Formal Modeling and Initial Analysis of the 4SECURail Case Study

Franco Mazzanti
ISTI-CNR
Pisa, Italy
franco.mazzanti@isti.cnr.it

Dimitri Belli
ISTI-CNR
Pisa, Italy
dimitri.belli@isti.cnr.it

We present the case study developed in the context of the 4SECURail project and the approach used for its formal modeling and analysis. Starting from a simple SysML/UML behavioral model of the system requirements, three formal models have been developed using three different frameworks, namely UMC, ProB, and CADP/LNT. The paper shows how the different ways to represent and analyze the system from the three different points of view allow us to take advantage of the resulting diversity.

1 Introduction

One of the goals of the 4SECURail project[1] is to observe the possible approaches, benefits, limits, and costs of introducing formal methods inside the *requirements definition* process in the context of railway-signaling systems. This has been done with the set up of a "demonstrator" with the purpose to exemplify the application of state-of-the-art tools and methodologies to a selected railway case study with the collection of meaningful information on the costs and benefits of the process. The overall context and objectives of this project and experimentation are described in [16, 20]; in this paper, we describe specifically the approach that has been followed for the formal modeling and initial analysis of the case study, which has seen the exploitation of three different formal verification frameworks. The rest of the paper is structured as follows: In Section 2, we provide details about the case study that has been the object of the experimentation; in Section 3, we present the formal modeling approach that has been adopted in the demonstrator process; in Section 4, we describe the various kinds of analysis performed. In Sections 5 and 6, we respectively hint at some related works and draw our conclusions.

2 The reference case study

The transit of a train from an area supervised by a Radio Block Centre (RBC) to an adjacent area supervised by another RBC occurs during the so-called RBC-RBC handover phase and requires the exchange of information between the two RBCs according to a specific protocol. This exchange of information is supported by the communication layer specified within the UNISIG SUBSET-039 [30], UNISIG SUBSET-098 [28], and UNISIG SUBSET-037 [29], and the whole stack is implemented by both sides of the communication channel. Figure 1 summarizes the overall structure of the UNISIG standards, supporting the handover of a train. The 4SECURail case study is based on two main sub-components of the communication layers constituting the RBC-RBC handover. The considered components are the Communication Supervision Layer (CSL) of the SUBSET-039 and the Safe Application Intermediate

[1] https://4SECURail.eu November 2019 – November 2021.

Clemens Dubslaff and Bas Luttik (Eds.):
Models for Formal Analysis of Real Systems (MARS 2022)
EPTCS 355, 2022, pp. 118–144, doi:10.4204/EPTCS.355.6

© F. Mazzanti & D. Belli
This work is licensed under the
Creative Commons Attribution License.

SubLayer (SAI) of the SUBSET-098. These two components are the main actors that support the creation/deletion of safe communication lines and protect the transmission of messages exchanged on such lines. In particular, the CSL is responsible for requesting the activation – and in case of failure, the re-establishment – of the communication line, for continuously controlling its liveliness, and for the forwarding of the handover transaction messages. The SAI is responsible for ensuring the absence of excessive delays, repetitions, losses, or re-ordering of messages during their transmissions. This is achieved by adding sequence numbers and time-related information to the RBC messages. The RBC/RBC communication line consists of two sides that are properly configured as "initiator" and "called". With respect to

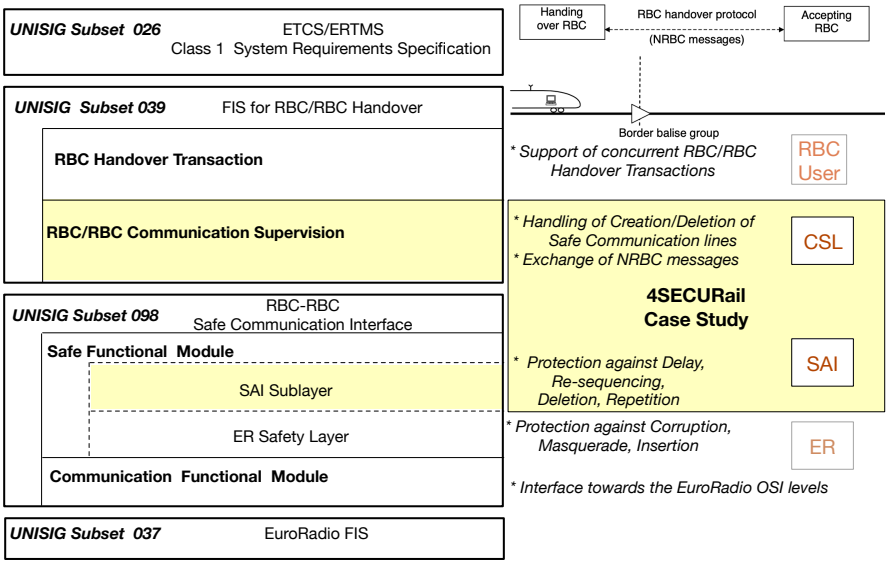

Figure 1: Overall structure of the 4SECURail case study

the SUBSET-098, the 4SECURail case study neither includes the EuroRadio Safety Layer (ER), which is responsible for preventing corruption, masquerading and insertion issues during the communications, nor the lower Communication Functional Module (CFM) interface. With respect to the SUBSET-039, the 4SECURail case study does not include the description of the activation of multiple, concurrent RBC-RBC handover transactions when trains move from a zone supervised by an RBC to an adjacent zone supervised by another RBC. From the point of view of the CSL, the RBC messages are forwarded to/from the other RBC side without the knowledge of their specific contents or session to which they belong. The case study of the project, as derived from the above-mentioned standards, is described in natural language in Deliverable D2.3 [23], along with the rationale for its choice. Of course, the level of abstraction of these requirement documents is not that one of an executable system specification, but a higher level.

3 The formal modeling

3.1 From natural language to executable UML specifications

As shown in Figure 2, the first step towards the generation of formal models of the system is the description – in terms of extremely simple SysML/UML features – of the system components described by the

natural language requirements. It is well known that requirements described in free-style natural language suffer the risk of being unclear (e.g., redundant), potentially ambiguous, in part contradictory, and possibly not describing essential aspects. Moreover, since the railway infrastructure is essentially a system of systems, specifying and guaranteeing the desired interoperability among the various components is a more challanging task than specifying and guaranteeing the independent safety of each singularly specified component.

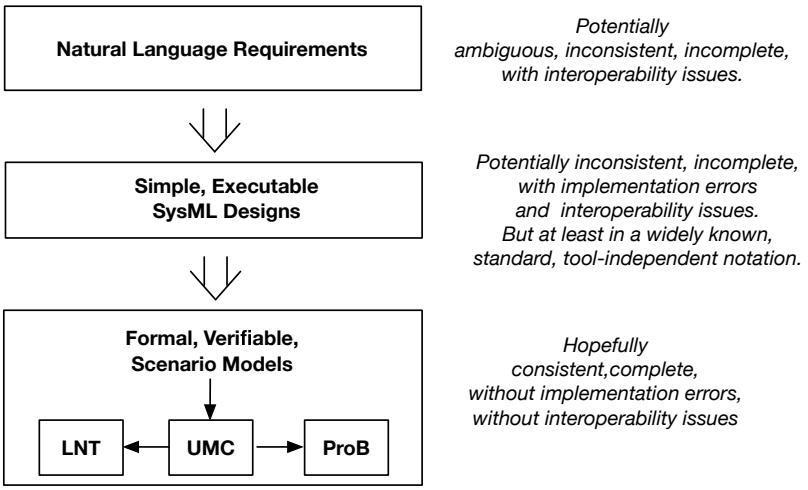

Figure 2: From natural language to formal models

Constructing a possible implementation using a potentially executable notation *with clear semantics* allows (at least) to remove the underlying ambiguities but, until the system is thoroughly tested or verified, the risk of logical deficiencies persists. However, beyond the beneficial "natural language interpretation" step, the "executable implementation" step risks being a critical source of mistakes. Therefore, the subsequent formal modeling and analysis step of the "executable implementation" becomes essential. The association of the term "clear semantics" with the term "UML" can be, in general, quite problematic. In our case, we have used the very minimal set of UML features needed for our executable modeling, avoiding all the complexities related, for example, to composite states, transition priorities, deferred events, and making the explicit assumption of FIFO event queues. The extreme simplicity of the resulting subset aims not only to the association of a "precise semantics" and a *simple intituive meaning* to the designs but also to an "easy translation" of the designs into several formal notations. Appendix A shows our reference UML state-machine diagrams for the CSL and SAI system components of the case study, in both their *initiator-side* and *called-side* version.

3.2 From executable UML specifications to verifiable scenarios

The system requirements in the Deliverable D2.3 [23] have been the base for the design of the executable models of the CSL and SAI components. However, in order to have an actually verifiable system, we need a *closed* system that contains the specified components plus all the needed environment components that stimulate, receive data, and forward messages from the initiator to called side of the system. In order to deal with the time-related aspects of the specification, we also introduce a timer component that allows

all the other components to proceed in parallel in an asynchronous way but relatively at the same speed[2]. Figure 3 shows the resulting structure of the whole system. Also all the added environment and timer components can be designed in UML to facilitate the system encoding into the selected formal notations. An example of these environment components is contained in Appendix A.

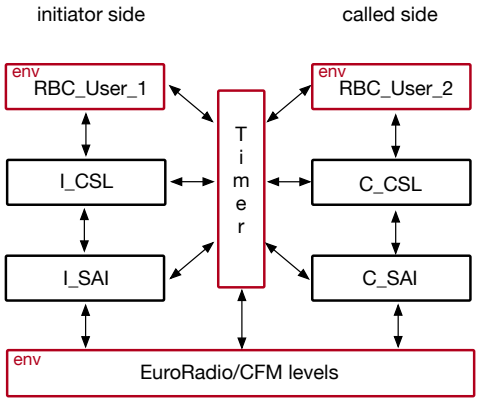

Figure 3: The complete executable system structure

It is infeasible/impractical to define these environment components in their full possible generality because the system components are heavily dependent on several configuration parameters. It makes more sense to define them according to the properties we intend to verify on the complete closed system. As first step of formal modeling, the executable UML system diagrams corresponding to a given scenario are translated into the notation accepted by the UMC[3] tool. At the beginning of the project, the possibility of designing the SysML system using a commercial MBSE framework – namely SPARX-EA[4] – has been evaluated. This approach has been abandoned because of time and effort constraints of the project. Implementing a translator from the SPARX-generated XMI towards UMC would have been a significant effort. Moreover, it would have tied the whole analysis approach to a specific commercial tool, a fact which was not considered desirable. Therefore our initial SysML models have the structure of simple graphical designs; their role is just that one of constituting an intermediate, easy-to-understand documentation halfway between the natural language requirements and the formal models[5]. Starting from the UMC notation, further formal models have been automatically generated in the notations accepted by the ProB[6] and CADP/LNT[7] tools. UMC [3, 4, 12] has been chosen as the target of the initial formal encoding because it is a tool natively oriented to fast-prototyping of SysML systems. It supports a textual notation of UML state-machine diagrams that directly reflects the graphical counterpart, allows fast state-space exploration, state- and event-based (on-the-fly) model checking, and detailed debugging of the system. Last but not least, it is part of a framework developed locally at ISTI. We have a deep insider knowledge that allowed us to easily implement translators towards the other formal notations within the time and effort constraint of the project. However, UMC is essentially a teaching/research-oriented aca-

[2]Since all the system components are modeled as executing a cyclic activity, the timer component just constrains the frequency of the cycles to be the same while allowing the overlapping of their behavior.

[3]https://fmt.isti.cnr.it/umc

[4]https://sparxsystems.com/products/ea/index.html

[5]more details can be found in [20]

[6]https://prob.hhu.de/

[7]https://cadp.inria.fr/

demic tool and lacks the maturity, stability, and support level required by an industry-usable framework. Also for this reason we have planned inside the project the exploitation of further, more industry-ready formal frameworks. ProB [15] has been selected as the second target of the formal encoding because of its recognized role (see e.g. [6, 7]) in the field of formal railway-related modeling. Is it supported by (more than one) very user-friendly GUI. It allows LTL/CTL model checking, state-space exploration, state-space projections, and trace descriptions in the form of sequence diagrams. Last but not least, it is a framework with which we have already had some previous modeling experience [8], and that did not require a learning-from-scratch step. CADP/LNT [10, 11] has been selected as the third target of the formal encoding because of its theoretical roots on LTS-related theories. These allow to reason in terms of minimizations, bisimulations, and compositional verifications. CADP is a rich toolbox that supports a wide set of μ-calculus-based branching-time logic and a powerful scripting language (SVL [9]) to support verification. Also in this case its choice has been influenced by the previous experiences we have had with this framework [13, 14]. There are several ways in which SysML/UML designs might be encoded into the ProB and LNT formal notations. In our case, we made the choice to generate both ProB and LNT models *automatically* from the UMC model. The translation implemented in our demonstrator is still a preliminary version and does not exploit at best all the features potentially offered by the target framework [8]. Nevertheless, the availability of the automatic translation proved to be an essential aspect of the demonstrated approach. Our models and scenarios have been developed incrementally, with a long sequence of refinements and extensions. At every single step, we have been able to quickly perform the lightweight formal verification of interest with almost no effort. This would not have been possible without an automatic generation of the ProB and LNT models. In the following, we will give some details on the overall structure of the generated models, referring to D2.5 [17] for a broader presentation. All the UMC/ProB/LNT models, specifying the scenarios of interest, are available from an open access repository [18], as well as the source code of the applied translators [19].

3.2.1 UMC encoding

In UMC, a system definition is specified as a set of active objects that are instances of class definitions. A class declaration specifies a template of state-machine, defining the set of events accepted by the machine, its local variables, and the state-machine behavior when state transitions are triggered. State machine transitions are encoded in a simple textual form and specify, as shown in Figure 4:

- an optional transition label (R9_ICSL_userdataind),
- the source and target states of the transition (COMMS, COMMS),
- a block {...} containing: the triggering event of the transition (ISAI_DATA_indication), possibly with parameters and guards, and the sequence of actions to be performed as an effect of the transition (the sending of the IRBC_User_Data_indication signal to the RBC_User component and the assignment to the receiveTimer variable).

Appendix B shows the UMC encoding of the component I_CSL whose UML state-machine diagram is shown in Appendix A. The mapping of the UML diagrams to the UMC encoding is almost direct. There are only a few aspects that deserve some attention. One point is that UMC transitions are "atomic" also at the system level, while the UML transitions are "atomic" only with respect to the state-machine to which they belong. Therefore, if we have a UML transition that sends several signals to other objects, a correct modeling of the behavior requires splitting the UML transition into several atomic steps. An

[8] E.g. all message parameters are mapped into integer values without considering the specific subrange to which that might belong.

example of this is shown in Figure 5, where a UML transition sending three signals is split into a sequence of three UMC transitions. The second point is that in UML, when a dispatched event does not trigger any transition it is simply removed from the event queue and discarded. This behavior is implicit in the state-machine diagram, but it is reasonable to make it explicit in the UMC designs to simplify the translation of the models into the other notations. This also allows distinguishing more clearly the case in which an event is intentionally (correctly) discarded from the cases in which the arrival of the event is simply a not relevant situation or the case in which it is a really unintended behavior highlighting a case of system malfunctioning.

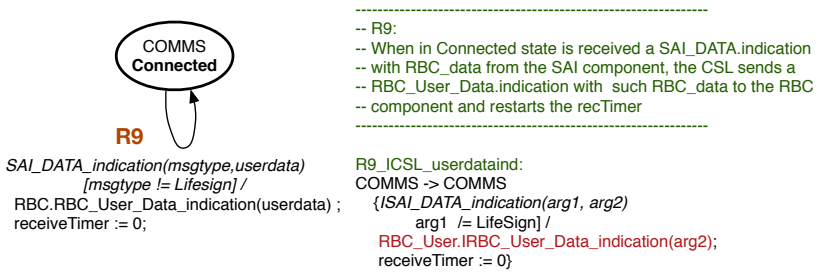

Figure 4: Textual encoding of a state-machine transition

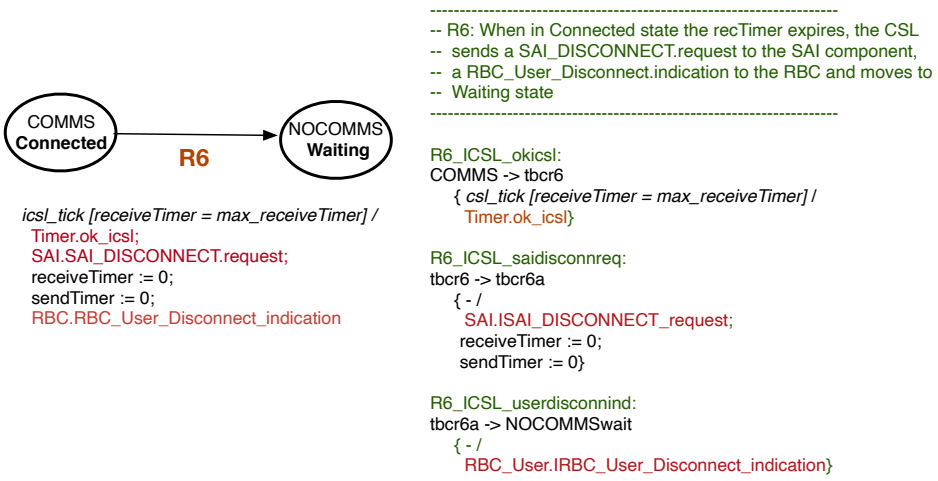

Figure 5: Splitting a UML transition into a sequence of UMC atomic transitions

3.2.2 ProB encoding

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

- The class attributes of UML state-machines must be merged into a single B state-machine definition. This may require the prefixing of the variable names with the component names to avoid

name clashes. The same manipulation has to be done for the operation names (transition labels in UMC) and the other entities that may require duplication.

- The currently active state of a UML state-machine is represented in B by the current value of an ad-hoc variable *statemachine_STATE*. There is one such variable for each UML state-machine.
- Within the B machine structure, all types, constants, and variable definitions and initializations must appear at the beginning of the machine definition. This disrupts the original structure of the system, forcing us to spread the UML state-machine definition into several places in the B machine specification.
- In UML state-machines, the event pool (a buffer implementing asynchronous communications that contains at each moment the set of signals arrived in a state-machine but not yet dispatched or discarded) is part of the engine support and thus is not explicitly modelled. In B these event-pool components must be explicitly modelled. This is because, contrary to UMC, B is not a tool natively designed for handling UML state-machines. Therefore "buffer" variables representing the state-machine event pool are added to the B model. Consequently, the action of sending a signal to another state-machine will be modelled with the insertion of a value to the corresponding variable buffer, and the dispatching of a signal to trigger a transition will be modelled with the extraction of the first element of such a buffer.
- Each transition rule definition of the UMC state-machine design is mapped onto an equivalent operation of the B machine.

Figure 6 shows, as an example, the ProB encoding of the UML transition R4 of the initiator CSL component, while the full code of the state-machine is shown in Appendix C.

```
Class I_CSL
  ...
  ...
Behaviour
  ...
  ...
R4_ICSL_userconnind :
NOCOMMSconnecting -> COMMS
  { ISAI_CONNECT_confirm  /
      RBC_User.RBC_User_Connect_indication;
      connectTimer := max_connectTimer;
      receiveTimer := 0;
      sendTimer := max_sendTimer;
  }
  ...
  ...
end I_CSL;
```

```
MACHINE SYS
  ...
  ...
OPERATIONS
  ...
  ...
R4_ICSL_userconnind =
PRE
    ICSL_buff /= [] &
    first(ICSL_buff) = ISAI_CONNECT_confirm &
    ICSL_STATE = ICSL_NOCOMMSconnecting
THEN
    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);
    ICSL_STATE = ICSL_COMMS
END;
  ...
  ...
END;
```

Figure 6: Textual encoding of a UMC (left) and ProB (right) state-machine transition

3.2.3 CADP/LNT encoding

LNT is one of the formal notations accepted by the CADP verification framework. The notation is a simplified variant of E-LOTOS [27], of which it preserves the expressiveness but adopts a more user-friendly and regular notations borrowed from imperative and functional programming languages. A system is described in LNT as a parallel composition of (parametric) processes, which synchronize upon a statically defined set of events. A process can have private variables that can be manipulated with

classical imperative statements. The global environment is constituted by the data types and functions used by the processes. An LNT specification is internally translated into the LOTOS [26] algebraic notation and can be analyzed using the CADP toolbox.

In this case, each UMC state-machine is associated with an independent LNT process. All the processes do not share any memory and interact through synchronous actions in the typical style of process algebras. Each process handles a local event pool modelled as a FIFO buffer and is *always* enabled to accept synchronizations from other processes willing to push a new event in the queue. Beyond accepting incoming messages, the LNT process can internally evolve, performing internal steps that transform the local status or synchronizing with other processes by sending messages towards other state-machines. The final system is obtained by composing in parallel all the processes which synchronize the corresponding actions of sending and receiving a message. Figure 7 shows the overall structure of a state-machine process corresponding to the initiator CSL, while the full code of the process is shown in Appendix D.

```
Class I_CSL
  ...
  ...
Behaviour
  ...
  ...
R4_ICSL_userconnind :
NOCOMMSconnecting -> COMMS
  { ISAI_CONNECT_confirm  /
      RBC_User.RBC_User_Connect_indication;
      connectTimer := max_connectTimer;
      receiveTimer := 0;
      sendTimer := max_sendTimer;
  }
  ...
  ...
end I_CSL;
```

```
process ICSL [..] is
  ...
  var mybuff: ICSL_BUFF,
     ...
     ...
  in
    loop
      select
       ----------------------------------------
       --- buffering incoming signals
       ----------------------------------------
        var arg1: Int in
           IRBC_User_Data_request(?arg1);
           mybuff := append(IRBC_User_Data_request, mybuff);
           mynatbuff := append(arg1, mynatbuff)
         end var
      []
         ISAI_CONNECT_confirm;
         mybuff := append(ISAI_CONNECT_confirm, mybuff)
      []
         ...
         ...
       ----------------------------------------
       --- triggering statemachine transitions from events pool
       ----------------------------------------
      []
       -- R4_ICSL_userconnind
       only if
         mybuff /= nil and
         head(mybuff) = ISAI_CONNECT_confirm and
         STATE = NOCOMMSconnecting
       then
         RBC_User_Connect_indication;
         connect_timer := ICSL_max_connect_timer;
         receive_timer := 0;
         send_timer := 0;
         mybuff := tail(mybuff);
         STATE = COMMS
       end if
      []
        ...
        ...
      end select
    end loop
  end var
end process
```

Figure 7: The LNT structure corresponding to the initiator CSL state-machine

4 The formal analysis

The first goal of our analysis has been the proof that all the three generated models are equivalent. This has been done by saving the possible behavior of the models in the form of Labelled Transition System [9], and by applying comparison tools [10] to verify that the three ProB and LNT models are strongly equivalent to the UMC models [11]. The main goal of the demonstrator, however, is *not* the complete formal verification of a (fragment of a) standard, but the exemplification of the *categories* of costs and benefits that may come to play with the choice of exploiting formal methods for the improvement of system requirements documents.The focus of our formal analysis is, therefore, to show with some evidence *how* formal methods may be of help in detecting the design errors potentially introduced while producing the UML executable model, in verifying the high-level properties expected by the full system and by its specific components, and in generating clear and rigorous (graphical) feedback on the specified system to the requirements designers. The detailed analysis of the costs and benefits, not only qualitative but also as far as possible quantitative, is the object of a separate 4SECURail deliverable [31]).

From our experience, it has become evident that formal methods can be used in a lightweight (i.e., almost "push button") way or in an "advanced" way. These two degrees of exploitation of formal methods require a very different level of effort and background. A rigorous static analysis of the formal models is probably the simplest example of lightweight use of formal methods. Just loading a system specification in the verification tool may immediately reveal mistakes and anomalies in the code (type violation, non-relevant updates, missing initializations, mismatch of parameters in messages, etc.).

Other behavioral properties like the absence of deadlocks or examples of reachability of certain states or events can still be verified with just a button-pushing or by writing extremely simple logical properties. Trace examples or counter-examples can be visualised in the form of a UML message sequence diagram [12]. Further information can be gathered by monitoring the generation and the statistics on the system state-space (if not too large). The visualization of state-space projections (i.e., graphical views of the system state-space once reduced after making observable only some specific detail of the system) can be of great help in understanding and confirming the system behavior without resorting to the encoding of complex temporal logic formulas.

The analysis of more complex behavioral properties, however, may require the writing of more complex temporal logics formulas. This activity may require a greater background and more advanced knowledge of the verification tools. Figure 8 shows a table of *some* of the features provided by our three frameworks. In the table, the features that can be easily exploited without any particularly advanced formal methods and tool knowledge, in an almost "push button" way, are those appearing in black. As it can be seen by observing the mentioned table of features, an advantage of our "formal methods diversity" approach is the possibility of exploiting the power offered by the whole set of frameworks, like state- or event-based model checking, linear- or branching-time model checking, state-space projections, custom system observations, and various state-space minimizations or reductions. In our experimentation, the following features have been the most used (more details can be found in Deliverable D2.5 [17]:

- static analysis in UMC/ProB/LNT

[9] While in the case of CADP and UMC saving a model in the .aut textual LTS is available as a builtin feature of the framework, in the case of ProB this LTS generation has been obtained through a automated transformation of the model state-space originally saved in the ProB ".statespace" textual format.

[10] e.g. mCRL2 ltscompare or CADP bcg_cmp.

[11] UMC can be configured to associate the LTS transition labels with the UMC transition labels, or with the occurring communications actions, or with other observable events.

[12] This is natively possible in the UMC framework and very recently also in the Tck/Tk version of ProB.

- explanations and animations in UMC/ProB
- weak-complete-divergence-sensitive-trace generation in UMC
- fast state-space generation in UMC
- divbranching minimizations in CADP

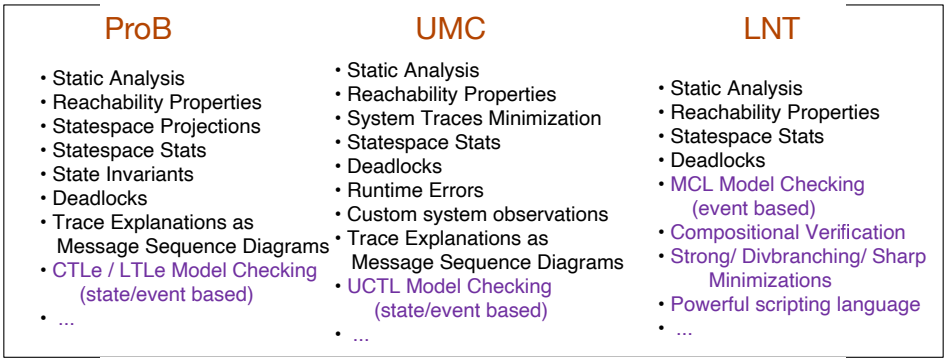

Figure 8: Table of verification features

When the same feature is available on multiple platforms, also usability aspects play an important role in selecting which one to exploit. E.g. CADP does not allow to observe the evolution of the values of the process variables during the animation of the behavior or the observation of a counter-example, ProB and UMC have richer visualization system, allowing among the other things to observe a trace in the form of a Sequence Message chart, SVL scripting in CADP makes easier the structuring and documentation of the ongoing verification process. The downside of this *formal methods diversity* approach is that becoming expert in the use of all these frameworks is likely to require a steep learning curve, with the needed single effort to be multiplied by the number of frameworks and with the risk of not becoming expert in any of them.

4.1 Properties and scenarios

The formal analysis of the system that has been performed during the the project activity is surely not complete, but sufficient to become reasonably confident in the absence of implementation or logical errors. Further tests and verification are still in progress, e.g., from the point of view of compositional verification in the context of the CADP/LNT framework. Several kinds of architectures can be generated to observe the system properties or the properties of single components. Figure 3 shows the case of a “complete” architecture, where all the system components are composed together with the needed environment components. Several flavors of this architecture can be designed, depending on the properties we want to observe, on the limit to the complexity we want to set, and on the kind of behavior of the environment we want to consider. Once the desired behavior of the environment components is established, they need to be instantiated into an executable scenario with the setting of a list of internal parameters fixing the parametric aspects of the specification. The simplest architecture we have built is when the two RBC remain “silent” (not sending any message) and just receive connection/disconnection indications from the CSL layer. In this architecture, the Euroradio level is imagined to be “nice”, i.e., introducing at most small delays in the communications, not losing nor reordering messages, and not autonomously aborting the existing active communication channel. The set of UML state-machine diagrams describing the components of this architecture is shown in Appendix A. In this case, the system is

simply expected to set up a communication line, keep it alive by exchanging life-signs, and re-establish it in case of failures. Failures can still occur depending on the specific values of the parameters used to instantiate the scenario. In particular, the most important parameters affecting the system behavior in this scenario are:

- The timeout (max_connectTimer) representing the maximum delay that initiator CSL is allowed to wait before receiving a reply to a connection request (after which a new connection request can be retried).
- The timeout (max_initTimer) representing the maximum delay that each SAI is allowed to wait for the successful initialization of a new communication line before aborting the creation process.
- The timeout (max_sendTimer) triggering the periodic sending by the CSL of a new life-sign to keep the communication line alive.
- The timeout (max_receiveTimer) representing the maximum delay a CSL is allowed to wait before receiving a life-sign or a rbc message from the other side (whose expiration causes the abort of the re-establishment of the communication line).

Other important system parameters, like

- The limit (N) of consecutive loss of messages (detected by the observation of sequence numbers) acceptable by the SAI components, before aborting the safe connection line.
- The maximum traveling delay (K) acceptable for incoming messages whose violation forces the discarding of the message.

do not play a relevant role in this scenario.

In this case, we can observe that if the connection, initialization, and receive timer are sufficiently large[13] (e.g., max_connectTimer = max_initTimer = 20, max_receiveTimer = 15, max_sendTimer = 5) the system successfully establishes an initial communication line without ever losing it. If we instead reduce the max_receiveTimer parameter to 8, communication failures and communications line restarts begin to appear (and the system state-space grows from 19,788,895 to 74,713,472 states).

An extension of the previous architecture is where the RBCs are allowed to send slots of "nmax" messages. In this case, we can observe how messages, if they arrive, are delivered to the target RBC without reordering, duplications, and within a maximum delay. In this case, the system state-space grows to 65,386,049 and to 84,883,327 states when slots of 1 or 2 messages are sent by just the RBC on the initiator side. Beyond the complete architectures described above, other kinds of architectures have been set up. For example, an "ICSL testing" architecture, where the Initiator CSL component is stimulated with an abstract model of the SAI and RBC components, and an "Initiator-side testing" architecture, where the whole ER layer and CSL/SAI/RBC on the "called side" of the system are abstracted by environment components. In this latter case, we can observe how the messages received from the RBC environment, in the absence of disconnections, are always delivered to the EuroRadio level without losses, duplications, reordering, and within a limited delay.

Further examples of the verifications that have been done on the models can be found on [17].

5 Related works

The experimentation of formal methods diversity for the analysis of the same specification has already been described by one of the authors in [22, 21]. In that case, the focus was on a much simpler case

[13] time unit are measured as multiples of the basic system execution cycle.

study that did not have the complexity of a parametric signaling standard. As a collateral activity of the project, the same fragment of UNISIG SUBSET 98 has been modelled and verified with UPPAAL by Basile et al. in [2]. The current translation of UML state-machine diagrams into ProB has been initially experimented in [8], but other approaches are possible; the UML-like UML-B notation [25, 24] has been proposed as a bridge between an Eclipse-based model framework (Rodin) and the Event-B modeling notation; the suggested approach seems, however, to be tailored to the verification and refinement of single state-machines and not to the analysis of the overall behavior of a set of interacting state-machines. Many other formal notations have been the target of translations from SysML design. Another work very similar to our from the point of view of the goal is the one described by Bouwman et al. [5]. Also in that case the goal was aimed at the analysis of a signaling standard under development rather than the verification of a specific system. The target notation and framework is, in that case, mCRL2.

6 Conclusions

Formal analysis of a still fluid, parametric, and environment-depending requirements specification (i.e., requirements elicitation and validation) is a very different kind of activity than verifying that a given implementation is correct with respect to a specific, stable, and rigorous specification. The possibility to exploit the analysis features offered by more than one verification framework can be of help in approaching this activity. Moreover, the design of several different scenarios can be necessary to observe the system behavior under various assumptions. From this point of view, the possibility to *automatically* generate the formal models to be analyzed from some executable, widely known, standard, tool-independent notation is a crucial point to make the analysis process accessible also from to people with the relevant railway-signaling knowledge. The formal methods diversity approach experienced in the project has shown how a lightweight use of formal verification frameworks can already, with a small effort, produce important feedback on the quality of the design. A deeper and more advanced exploitation of all the available features, however, remains a difficult and daunting task, especially when the system complexity and size grow to a level requiring ad hoc mitigation approaches. The activity shown with our experimentation can be continued and improved in several directions. The executable UML subset used in the project can be greatly extended, still preserving its clear and rigorous semantics and its possibility of automatic translation into several formal notations. Also, the set of target formal notations (currently limited to UMC, ProB, and LNT) can be extended with a likely small effort to further frameworks like mCRL2, Spin, nuXmv, just to mention some. The detailed description of the project results, the initial executable UML designs, their formal encoding, the source code of the translators, are all publicly available [1, 18, 19].

Acknowledgements

This work has been partially funded by the 4SECURail project. The 4SECURail project received funding from the Shift2Rail Joint Undertaking under the European Union's Horizon 2020 research and innovation programme under grant agreement No 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 content of this paper reflects only the authors' view and the Shift2Rail Joint Undertaking is not responsible for any use that may be made of the included information. We are grateful to the colleagues of the Work Stream 1 of project 4SECURail, and in particular to Alessandro Fantechi, Stefania Gnesi, Davide Basile, Alessio Ferrari, Maurice ter Beek, Andrea Piattino, Laura Masullo and Daniele Trentini for the comments and suggestions during the project.

References

[1] F. Mazzanti et al. (2020): *Work Stream 1 Deliverables*. 4SECURail, doi:10.5281/zenodo.5807738.

[2] D. Basile, A. Fantechi & I. Rosadi (2021): *Formal Analysis of the UNISIG Safety Application Intermediate Sub-layer - Applying Formal Methods to Railway Standard Interfaces*. In Alberto Lluch-Lafuente & Anastasia Mavridou, editors: *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021*, *LNCS* 12863, Springer, pp. 174–190, doi:10.1007/978-3-030-85248-1_11.

[3] M. ter Beek, S. Gnesi & F. Mazzanti (2015): *From EU Projects to a Family of Model Checkers - From Kandinsky to KandISTI*. In Rocco De Nicola & Rolf Hennicker, editors: *Software, Services, and Systems*, *LNCS* 8950, Springer, pp. 312–328, doi:10.1007/978-3-319-15545-6_20.

[4] M.H. ter Beek, A. Fantechi, S. Gnesi & F. Mazzanti (2011): *A state/event-based model-checking approach for the analysis of abstract system properties*. *Science of Computer Programming* 76(2), pp. 119–135, doi:10.1016/j.scico.2010.07.002.

[5] M. Bouwman, D van der Wal, Luttik, M. Stoelinga & A. Rensink (2020): *What is the Point: Formal Analysis and Test Generation for a Railway Standard*. In Piero Baraldi, Francesco Di Maio & Enrico Zio, editors: *Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference*, pp. 921–928, doi:10.3850/978-981-14-8593-0_4410-cd.

[6] A. Ferrari, F. Mazzanti, D. Basile, M. H. ter Beek & A. Fantechi (2020): *Comparing Formal Tools for System Design: a Judgment Study*. In: *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, ACM, pp. 62–74, doi:10.1145/3377811.3380373.

[7] A. Ferrari, F. Mazzanti, D. Basile & M. Ter Beek (2021): *Systematic Evaluation and Usability Analysis of Formal Methods Tools for Railway Signaling System Design*. *IEEE Transactions on Software Engineering*, pp. 1–1, doi:10.1109/TSE.2021.3124677.

[8] A. Ferrari, F. Mazzanti, D. Basile, A. Fantechi, S. Gnesi, D. Trentini, A. Piattino & B. Sturani (2012): *ASTRAIL Deliverable D4.3 - Validation Report*. Available at `http://www.astrail.eu/download.aspx?id=d7ae1ebf-52b4-4bde-b25e-ae251fd906df`.

[9] H. Garavel & F. Lang (2001): *SVL: A Scripting Language for Compositional Verification*. In: *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, August 28-31, 2001, Cheju Island, Korea*, *IFIP Conference Proceedings* 197, Kluwer, pp. 377–394, doi:10.1007/0-306-47003-9_24.

[10] H. Garavel, F. Lang, R. Mateescu & W. Serwe (2013): *CADP 2011: a toolbox for the construction and analysis of distributed processes*. *Int. J. Softw. Tools Technol. Transf.* 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z.

[11] H. Garavel, F. Lang & W. Serwe (2017): *From LOTOS to LNT*. In: *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10500, Springer, pp. 3–26, doi:10.1007/978-3-319-68270-9_1.

[12] S. Gnesi & F. Mazzanti (2011): *An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems*, pp. 390–407. *Lecture Notes in Computer Science* 6582, Springer, doi:10.1007/978-3-642-20401-2_18.

[13] F. Lang, R. Mateescu & F. Mazzanti (2020): *Sharp Congruences Adequate with Temporal Logics Combining Weak and Strong Modalities*. In Armin Biere & David Parker, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, *Lecture Notes in Computer Science* 12079, Springer, pp. 57–76, doi:10.1007/978-3-030-45237-7_4.

[14] F. Lang, R. Mateescu & F. Mazzanti (2021): *Compositional verification of concurrent systems by combining bisimulations*. *Formal Methods in System Design*, doi:10.1007/s10703-021-00360-w.

[15] M Leuschel & M.J. Butler (2008): *ProB: an automated analysis toolset for the B method*. Int. J. Softw. Tools Technol. Transf. 10(2), pp. 185–203, doi:10.1007/s10009-007-0063-9.

[16] F. Mazzanti, D. Basile, A. Fantechi, S. Gnesi & A. Ferrari (2020): *D2.1: Specification of formal development demonstrator*. In: Work Stream 1 Deliverables, 4SECURail, doi:10.5281/zenodo.5807738.

[17] F. Mazzanti & D. Belli (2020): *D2.1: Formal development demonstrator prototype, final release*. In: Work Stream 1 Deliverables, 4SECURail, doi:10.5281/zenodo.5807738.

[18] F. Mazzanti & D. Belli (2020): *Formal models of the SAI /CSL systems of the 4SECURail case study*, doi:10.5281/zenodo.5541307.

[19] F. Mazzanti & D. Belli (2020): *The UMC2LNT and UMC2PROB model transformation tools*, doi:10.5281/zenodo.5541350.

[20] F. Mazzanti & D.Belli (2022): *The 4SECURail Formal Methods Demonstrator*. In: The 4th International Conference on Reliability, Safety and Security of Railway Systems (RSSRAIL), Lecture Notes in Computer Science 13294, Springer, doi:10.5281/zenodo.6245955.

[21] F. Mazzanti & A. Ferrari (2018): *Ten Diverse Formal Models for a CBTC Automatic Train Supervision System*. In John P. Gallagher, Rob van Glabbeek & Wendelin Serwe, editors: Proceedings of the 3rd Workshop on Models for Formal Analysis of Real Systems and the 6th International Workshop on Verification and Program Transformation (MARS/VPT'18), EPTCS 268, pp. 104–149, doi:10.4204/EPTCS.268.4.

[22] F. Mazzanti, A. Ferrari & G. O. Spagnolo (2018): *Towards formal methods diversity in railways: an experience report with seven frameworks*. Int. J. Softw. Tools Technol. Transf. 20(3), pp. 263–288, doi:10.1007/s10009- 018-0488-3.

[23] A. Piattino (2020): *D2.1: Case study requirements and specification*. In: Work Stream 1 Deliverables, 4SECURail, doi:10.5281/zenodo.5807738.

[24] S. Salunkhe, R. Berglehner & A. Rasheeq (2021): *Automatic Transformation of SysML Model to Event-B Model for Railway CCS Application*. In Alexander Raschke & Dominique Méry, editors: Rigorous State-Based Methods - 8th International Conference, ABZ 2021, Ulm, Germany, June 9-11, 2021, Proceedings, Lecture Notes in Computer Science 12709, Springer, pp. 143–149, doi:10.1007/978-3-030-77543-8_14.

[25] C. F. Snook & M. J. Butler (2006): *UML-B: Formal modeling and design aided by UML*. ACM Trans. Softw. Eng. Methodol. 15(1), pp. 92–122, doi:10.1145/1125808.1125811.

[26] Geneva. ISO/IEC International Organization for Standardization Information Technology (1989): *International Standard 8807 - LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*.

[27] Geneva. ISO/IEC International Organization for Standardization Information Technology (2001): *International Standard 15437:2001 - Enhancements to LOTOS (E-LOTOS)*.

[28] UNISIG (2012): *SUBSET-098, RBC/RBC Safe Communication Interface, v3.0.3*. Available at `https://www.era.europa.eu/sites/default/files/filesystem/ertms/ccs_tsi_annex_a_-_mandatory_specifications/set_of_specifications_1_etcs_b2_gsm-r_b1/index063_-_subset-098_v100.pdf`.

[29] UNISIG (2015): *SUBSET-037, EuroRadio FIS v3.2.0*. Available at `https://www.era.europa.eu/sites/default/files/filesystem/ertms/ccs_tsi_annex_a_-_mandatory_specifications/set_of_specifications_3_etcs_b3_r2_gsm-r_b1/index010_-_subset-037_v320.pdf`.

[30] UNISIG (2015): *SUBSET-039, FIS for the RBC/RBC Handover v3.2.0*. Available at `https://www.era.europa.eu/sites/default/files/filesystem/ertms/ccs_tsi_annex_a_-_mandatory_specifications/set_of_specifications_3_etcs_b3_r2_gsm-r_b1/index012_-_subset-039_v320.pdf`.

[31] C. Vaghi (2021): *Specification of Cost-Benefit Analysis and learning curves, final release*. In: Work Stream 1 Deliverables, 4SECURail, doi:10.5281/zenodo.5807738.

Appendix A: UML diagrams for all the CSL and SAI components

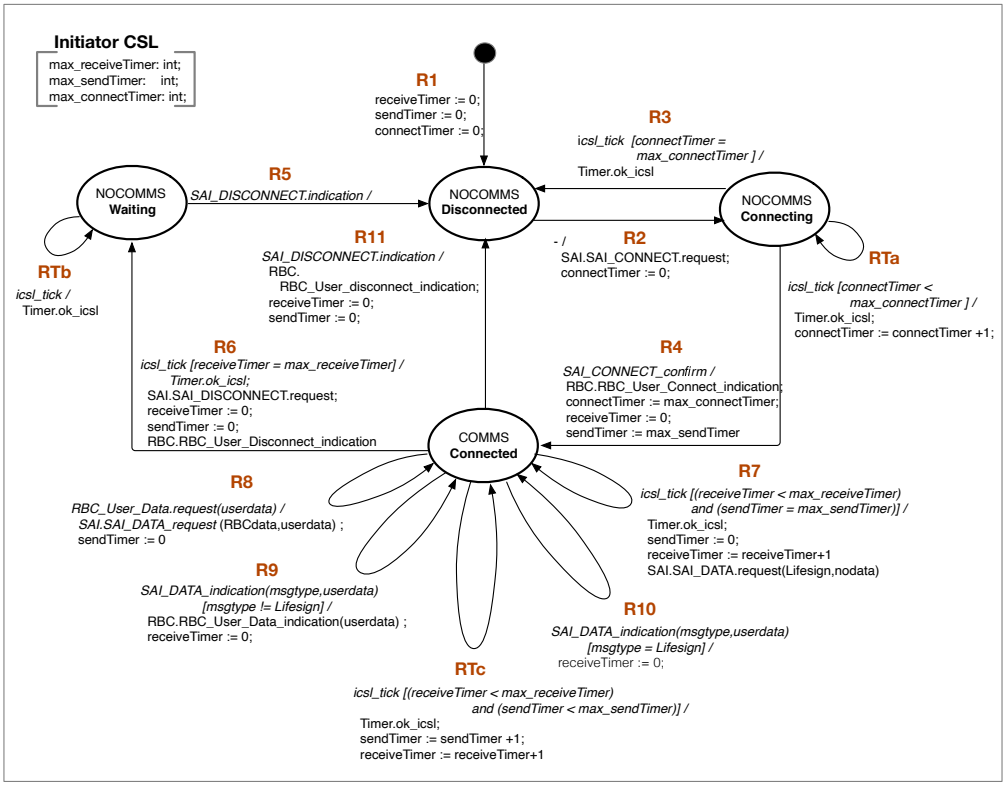

Figure 9: The Initiator CSL state-machine

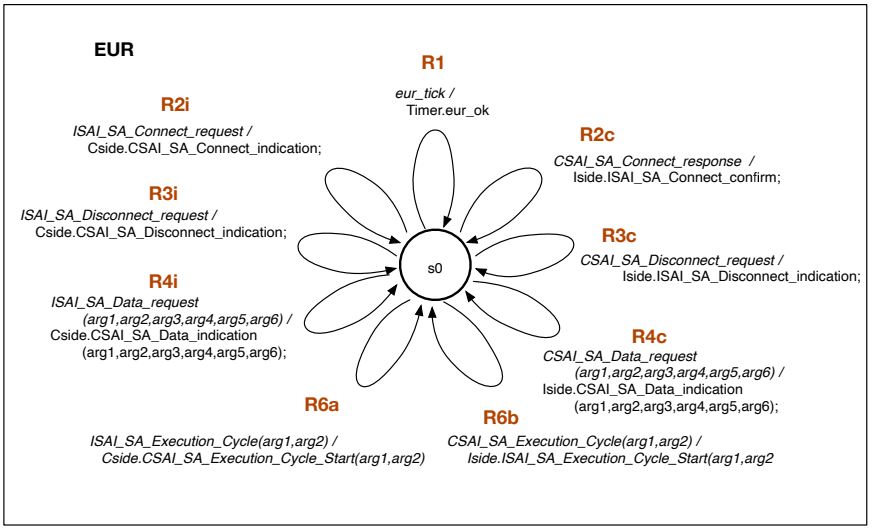

Figure 10: The *nice* Euroradio component of the Full scenario

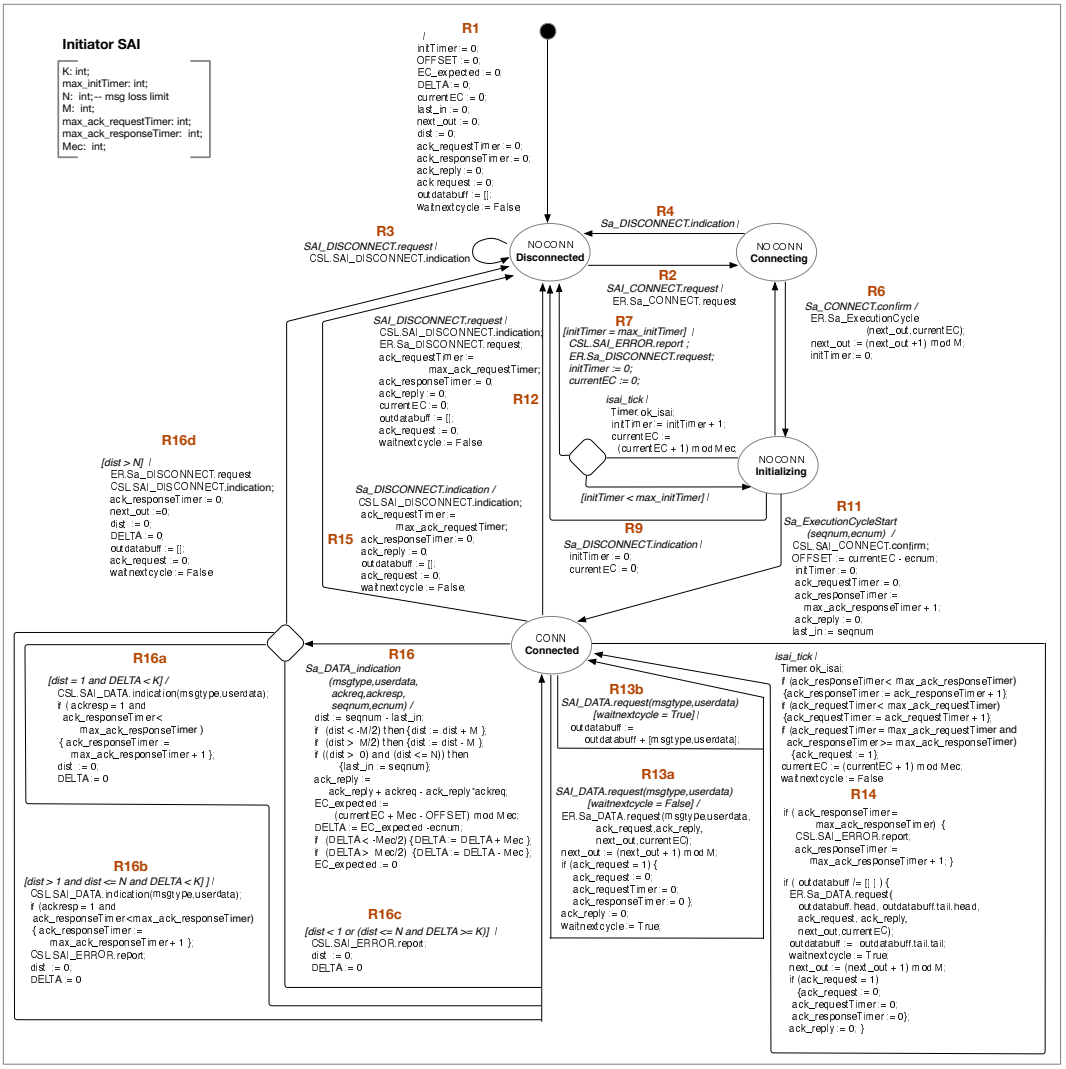

Figure 11: The Initiator SAI state-machine

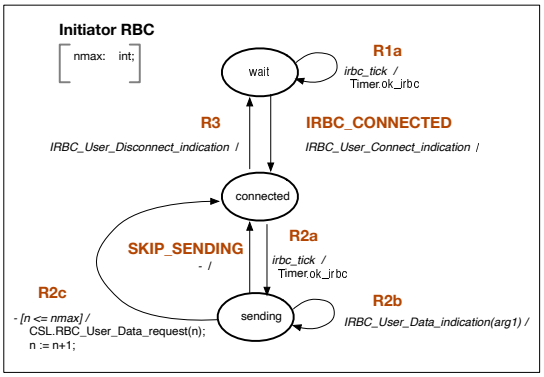

Figure 12: The Initiator-side RBC state-machine

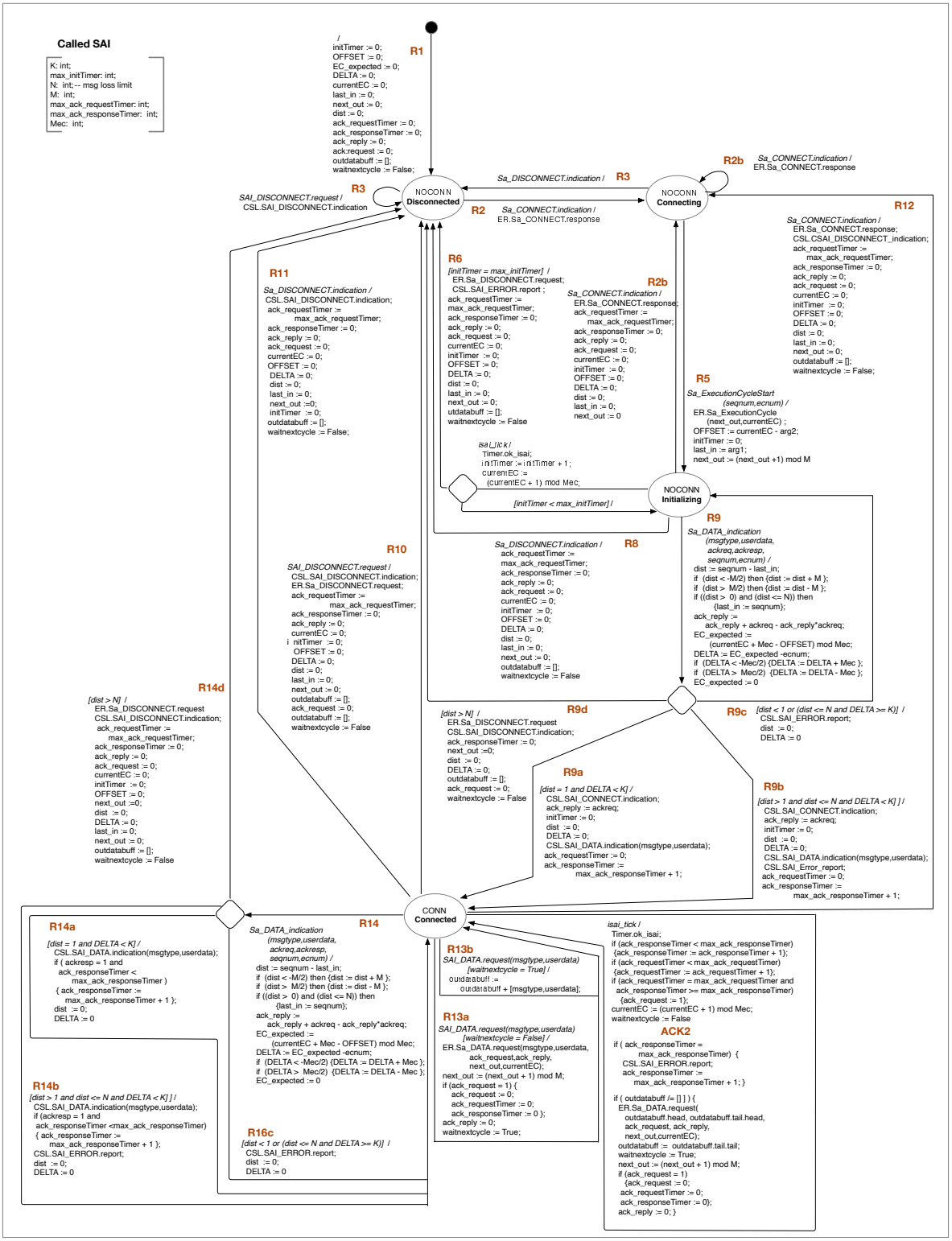

Figure 13: The Called SAI state-machine

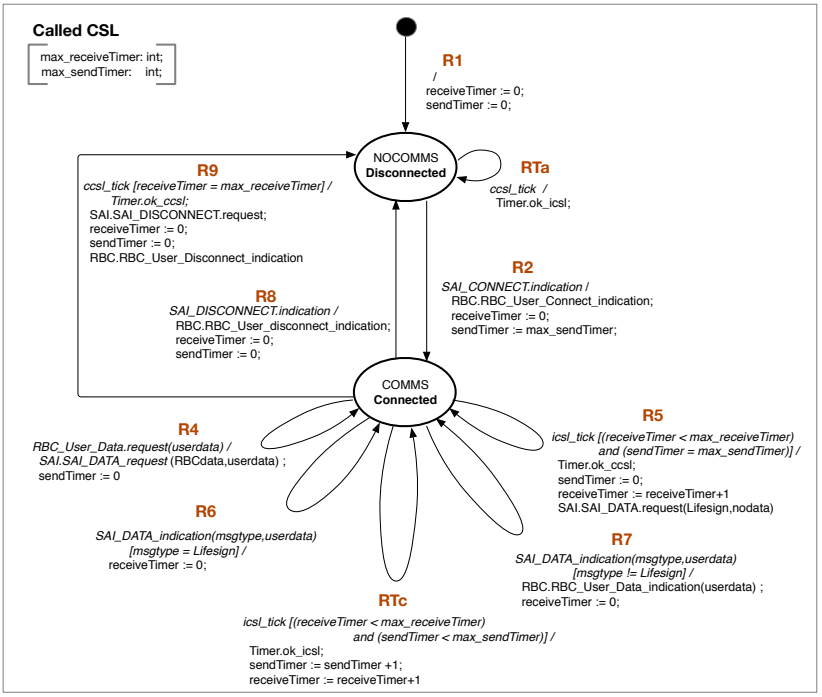

Figure 14: The Called CSL state-machine

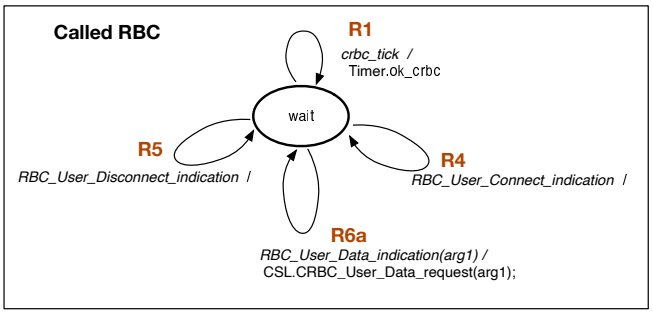

Figure 15: The Called-side RBC state-machine

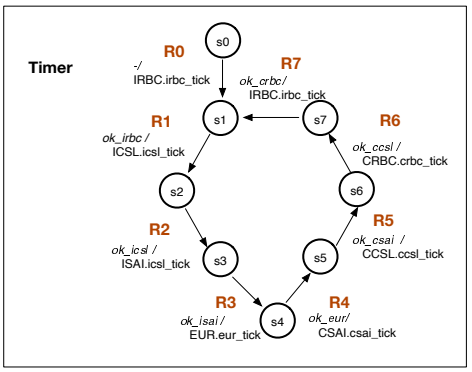

Figure 16: The Timer component of the Full scenario

Appendix B: UMC encoding of the initiator CSL class

```
-----------------------------------------------
Class I_CSL is
-----------------------------------------------
Signals
-- from RBC
IRBC_User_Data_request(arg1: int);
-- from I_SAI
ISAI_CONNECT_confirm;
ISAI_DISCONNECT_indication;
ISAI_Error_report;
ISAI_DATA_indication(arg1: Token, arg2: int);
-- from Timer
icsl_tick;
-- outgoing to RBC
-- IRBC_User_Connect_indication;
-- IRBC_User_Disconnect_indication;
-- IRBC_User_Data_indication(arg1:int);
-- outgoig to SAI
-- ISAI_CONNECT_request;
-- ISAI_DISCONNECT_request;
-- ISAI_DATA_request(arg1:Token,arg2:int);
-- outgoig to Timer
-- ok_icsl

Vars
------------- PORTS
RBC_User: I_RBC;
SAI: I_SAI;
--
------------- CONFIGURATION PARAMS
max_receiveTimer: int; -- CONFIGURATION PARAM
max_sendTimer: int;    -- CONFIGURATION PARAM
max_connectTimer: int; -- CONFIGURATION PARAM
--
------------- LOCAL VARS
receiveTimer: int := 0;
sendTimer: int := 0;
connectTimer: int := 0;

Behaviour
-----------------------------------------------
-- R1: At startup, the CSL is in Disconnected
-- state
-----------------------------------------------
R1_ICSL:
initial -> NOCOMMSready

-----------------------------------------------
-- R2: When in Disconnected state, the CSL
-- immediately sends a SAI_CONNECT.request to
-- the SAI component,starts a connTimer, and
-- moves to Connecting state
-----------------------------------------------
R2_ICSL_connecting:
NOCOMMSready -> NOCOMMSconnecting
{- /
 SAI.ISAI_CONNECT_request;
 connectTimer := 0;}

-----------------------------------------------
-- R3: When in Connecting state the connTimer
-- expires, the CSL moves to Disconnected
-- state. While connecting in case of timeout
-- become ready to retry.
-----------------------------------------------
R3_ICSL_okicsl_connect:
NOCOMMSconnecting -> NOCOMMSready
{icsl_tick [connectTimer = max_connectTimer] /
 Timer.ok_icsl}

-----------------------------------------------
-- R4: When in Connecting state is received a
-- SAI_CONNECT.confirm from the SAI component,
-- the CSL sends a RBC_User_Connect.indication
-- to the RBC component, starts both the
-- sendTimer and the recTimer,and moves to
-- Connected state
-----------------------------------------------
R4_ICSL_userconnind:
NOCOMMSconnecting -> COMMS
{ISAI_CONNECT_confirm /
 RBC_User.IRBC_User_Connect_indication;
 connectTimer := max_connectTimer;
 receiveTimer := 0;
 sendTimer := max_sendTimer}

-----------------------------------------------
-- R5: When in Waiting state is received a
-- SAI_DISCONNECT.indication from the SAI
-- component, the CSL moves to Disconnected
-- state
-----------------------------------------------
R5_ICSL_becomeready:
NOCOMMSwait -> NOCOMMSready
{ISAI_DISCONNECT_indication}

-----------------------------------------------
-- R6: When in Connected state the recTimer
-- expires, the CSL sends a
-- SAI_DISCONNECT.request to the SAI
-- component, a RBC_User_Disconnect.indication
-- to the RBC and moves to Waiting state
-----------------------------------------------
R6_ICSL_okicsl:
COMMS -> tbcr6
{icsl_tick [receiveTimer = max_receiveTimer] /
 Timer.ok_icsl}

R6_ICSL_saidisconnreq:
tbcr6 -> tbcr6a
{ - /
 SAI.ISAI_DISCONNECT_request;
 receiveTimer := 0;
 sendTimer := 0}

R6_ICSL_userdisconnind:
tbcr6a -> NOCOMMSwait
{ - /
 RBC_User.IRBC_User_Disconnect_indication}

-----------------------------------------------
-- R7: Each time that in Connected state the
-- sendTimer expires, the CSL sends a
-- SAI_DATA.request with a life_sign to the
-- SAI component
-----------------------------------------------
R7_ICSL_okicsl:
COMMS -> tbcr7
{icsl_tick [(receiveTimer < max_receiveTimer)
   and (sendTimer = max_sendTimer)] /
 Timer.ok_icsl;
 sendTimer := 0;
 receiveTimer := receiveTimer+1}

R7_ICSL_saidatareq:
tbcr7 -> COMMS
{ - /
 SAI.ISAI_DATA_request(LifeSign,0)}
```

```
-----------------------------------------------
-- R8: When in Connected state is received a
-- RBC_User_Data.request with RBC_data from
-- the RBC component, the CSL sends a
-- SAI_DATA.request with such RBC_data to the
-- AI component
-----------------------------------------------
R8_ICSL_saidatareq:
COMMS -> COMMS
{IRBC_User_Data_request(arg1) /
 SAI.ISAI_DATA_request(RBCdata, arg1);
 sendTimer := 0}

-----------------------------------------------
-- R9: When in Connected state is received a
-- SAI_DATA.indication with SAI_data
-- from the SAI component, the CSL sends a
-- RBC_User_Data.indication with
-- such SAI_data to the RBC component and
-- restarts the recTimer
-----------------------------------------------
R9_ICSL_userdataind:
COMMS -> COMMS
{ISAI_DATA_indication(arg1, arg2)
   [arg1 /= LifeSign] /
 RBC_User.IRBC_User_Data_indication(arg2);
 receiveTimer := 0}

-----------------------------------------------
-- R10: When in Connected state is received a
-- SAI_DATA.indication with a life_sign from
-- the SAI component, the CSL restarts the
-- recTimer
-----------------------------------------------
R10_ICSL_handlelifesign:
COMMS -> COMMS
{ISAI_DATA_indication(arg1, arg2)
   [arg1 = LifeSign] /
 receiveTimer := 0}

-----------------------------------------------
-- R11: When in Connected state is received a
-- SAI_DISCONNECT.indication from the
-- SAI component, the CSL sends a
-- RBC_User_Disconnect.indication to the RBC
-- component and moves to Disconnected state
-----------------------------------------------
R11_ICSL_userdisconnind:
COMMS -> NOCOMMSready
{ISAI_DISCONNECT_indication /
 RBC_User.IRBC_User_Disconnect_indication;
 receiveTimer := 0;
 sendTimer := 0}

-----------------------------------------------
-- RD1: When in Disconnected state the CSL
-- does not accept any kind of message
-- RD2: When in Connecting state, the CSL
-- discards any message except for
-- SAI_CONNECT.confirm from the SAI component
-- RD3: When in Waiting state, the CSL
-- discards any message except for
-- SAI_DISCONNECT.indication from the SAI
-- component
-- RD4: When in Connected state, the CSL
-- component discards only SAI_CONNECT.confirm
-- and SAI_ERROR.report messages from
-- the SAI component
-----------------------------------------------
RD2a_ICSL_discuserdata:
NOCOMMSconnecting -> NOCOMMSconnecting
{IRBC_User_Data_request(arg1)}

RD2b_ICSL_discdisconnind:
NOCOMMSconnecting -> NOCOMMSconnecting
{ISAI_DISCONNECT_indication}

RD2c_ICSL_discerrorreport:
NOCOMMSconnecting -> NOCOMMSconnecting
{ISAI_Error_report}

RD2d_ICSL_discdataind:
NOCOMMSconnecting -> NOCOMMSconnecting
{ISAI_DATA_indication(arg1, arg2)}

RD3a_ICSL_discuserdata:
NOCOMMSwait -> NOCOMMSwait
{IRBC_User_Data_request(arg1)}

RD3b_ICSL_discerrorreport:
NOCOMMSwait -> NOCOMMSwait
{ISAI_Error_report}

RD3c_ICSL_discdataind:
NOCOMMSwait -> NOCOMMSwait
{ISAI_DATA_indication(arg1, arg2)}

RD3d_ICSL_discconfirm:
NOCOMMSwait -> NOCOMMSwait
{ISAI_CONNECT_confirm}

RD4a_ICSL_disccommconfirm:
COMMS -> COMMS
{ISAI_CONNECT_confirm}

RD4b_ICSL_usererror:
COMMS -> COMMS
{ISAI_Error_report}

------------------------
-- clock cycles handling
------------------------
RTa_ICSL_okicsl_incr:
NOCOMMSconnecting -> NOCOMMSconnecting
{icsl_tick [connectTimer < max_connectTimer] /
 Timer.ok_icsl;
 connectTimer := connectTimer +1}

RTb_ICSL_okicsl_incr:
NOCOMMSwait -> NOCOMMSwait
{icsl_tick /
 Timer.ok_icsl}

RTc_ICSL_okicsl:
 COMMS -> COMMS
{icsl_tick [(receiveTimer < max_receiveTimer)
   and (sendTimer < max_sendTimer)] /
 Timer.ok_icsl;
 sendTimer := sendTimer +1;
 receiveTimer := receiveTimer+1}

 end I_CSL;
```

Appendix C: ProB encoding of the initiator CSL class

```
MACHINE SYS
SETS
 ICSL_STATES = {
   ICSL_NOCOMMSready,
   ICSL_NOCOMMSconnecting,
   ICSL_COMMS,
   ICSL_NOCOMMSwait,
   ICSL_tbcr6,
   ICSL_tbcr6a,
   ICSL_tbcr7};
 ...

 ICSL_SIGNALS = {
   IRBC_User_Data_request,
   ISAI_CONNECT_confirm,
   ISAI_DISCONNECT_indication,
   ISAI_Error_report,
   ISAI_DATA_indication,
   icsl_tick};

DEFINITIONS
 SET_PREF_MAXINT == 30;
 SET_PREF_MININT == -30;

CONSTANTS
 LifeSign,
 RBCdata

PROPERTIES
 LifeSign = 2 &
 RBCdata = 3

VARIABLES
 // ICSL
 ICSL_max_receiveTimer,
 ICSL_max_sendTimer,
 ICSL_max_connectTimer,
 ICSL_receiveTimer,
 ICSL_sendTimer,
 ICSL_connectTimer,
 ICSL_eventsfifobuff,
 ICSL_eventsdatabuff,
 ICSL_STATE,
 ...

INVARIANT
 // ICSL
 ICSL_max_receiveTimer: MININT..MAXINT &
 ICSL_max_sendTimer: MININT..MAXINT &
 ICSL_max_connectTimer: MININT..MAXINT &
 ICSL_receiveTimer: MININT..MAXINT &
 ICSL_sendTimer: MININT..MAXINT &
 ICSL_connectTimer: MININT..MAXINT &
 ICSL_eventsfifobuff: seq(ICSL_SIGNALS) &
 ICSL_eventsdatabuff: seq(MININT..MAXINT) &
 ICSL_STATE: ICSL_STATES &
 ...

INITIALISATION
 // ICSL
 ICSL_max_receiveTimer := 15;
 ICSL_max_sendTimer := 05;
 ICSL_max_connectTimer := 20;
 ICSL_receiveTimer := 0;
 ICSL_sendTimer := 0;
 ICSL_connectTimer := 0;
 ICSL_eventsfifobuff := [];
 ICSL_eventsdatabuff :=  [];
 ICSL_STATE := ICSL_NOCOMMSready;
 ...

OPERATIONS
 ////////////////////////////////////////////////
 //   State Machine ICSL
 ////////////////////////////////////////////////
```

```
 // ------------------------------------------------
 // R2: When in Disconnected state, the CSL
 // immediately sends a SAI_CONNECT.request to
 // the SAI component, starts a connTimer, and
 // moves to Connecting state
 // ------------------------------------------------
R2_ICSL_connecting =
PRE
  ICSL_STATE = ICSL_NOCOMMSready
THEN
  ISAI_eventsfifobuff :=
    ISAI_eventsfifobuff <- ISAI_CONNECT_request;
  ICSL_connectTimer := 0;
  ICSL_STATE := ICSL_NOCOMMSconnecting
END;

// ------------------------------------------------
// R3: When in Connecting state the connTimer
// expires, the CSL moves to Disconnected state
// while connecting in case of timeout become
// ready to retry
// ------------------------------------------------
R3_ICSL_okicsl_connect =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  (ICSL_connectTimer = ICSL_max_connectTimer) &
   ICSL_STATE = ICSL_NOCOMMSconnecting
THEN
  Timer_eventsfifobuff :=
    Timer_eventsfifobuff <- ok_icsl;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_NOCOMMSready
END;



// ------------------------------------------------
// R4: When in Connecting state is received a
// SAI_CONNECT.confirm from the SAI component,
// the CSL sends a RBC_User_Connect.indication
// to the RBC component, starts both the
// sendTimer and the recTimer,and moves to
// Connected state
// ------------------------------------------------
R4_ICSL_userconnind =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff)=ISAI_CONNECT_confirm &
  ICSL_STATE = ICSL_NOCOMMSconnecting
THEN
  IRBC_eventsfifobuff :=
     IRBC_eventsfifobuff <-
       IRBC_User_Connect_indication;
  ICSL_connectTimer := ICSL_max_connectTimer;
  ICSL_receiveTimer := 0;
  ICSL_sendTimer := ICSL_max_sendTimer;
  //
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_COMMS
END;

// ------------------------------------------------
// R5: When in Waiting state is received a
// SAI_DISCONNECT.indication from the SAI
// component, the CSL moves to Disconnected state
// ------------------------------------------------
R5_ICSL_becomeready =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) =
     ISAI_DISCONNECT_indication &
  ICSL_STATE = ICSL_NOCOMMSwait
THEN
  skip;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_NOCOMMSready
END;
```

```
// ------------------------------------------------
// R6: When in Connected state the recTimer
// expires, the CSL sends a SAI_DISCONNECT.request
// to the SAI component, a
// RBC_User_Disconnect.indication to the RBC and
// moves to Waiting state
// ------------------------------------------------
R6_ICSL_okicsl =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  ICSL_receiveTimer = ICSL_max_receiveTimer &
  ICSL_STATE = ICSL_COMMS
THEN
  Timer_eventsfifobuff :=
     Timer_eventsfifobuff <- ok_icsl;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_tbcr6
END;

R6_ICSL_saidisconnreq =
PRE
  ICSL_STATE = ICSL_tbcr6
THEN
  ISAI_eventsfifobuff :=
    ISAI_eventsfifobuff <- ISAI_DISCONNECT_request;
  ICSL_receiveTimer := 0;
  ICSL_sendTimer := 0;
  ICSL_STATE := ICSL_tbcr6a
END;

R6_ICSL_userdisconnind =
PRE
  ICSL_STATE = ICSL_tbcr6a
THEN
  IRBC_eventsfifobuff :=
    IRBC_eventsfifobuff <-
      IRBC_User_Disconnect_indication;
  ICSL_STATE := ICSL_NOCOMMSwait
END;

// ------------------------------------------------
// R7: Each time that in Connected state the
// sendTimer expires, the CSL sends a
// SAI_DATA.request with a life_sign to the
//  SAI component
// ------------------------------------------------
R7_ICSL_okicsl =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  (ICSL_receiveTimer < ICSL_max_receiveTimer) &
  (ICSL_sendTimer = ICSL_max_sendTimer) &
  ICSL_STATE = ICSL_COMMS
THEN
  Timer_eventsfifobuff :=
     Timer_eventsfifobuff <- ok_icsl;
  ICSL_sendTimer := 0;
  ICSL_receiveTimer := ICSL_receiveTimer + 1;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_tbcr7
END;

R7_ICSL_saidatareq =
PRE
  ICSL_STATE = ICSL_tbcr7
THEN
  ISAI_eventsfifobuff :=
     ISAI_eventsfifobuff <- ISAI_DATA_request;
  ISAI_eventsdatabuff :=
     ISAI_eventsdatabuff <- LifeSign;
  ISAI_eventsdatabuff := ISAI_eventsdatabuff <- 0;
  ICSL_STATE := ICSL_COMMS
END;
```

```
// ------------------------------------------------
// R8: When in Connected state is received a
// RBC_User_Data.request with RBC_data from the RBC
// component, the CSL sends a SAI_DATA.request with
// such RBC_data to the SAI component
// ------------------------------------------------
R8_ICSL_saidatareq(arg1) =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) =
     IRBC_User_Data_request &
  arg1 = first(ICSL_eventsdatabuff) &
  ICSL_STATE = ICSL_COMMS
THEN
  ISAI_eventsfifobuff :=
     ISAI_eventsfifobuff <- ISAI_DATA_request;
  ISAI_eventsdatabuff :=
     ISAI_eventsdatabuff <- RBCdata;
  ISAI_eventsdatabuff :=
     ISAI_eventsdatabuff <- arg1;
  ICSL_sendTimer := 0;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_eventsdatabuff := tail(ICSL_eventsdatabuff);
  ICSL_STATE := ICSL_COMMS
END;

// ------------------------------------------------
// R9: When in Connected state is received a
// SAI_DATA.indication with SAI_data from the SAI
// component, the CSL sends a
// RBC_User_Data.indication with such SAI_data to
// the RBC component and restarts the recTimer
// ------------------------------------------------
R9_ICSL_userdataind(arg1,arg2) =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff)=ISAI_DATA_indication &
  arg1 = first(ICSL_eventsdatabuff) &
  arg2 = first(tail(ICSL_eventsdatabuff)) &
  arg1 /= LifeSign &
  ICSL_STATE = ICSL_COMMS
THEN
  IRBC_eventsfifobuff :=
    IRBC_eventsfifobuff <-
      IRBC_User_Data_indication;
  IRBC_eventsdatabuff :=
    IRBC_eventsdatabuff <- arg2;
  ICSL_receiveTimer := 0;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_eventsdatabuff :=
     tail(tail(ICSL_eventsdatabuff));
  ICSL_STATE := ICSL_COMMS
END;

// ------------------------------------------------
// R10: When in Connected state is received a
// SAI_DATA.indication with a life_sign from the
// SAI component, the CSL restarts the recTimer
// ------------------------------------------------
R10_ICSL_handlelifesign(arg1,arg2) =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff)=ISAI_DATA_indication &
  arg1 = first(ICSL_eventsdatabuff) &
  arg2 = first(tail(ICSL_eventsdatabuff)) &
  arg1 = LifeSign &
  ICSL_STATE = ICSL_COMMS
THEN
  ICSL_receiveTimer := 0;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_eventsdatabuff :=
      tail(tail(ICSL_eventsdatabuff));
  ICSL_STATE := ICSL_COMMS
END;
```

```
RD4b_ICSL_usererror =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = ISAI_Error_report &
  ICSL_STATE = ICSL_COMMS
THEN
  skip;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_COMMS
END;

// -------------------------
// -- clock cycles handling
// -------------------------
RTa_ICSL_okicsl_incr =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  (ICSL_connectTimer < ICSL_max_connectTimer)  &
  ICSL_STATE = ICSL_NOCOMMSconnecting
THEN
  Timer_eventsfifobuff :=
    Timer_eventsfifobuff <- ok_icsl;
  ICSL_connectTimer := ICSL_connectTimer + 1;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_NOCOMMSconnecting
END;
```

```
RTb_ICSL_okicsl_incr =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  ICSL_STATE = ICSL_NOCOMMSwait
THEN
  Timer_eventsfifobuff :=
    Timer_eventsfifobuff <- ok_icsl;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_NOCOMMSwait
END;

RTc_ICSL_okicsl =
PRE
  ICSL_eventsfifobuff /= [] &
  first(ICSL_eventsfifobuff) = icsl_tick &
  ((ICSL_receiveTimer < ICSL_max_receiveTimer) &
   (ICSL_sendTimer < ICSL_max_sendTimer)) &
  ICSL_STATE = ICSL_COMMS
THEN
  Timer_eventsfifobuff :=
    Timer_eventsfifobuff <- ok_icsl;
  ICSL_sendTimer := ICSL_sendTimer + 1;
  ICSL_receiveTimer := ICSL_receiveTimer + 1;
  ICSL_eventsfifobuff := tail(ICSL_eventsfifobuff);
  ICSL_STATE := ICSL_COMMS
END;
  ...
END
```

Appendix D: LNT encoding of the initiator CSL class

```
------------------------------
-- System wide definitions
------------------------------
function one: int is
   return 1
end function

function zero: int is
   return 0
end function

function LifeSign: int is
    return 2   -- value from UMC encoding
end function

function RBCdata: int is
    return 3
end function

channel Msg1 is
   (arg1: Int)
end channel

channel Msg2 is
    (arg1: Int,
     arg2: Int)
end channel
...

type Intlist is
  list of Int
with append, head, empty, length, tail, union, !=, ==
end type

channel DataMsg is
   (arglist: Intlist)
end channel

type Databuff is
  list of Intlist
with append, head, empty, length, tail, union, !=, ==
end type
...

------------------------------
-- ICSL  types
------------------------------
-- enumeration type for class signals used for
--  signals buff
type ICSL_signals is
   IRBC_User_Data_request,
   ISAI_CONNECT_confirm,
   ISAI_DISCONNECT_indication,
   ISAI_Error_report,
   ISAI_DATA_indication,
   icsl_tick
with  ==, !=
end type

type ICSL_BUFF is
  list of ICSL_signals
with append, head, empty, length, tail, !=, ==
end type

type ICSL_states is
  NOCOMMSready, NOCOMMSconnecting, COMMS,
  NOCOMMSwait, tbcr6, tbcr6a, tbcr7
with ==, !=
end type
```

```
------------------------------
-- ICSL State machine process
------------------------------
process ICSL [
 -- ICSL signals
 IRBC_User_Data_request: Msg1,
 ISAI_CONNECT_confirm: none,
 ISAI_DISCONNECT_indication: none,
 ISAI_Error_report: none,
 ISAI_DATA_indication: Msg2,
 icsl_tick: none,
 ISAI_CONNECT_request: none,
 ok_icsl: none,
 IRBC_User_Connect_indication: none,
 R5_ICSL_becomeready:none,
 ISAI_DISCONNECT_request: none,
 IRBC_User_Disconnect_indication: none,
 ISAI_DATA_request: Msg2,
 IRBC_User_Data_indication: Msg1,
 R10_ICSL_handlelifesign:none,
 RD2a_ICSL_discuserdata:none,
 RD2b_ICSL_discdisconnind:none,
 RD2c_ICSL_discerrorreport:none,
 RD2d_ICSL_discdataind:none,
 RD3a_ICSL_discuserdata:none,
 RD3b_ICSL_discerrorreport:none,
 RD3c_ICSL_discdataind:none,
 RD3d_ICSL_discconfirm:none,
 RD4a_ICSL_disccommconfirm:none,
 RD4b_ICSL_usererror:none] is
var
 max_receiveTimer: Int,
 max_sendTimer: Int,
 max_connectTimer: Int,
 receiveTimer: Int,
 sendTimer: Int,
 connectTimer: Int,
 mybuff: ICSL_BUFF,
 mynatbuff: Intlist,
 STATE: ICSL_STATES
in
 max_receiveTimer := 20;
 max_sendTimer := 1;
 max_connectTimer := 15;
 receiveTimer := 0;
 sendTimer := 0;
 connectTimer := 0;
 mybuff := nil;
 mynatbuff := nil;
 STATE := NOCOMMSready;
 loop
  select
   ----------------------------------------
   --- buffering incoming signals
   ----------------------------------------
   var arg1: Int in
    IRBC_User_Data_request(?arg1);
    mybuff :=
      append(IRBC_User_Data_request, mybuff);
    mynatbuff := append(arg1, mynatbuff)
   end var []
   ISAI_CONNECT_confirm;
   mybuff := append(ISAI_CONNECT_confirm, mybuff)
  []
   ISAI_DISCONNECT_indication;
   mybuff :=
     append(ISAI_DISCONNECT_indication, mybuff)
  []
   ISAI_Error_report;
   mybuff := append(ISAI_Error_report, mybuff)
 []
   var arg1, arg2: Int in
    ISAI_DATA_indication(?arg1, ?arg2);
    mybuff := append(ISAI_DATA_indication, mybuff);
    mynatbuff := append(arg1, mynatbuff);
    mynatbuff := append(arg2, mynatbuff)
   end var []
   icsl_tick;
   mybuff := append(icsl_tick, mybuff)
  []
```

```
----------------------------------------
-- triggering statemachine transitions from
-- events pool
----------------------------------------

 ----------------------------------------
 -- R2_ICSL_connecting
 ----------------------------------------
 -- R2: When in Disconnected state, the CSL
 -- immediately sends a SAI_CONNECT.request to
 -- the SAI component,starts a connTimer, and
 -- moves to Connecting state
 ------------------------------------------------
 only if
  STATE == NOCOMMSready
 then
  ISAI_CONNECT_request;
  connectTimer := 0 of int;
  STATE := NOCOMMSconnecting
 end if
[]
 -- R3_ICSL_okicsl_connect
 ------------------------------------------------
 -- R3: When in Connecting state the connTimer
 -- expires, the CSL moves to Disconnected state
 -- While connecting in case of timeout become
 -- ready to retry
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == icsl_tick and
  connectTimer == max_connectTimer and
  STATE == NOCOMMSconnecting
 then
  ok_icsl;
   null;
   mybuff := tail(mybuff);
   STATE := NOCOMMSready
 end if
[]
 -- R4_ICSL_userconnind
 ------------------------------------------------
 -- R4: When in Connecting state is received a
 -- SAI_CONNECT.confirm from the SAI component,
 -- the CSL sends a RBC_User_Connect.indication
 -- to the RBC component, starts both the
 -- sendTimer and the recTimer, and moves to
 -- Connected state
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_CONNECT_confirm and
  STATE == NOCOMMSconnecting
 then
  IRBC_User_Connect_indication;
  connectTimer := max_connectTimer;
  receiveTimer := 0 of int;
  sendTimer := max_sendTimer;
  mybuff := tail(mybuff);
  STATE := COMMS
 end if
[]
 -- R5_ICSL_becomeready
 ------------------------------------------------
 -- R5: When in Waiting state is received a
 -- SAI_DISCONNECT.indication from the SAI
 -- component, the CSL moves to Disconnected
 -- state
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff)==ISAI_DISCONNECT_indication and
  STATE == NOCOMMSwait
 then
  R5_ICSL_becomeready;
  null;
  mybuff := tail(mybuff);
  STATE := NOCOMMSready
 end if
[]
```

```
-- R6_ICSL_okicsl
 ------------------------------------------------
 -- R6:When in Connected state the recTimer
 -- expires the CSL sends SAI_DISCONNECT.request
 -- to the SAI component, a
 -- RBC_User_Disconnect.indication to the RBC and
 -- moves to Waiting state
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == icsl_tick and
  receiveTimer) == max_receiveTimer and
  STATE == COMMS
 then
  ok_icsl;
  null;
  mybuff := tail(mybuff);
  STATE := tbcr6
 end if
[]
 -- R6_ICSL_saidisconnreq
 only if
  (STATE == tbcr6)
 then
  ISAI_DISCONNECT_request;
  receiveTimer := 0 of int;
  sendTimer := 0 of int;
  STATE := tbcr6a
 end if
[]
 -- R6_ICSL_userdisconnind
 only if
  STATE == tbcr6a
 then
  IRBC_User_Disconnect_indication;
  null;
  STATE := NOCOMMSwait
 end if []
  -- R7_ICSL_okicsl
  ------------------------------------------------
  -- R7: Each time that in Connected state the
  -- sendTimer expires, the CSL sends a
 -- SAI_DATA.request with a life_sign to the
 -- SAI component
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == icsl_tick and
  receiveTimer < max_receiveTimer and
  sendTimer == max_sendTimer and
  STATE == COMMS
 then
  ok_icsl;
  sendTimer := 0 of int;
  receiveTimer := receiveTimer + 1 of int;
  mybuff := tail(mybuff);
  STATE := tbcr7
 end if
[]
 -- R7_ICSL_saidatareq
 only if
  STATE == tbcr7
 then
  ISAI_DATA_request(LifeSign,0 of int);
  null;
  STATE := COMMS
 end if
[]
 -- R8_ICSL_saidatareq
 ------------------------------------------------
 -- R8: When in Connected state is received a
 -- RBC_User_Data.request with RBC_data from the
 -- RBC component, the CSL sends a
 -- SAI_DATA.request with such RBC_data to the
 -- SAI component
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == IRBC_User_Data_request and
  STATE == COMMS
 then
  ISAI_DATA_request(RBCdata,head(mynatbuff));
  sendTimer := 0 of int;
  mybuff := tail(mybuff);
  mynatbuff := tail(mynatbuff);
  STATE := COMMS
 end if
[]
```

```
 -- R9_ICSL_userdataind
 ------------------------------------------------
 -- R9: When in Connected state is received a
 -- SAI_DATA.indication with SAI_data
 -- from the SAI component, the CSL sends a
 -- RBC_User_Data.indication with such SAI_data
 -- to the RBC component and restarts the
 -- recTimer
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_DATA_indication and
  (head(mynatbuff)) /= (LifeSign) and
  STATE == COMMS
 then
  IRBC_User_Data_indication(
    head(tail(mynatbuff)));
  receiveTimer := 0 of int;
  mybuff := tail(mybuff);
  mynatbuff := tail(mynatbuff);
  mynatbuff := tail(mynatbuff);
  STATE := COMMS
 end if
[]
 -- R10_ICSL_handlelifesign
 ------------------------------------------------
 -- R10: When in Connected state is received a
 -- SAI_DATA.indication with a life_sign from the
 -- SAI component, the CSL restarts the recTimer
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_DATA_indication and
  (head(mynatbuff)) == (LifeSign) and
  STATE == COMMS
 then
  R10_ICSL_handlelifesign;
  receiveTimer := 0 of int;
  mybuff := tail(mybuff);
  mynatbuff := tail(mynatbuff);
  mynatbuff := tail(mynatbuff);
  STATE := COMMS
 end if
[]
 -- R11_ICSL_userdisconnind
 ------------------------------------------------
 -- R11: When in Connected state is received a
 -- SAI_DISCONNECT.indication from the SAI
 -- component, the  CSL sends a
 -- RBC_User_Disconnect.indication to the RBC
 -- component and moves to Disconnected state
 ------------------------------------------------
 only if
  mybuff /= nil and
  head(mybuff)==ISAI_DISCONNECT_indication and
  STATE == COMMS
 then
  IRBC_User_Disconnect_indication;
  receiveTimer := 0 of int;
  sendTimer := 0 of int;
  mybuff := tail(mybuff);
  STATE := NOCOMMSready
 end if
[]
 -- RD2a_ICSL_discuserdata
 ------------------------------------------------
 -- RD1: When in Disconnected state the CSL does
 --  not accept any kind of message
 -- RD2: When in Connecting state, the CSL
 --  discards any message except for
 -- SAI_CONNECT.confirm from the SAI component
 -- RD3: When in Waiting state, the CSL discards
 --  any message except for
 -- SAI_DISCONNECT.indication from the SAI
 --  component
 -- RD4: When in Connected state, the CSL
 -- component discards only
 -- SAI_CONNECT.confirm and SAI_ERROR.report
 -- messages from the SAI component
 ------------------------------------------------
  only if
   mybuff /= nil and
   head(mybuff) == IRBC_User_Data_request and
   STATE == NOCOMMSconnecting
  then
   RD2a_ICSL_discuserdata;
   null;
   mybuff := tail(mybuff);
   mynatbuff := tail(mynatbuff);
   STATE := NOCOMMSconnecting
  end if
 []
  -- RD2b_ICSL_discdisconnind
  only if
   mybuff /= nil and
   head(mybuff)==ISAI_DISCONNECT_indication and
   STATE == NOCOMMSconnecting
  then
   RD2b_ICSL_discdisconnind;
   null;
   mybuff := tail(mybuff);
   STATE := NOCOMMSconnecting
  end if
 []
 -- RD2c_ICSL_discerrorreport
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_Error_report and
  STATE == NOCOMMSconnecting
 then
   RD2c_ICSL_discerrorreport;
   null;
   mybuff := tail(mybuff);
   STATE := NOCOMMSconnecting
  end if
[]
  -- RD2d_ICSL_discdataind
  only if
   mybuff /= nil and
   head(mybuff) == ISAI_DATA_indication and
   STATE == NOCOMMSconnecting
  then
   RD2d_ICSL_discdataind;
   null;
   mybuff := tail(mybuff);
   mynatbuff := tail(mynatbuff);
   mynatbuff := tail(mynatbuff);
   STATE := NOCOMMSconnecting
  end if
 []
  -- RD3a_ICSL_discuserdata
  only if
   mybuff /= nil and
   head(mybuff) == IRBC_User_Data_request and
   STATE == NOCOMMSwait
  then
   RD3a_ICSL_discuserdata;
   null;
   mybuff := tail(mybuff);
   mynatbuff := tail(mynatbuff);
   STATE := NOCOMMSwait
  end if
 []
  -- RD3b_ICSL_discerrorreport
  only if
   mybuff /= nil and
   head(mybuff) == ISAI_Error_report and
   STATE == NOCOMMSwait
  then
   RD3b_ICSL_discerrorreport;
   null;
   mybuff := tail(mybuff);
   STATE := NOCOMMSwait
  end if
 []
```

```
 -- RD3c_ICSL_discdataind
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_DATA_indication and
  STATE == NOCOMMSwait
 then
  RD3c_ICSL_discdataind;
  null;
  mybuff := tail(mybuff);
  mynatbuff := tail(mynatbuff);
  mynatbuff := tail(mynatbuff);
  STATE := NOCOMMSwait
 end if
[]
 -- RD3d_ICSL_discconfirm
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_CONNECT_confirm and
  STATE == NOCOMMSwait
 then
  RD3d_ICSL_discconfirm;
  null;
  mybuff := tail(mybuff);
  STATE := NOCOMMSwait
 end if
[]
 -- RD4a_ICSL_disccommconfirm
 only if
  mybuff /= nil and
  head(mybuff) == ISAI_CONNECT_confirm and
  STATE == COMMS
 then
  RD4a_ICSL_disccommconfirm;
  null;
  mybuff := tail(mybuff);
  STATE := COMMS
 end if
[]
 -- RD4b_ICSL_usererror
 only if
  (mybuff /= nil) and
  (head(mybuff) == ISAI_Error_report) and
  (STATE == COMMS)
 then
  RD4b_ICSL_usererror;
  null;
  mybuff := tail(mybuff);
  STATE := COMMS
 end if
[]
  ------------------------
  -- clock cycles handling
  ------------------------
  -- RTa_ICSL_okicsl_incr
  only if
   mybuff /= nil and
   head(mybuff) == icsl_tick and
   connectTimer < max_connectTimer and
   STATE == NOCOMMSconnecting
  then
   ok_icsl;
   connectTimer := connectTimer + 1 of int;
   mybuff := tail(mybuff);
   STATE := NOCOMMSconnecting
  end if
 []
  -- RTb_ICSL_okicsl_incr
  only if
   mybuff /= nil and
   head(mybuff) == icsl_tick and
   STATE == NOCOMMSwait
  then
   ok_icsl;
   null;
   mybuff := tail(mybuff);
   STATE := NOCOMMSwait
  end if
 []   -- RTc_ICSL_okicsl
  only if
   mybuff /= nil and
   head(mybuff) == icsl_tick and
   receiveTimer < max_receiveTimer and
   sendTimer < max_sendTimer and
   STATE == COMMS
  then
   ok_icsl;
   sendTimer := sendTimer + 1 of int;
   receiveTimer := receiveTimer + 1 of int;
   mybuff := tail(mybuff);
   STATE := COMMS
  end if
 end select
end loop
end var
end process -- ICSL
...
end module
```