSL_IRBC_rbcuserconnectindication
Mapping to Requirements:
(see mapping from UMC rules to requirements)

Transition from NOCOMMS to COMMS
Guard: this.initiator==false
Effect: %SEND_EVENT("RBC_USER_CONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;

Triggers:
SAI_CONNECT_INDICATION Signal

Mapping to UMC rules:
R8_CCSL_CRBC_rbcuserconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

Transition from NOCOMMS to NOCOMMS
Guard: this.initiator==false

Triggers:
TICK Signal

Mapping to UMC rules:
R7_CCSL_Timer_okccsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

Transition from NOCOMMS to NOCOMMS

Guard: this.initiator==true && this.connect_timer<this.max_connect_timer
Effect: this.connect_timer = this.connect_timer+1;

Triggers:
  TICK Signal

Mapping to UMC rules:
R7_ICSL_Timer_okicsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

⬅ Transition from NOCOMMS to NOCOMMS
  Guard: this.initiator==true && this.connect_timer==this.max_connect_timer
  Effect: this.connect_timer = 0;

  %SEND_EVENT("SAI_CONNECT_REQUEST",CONTEXT_REF(SAI))%;

  Triggers:
    TICK Signal

Mapping to UMC rules:
R6_ICSL_Timer_okicsl_ISAI_connectrequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

⬅ Transition from NOCOMMS to NOCOMMS

  Triggers:
    RBC_USER_DATA_REQUEST Signal sig
    SAI_DISCONNECT_INDICATION Signal
    SAI_ERROR_REPORT Signal
    SAI_DATA_INDICATION Signal sig

Mapping to UMC rules:
R1_ICSL_discard_userdata,
R2_ICSL_discard_dsconnectindication,
R3_ICSL_discard_errorreport,
R4_ICSL_discard_dataindication,
R1_CCSL_discard_userdata,
R2_CCSL_discard_dsconnectindication,
R3_CCSL_discard_errorreport,
R4_CCSL_discard_dataindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

INCOMING BEHAVIORAL RELATIONSHIPS

➡ Transition from COMMS to NOCOMMS
  Effect: %SEND_EVENT("RBC_USER_DISCONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;
  this.receive_timer=0;
  this.send_timer=0;

Triggers:
    SAI_DISCONNECT_INDICATION Signal

Mapping to UMC rules:
R16_ICSL_IRBC_rbcuserdisconnectindication
R16_CCSL_IRBC_rbcuserdisconnectindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

⇒ Transition from COMMS to NOCOMMS
    Guard: this.receive_timer==this.max_receive_timer
    Effect: %SEND_EVENT("RBC_USER_DISCONNECT_INDICATION",CONTEXT_REF(RBC_USER))%;
    %SEND_EVENT("SAI_DISCONNECT_REQUEST",CONTEXT_REF(SAI))%;
    this.receive_timer=0;
    this.send_timer=0;

    Triggers:
       TICK Signal

Mapping to UMC rules:
R17_ICSL_Timer_okicsl_IRBC_rbcuserdisconnectindication_ISAI_saidisconnectrequest
R17_CCSL_Timer_okccsl_CRBC_rbcuserdisconnectindication_CSAI_saidisconnectrequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

⇒ Transition from Initial to NOCOMMS
    Effect: if (this.initiator) this.connect_timer=this.max_connect_timer;

Initially the connect_timer is reset to max_connect_timer for the initiator CSL, such that the request for connection is immediate

---

⇒ Transition from NOCOMMS to NOCOMMS
    Guard: this.initiator==false

    Triggers:
       TICK Signal

Mapping to UMC rules:
R7_CCSL_Timer_okccsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

⇒ Transition from NOCOMMS to NOCOMMS
    Guard: this.initiator==true && this.connect_timer<this.max_connect_timer
    Effect: this.connect_timer = this.connect_timer+1;

    Triggers:
       TICK Signal

Mapping to UMC rules:
R7_ICSL_Timer_okicsl

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from NOCOMMS to NOCOMMS
    Guard: this.initiator==true && this.connect_timer==this.max_connect_timer
    Effect: this.connect_timer = 0;

    %SEND_EVENT("SAI_CONNECT_REQUEST",CONTEXT_REF(SAI))%;

    Triggers:
        TICK Signal

Mapping to UMC rules:
R6_ICSL_Timer_okicsl_ISAI_connectrequest

Mapping to Requirements:
(see mapping from UMC rules to requirements)

➡ Transition from NOCOMMS to NOCOMMS

    Triggers:
        RBC_USER_DATA_REQUEST Signal sig
        SAI_DISCONNECT_INDICATION Signal
        SAI_ERROR_REPORT Signal
        SAI_DATA_INDICATION Signal sig

Mapping to UMC rules:
R1_ICSL_discard_userdata,
R2_ICSL_discard_dsconnectindication,
R3_ICSL_discard_errorreport,
R4_ICSL_discard_dataindication,
R1_CCSL_discard_userdata,
R2_CCSL_discard_dsconnectindication,
R3_CCSL_discard_errorreport,
R4_CCSL_discard_dataindication

Mapping to Requirements:
(see mapping from UMC rules to requirements)

Initial
Initial State owned by 'CSL', in package 'Package1'

The CSL state machine initial state

OUTGOING BEHAVIORAL RELATIONSHIPS

⬅ Transition from Initial to NOCOMMS
    Effect: if (this.initiator) this.connect_timer=this.max_connect_timer;

Initially the connect_timer is reset to max_connect_timer for the initiator CSL, such that the request for connection is immediate