EPTCS 399

Proceedings of the

Sixth Workshop on

Models for Formal Analysis of Real Systems

Luxembourg City, Luxembourg, 6th April 2024

Edited by: Frédéric Lang and Matthias Volk

Published: 27th March 2024
DOI: 10.4204/EPTCS.399
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of MARS 2024, the sixth workshop on Models for Formal Analysis of Real Systems, held on April 6, 2024 in Luxembourg City, Luxembourg, as part of ETAPS 2024, the European Joint Conferences on Theory and Practice of Software.

The MARS workshop series addresses the formal modelling of realistic systems. Making a formal model of a system is a necessary prerequisite for its formal analysis and for formal verification of its correctness.

To show the applicability of tools and techniques for verification and analysis, toy examples or tiny case studies are typically presented in research papers. Since the theory needs to be developed first, this approach is reasonable. However, to show that a developed approach actually scales to real systems, large case studies are essential. The development of formal models of real systems usually requires a perfect understanding of informal descriptions of the system —sometimes found in RFCs or other standard documents— which are usually just written in English. Based on the type of system, an adequate specification formalism needs to be chosen, and the informal specification needs to be translated into it. Examples for such formalisms include process and program algebra, Petri nets, variations of automata, as well as timed, stochastic and probabilistic extensions of these formalisms. Abstraction from unimportant details then yields an accurate, formal model of the real system.

The process of developing a detailed and accurate model usually takes a considerable amount of time, often months or years; without even starting a formal analysis. When publishing the results on a formal analysis in a scientific paper, details of the model usually have to be skipped due to lack of space, and often the lessons learnt from modelling are not discussed since they are not the main focus of the paper.

The MARS workshops aim at discussing exactly these unmentioned lessons. Examples are:

- Which formalism is chosen, and why?
- Which abstractions have to be made and why?
- How are important characteristics of the system modelled?
- Were there any complications while modelling the system?
- Which measures were taken to guarantee the accuracy of the model?

MARS emphasises modelling over verification. In particular, we invited papers that present full models of real systems, which may lay the basis for future comparison and analysis. The workshop thus intends to bring together researchers from different communities that all aim at verifying real systems and are developing formal models for such systems. An aim of the workshop is to present different modelling approaches and discuss pros and cons for each of them.

Full specifications of the contributed models are available online at the MARS Repository (`http://mars-workshop.org/repository.html`) —often including executable models— so that their quality can be evaluated. Alternative formal descriptions are encouraged, which should foster the development of improved specification formalisms.

The MARS 2024 workshop included talks by three invited speakers: Domenico Bianculli (University of Luxembourg, Luxembourg) presented work on runtime verification of signal-based temporal properties for cyber-physical systems; Stephan Merz (University of Lorraine, CNRS, Inria, LORIA, Nancy, France) presented work on the validation of traces of distributed programs against high-level specifications; and Bertrand Jeannet (Dassault Systèmes, France) presented work on the STIMULUS tool, dedicated to the early debugging and validation of functional real-time systems requirements.

The body of this volume contains six contributions. The submitted papers were carefully refereed by at least three members of the programme committee. The topics include:

- a model of the the Raft distributed consensus protocol and its properties in the language mCRL2,
- a comparison of formal models of the IEEE 1394 link layer written in the languages muCRL, mCRL2, and LNT,
- models of the Rijkswaterstaat tunnel control systems in mCRL2 and Dezyne,
- a comparison of an industrial (based on the PSS standard) and an academic (based on conformance testing and LNT) approaches to ensure resource isolation in hardware,
- a study of a system of controllers with redundancy modelled using the language Timed Rebecca and verified using the Afra model checker,
- optimization of the Beam Scheduling problem in robotic radiation therapy, by online model checking with the tool Uppaal.

We would like to thank the program committee members:

- Arnd Hartmanns (University of Twente, The Netherlands)
- John Hatcliff (Kansas State University, USA)
- Frédéric Lang (INRIA Grenoble Rhône-Alpes, France, co-chair)
- Lina Marsso (University of Toronto, Canada)
- Sjouke Mauw (University of Luxembourg, Luxembourg)
- Franco Mazzanti (ISTI-CNR, Italy)
- Dave Parker (University of Oxford, UK)
- Anne Remke (WWU Münster, Germany)
- Marjan Sirjani (Mälardalen University, Sweden)
- Matthias Volk (TU Eindhoven, The Netherlands, co-chair)

We are also grateful to the following reviewers:

- Sergiu Bursuc
- Zahra Moezkarimi

We wish to express our gratitude to the authors who submitted papers, the speakers, and the invited speakers. Thanks are also due to the EasyChair organisation for supporting the various tasks related to the selection of the contributions and also EPTCS and arXiv for publishing and hosting the proceedings.

Frédéric Lang and Matthias Volk

Signal-Based Temporal Properties for Cyber-Physical Systems: Specification, Monitoring, and Diagnostics

Domenico Bianculli
University of Luxembourg
domenico.bianculli@uni.lu

Run-time verification (RV) is an analysis technique that focuses on observing the execution of a system to check its expected behavior against some specification. It is used for software verification and validation activities, such as operationalizing test oracles and defining run-time monitors.

The three main components of an effective RV approach are: i) a specification language allowing users to formally express the system requirements to be checked; ii) a monitoring algorithm that checks a system execution trace against the property specifications and yields a verdict indicating whether the input traces satisfies the property being checked; iii) a diagnostics algorithm that explains the cause of a requirement violation, in case of a negative verdict.

In this talk, I will review these three aspects taking into account the perspective of signal-based temporal properties for cyber-physical systems and will report on the application of the proposed formal methods in the context of collaborative research projects with industrial partners.

F. Lang, M. Volk (Eds):
Models for Formal Analysis of Real Systems (MARS 2024)
EPTCS 399, 2024, pp. 1–1, doi:10.4204/EPTCS.399.1

Validating Traces of Distributed Programs against High-Level Specifications

Stephan Merz
University of Lorraine, CNRS, Inria, LORIA, Nancy, France
stephan.merz@inria.fr

This talk presents joint work with Horatiu Cirstea, Benjamin Loillier, and Markus Kuppe.

TLA$^+$ is widely used for describing and verifying distributed algorithms at a high level of abstraction. We present ongoing work on validating traces of distributed programs with respect to TLA$^+$ specifications. This work is supported by a library for instrumenting processes in order to log the values of variables of the TLA$^+$ specification as well as informations about the execution of events. After merging the logs of individual processes, a trace of the distributed execution is obtained, and the TLA$^+$ model checker is used to check if this trace corresponds to a prefix of an execution allowed by the TLA$^+$ specification.

Our experience with the approach has shown that although it cannot establish the correctness of an implementation, it is very effective for detecting discrepancies between executions of the distributed program and the high-level specification. Our framework requires neither the complete state of the TLA$^+$ specification nor all events to be represented in the trace because we rely on the model checker to resolve potential non-determinism, and we discuss tradeoffs between precision of tracing and complexity of model checking.

Debugging Embedded Systems Requirements with STIMULUS

Bertrand Jeannet
Dassault Systèmes
bertrand.jeannet@3ds.com

STIMULUS is an application dedicated to the early debugging and validation of functional real-time systems requirements, that has been developed in the start-up Argosim and since 2019 in Dassault Systèmes, and that addresses safety-critical embedded systems (transportation, aerospace, energy, etc.). It provides a high-level language to express textual yet formal requirements, and a solver-driven simulation engine to generate and analyze execution traces that satisfy these requirements. Visualizing what systems can do enables system architects to discover ambiguous, incorrect, missing or conflicting requirements before the design begins.

We first present the scientific foundations of STIMULUS, which is based on a constraint synchronous programming language, in which the data-flow equations of Lustre and Lucid Synchrone languages are generalized to data-flow constraints relating several signals.

We then demonstrate the use of STIMULUS on the specification of automatic headlights from the automotive industry. We show how this unique simulation technique enables to discover and to fix ambiguous and conflicting requirements, resulting in a clear and executable specification that can be shared among engineers.

Modelling the Raft Distributed Consensus Protocol in mCRL2

Parth Bora Pham Duc Minh Tim A.C. Willemse
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
p.bora@student.tue.nl
minh.pham@ximuis.eu
T.A.C.Willemse@tue.nl

The consensus problem is a fundamental problem in distributed systems. It involves a set of actors, or entities, that need to agree on some values or decisions. The Raft algorithm is a solution to the consensus problem that has gained widespread popularity as an easy-to-understand and implement alternative to Lamport's Paxos algorithm. In this paper we discuss a formalisation of the Raft algorithm and its associated correctness properties in the mCRL2 specification language.

1 Introduction

Consensus is the process of reaching an agreement on a particular issue or decision among a group of entities or individuals. In the context of distributed systems, reaching consensus is challenging, in particular because the entities are scattered across the network and need to use communication to reach agreement on decisions. Naive solutions to consensus may then lead to faulty decisions, mainly due to communication being asynchronous and potentially unreliable, or entities that may disappear and reappear. Consensus is a fundamental ingredient for guaranteeing security and reliability of, *e.g.*, blockchains and distributed ledgers.

The Paxos algorithm [8], devised by Lamport, and its variations, is one of the most well-known solutions to the consensus protocol. While the algorithm has been studied widely, it is considered to be rather involved and hard to understand and implement. Consequently, there have been many attempts to find alternative, simpler solutions to the consensus problem. The Raft algorithm [12, 11], proposed by Ongaro and Ousterhout in 2014, is one such alternative. It is generally regarded to be simpler because it breaks down the process of reaching consensus in smaller subproblems. Raft is used in, *e.g.*, *etcd*, a popular key-value store for coordinating distributed systems, facilitating service discovery, *etc*. The Raft algorithm is based on a leader-follower model, where a leader is elected among the entities to make decisions and propagate them to other entities. The other entities follow the leader's decisions and thereby reach consensus.

The Raft algorithm achieves fault tolerance using state machine replication. This is a technique for implementing a fault-tolerant service, which uses replication of servers and which coordinates the interactions between clients and server replicas. Each server hosts a state machine that generates an identical copy of a particular state [13], thus ensuring that in the event of (a limited number of) server failures, the system remains operational. Typically, state machine replication involves log replication. In the Raft algorithm, the log is simply a sequence of commands with some minimal additional information, which it keeps consistent. The logs are maintained by every server in the network and executed sequentially by these, and their uniformity guarantees that servers processes the same commands in the same order.

Given the practical significance of the consensus problem and the complexity of the solutions to the problem, found in the literature, formalising and analysing these solutions is highly relevant. The Raft algorithm has been modelled and verified in TLA+ [9] by Ongaro [1], one of the authors proposing the Raft algorithm. This specification contained a couple of minor mistakes which have been fixed, as pointed out by Evrard in [4], where an LNT [5] model of Raft is discussed. An earlier version of the LNT model has been used in the Model Checking contest [7] in 2015, where a few generic requirements were analysed. Another model of the Raft algorithm was presented in [14]. They used the Verdi framework to formally prove the State Machine Safety property, *i.e.*, the property that logs that appear in each node must provide a uniform, consistent view on the state of the servers.

In this paper, we discuss a model written in the mCRL2 language [6] and the formalisation of several properties coined in [12, 11]. The mCRL2 language is a process algebra with data; its process language is based on the algebra of communicating processes (ACP), whereas its data language is based on the theory of abstract data types. The language is supported by the mCRL2 tool set [1], which allows for generating and visualising state spaces, and which can be used to verify properties expressed in the modal μ-calculus with data. While both mCRL2 and LNT are process algebras, their syntax is quite different, and modelling in both languages requires quite a different style. We discuss the design decisions underlying our model of the Raft algorithm, and present modal μ-calculus formalisations of the properties.

Outline. We discuss our mCRL2 model of the Raft algorithm in Section 2. The mCRL2 language is introduced and explained using snippets of our model. For a full explanation of the language, we refer to [6]. In Section 3, we describe the properties that we formalised in the modal μ-calculus. We discuss some of our findings in Section 4, and end with conclusions and future work in Section 5. Full details of the model and the properties can be found in the accompanying artefact in the Mars repository[2].

2 Modelling RAFT in mCRL2

Our mCRL2 model of the Raft Algorithm focusses on the behaviour of the nodes in the network. For our models, we draw inspiration from the TLA+ [11] and LNT [4] specifications of the protocol and, like the LNT and TLA+ specifications, focus on leader election and log replication as these form the core of the protocol. Features such as cluster membership changes and log compaction have not been modelled for simplicity's sake and in the interest of keeping the state space minimal. We additionally model a communication infrastructure that facilitates reliable communication between nodes. Our network model can be modified easily to also capture unreliable communication, but this is not our initial focus. The nodes process commands that can be sent by clients; in our model, the latter is a simple process that has no other purpose than to send commands.

All actors are modelled as dedicated (parameterised) processes in mCRL2: we have `Node` processes, a `Network` process and a `Client` process. The actors run in parallel and can synchronise and exchange data by executing communicating actions. In mCRL2, this is defined by a top-level process such as:

```
init allow( {sendRPC, receiveRPC, clientCommand, advanceCommitIndex, timeout, sendRPCset ...},
            comm ( { sendClientRequest | recvClientRequest -> clientCommand,
                     sendToNetwork | receiveFromServer -> sendRPC,
                     sendToServer | receiveFromNetwork -> receiveRPC,
                     sendToNetworkSet | receiveFromServerSet -> sendRPCset },
```

[1]https://github.com/ongardie/raft.tla
[2]http://mars-workshop.org/repository.html

```
                    Client(1) || Node(...) || Node(...) || ... || HealthyNetwork(...)
                )
        );
```

Parallelism is modelled by means of the parallel operator '`||`'; which actions communicate is declared using the communication operator '*`comm`*', by specifying which pairs of action labels can engage in a communication. For instance, `sendToNetwork | receiveFromServer -> sendRPC` specifies that when a parameterised action with action label `sendToNetwork` and a parameterised action with action label `receiveFromServer` can happen simultaneously (provided their parameters match), this results in a `sendRPC` action carrying the parameters of the individual actions. By disallowing actions that are meant to communicate, synchronisation is enforced. This is achieved by means of the '*`allow`*' operator, which blocks any action other than the ones for which an action label is specified in the set of allowed action labels. For instance, by including the `sendRPC` action label, every action with an `sendToNetwork` action label is blocked and only actions with an `sendRPC` action label are allowed.

A Raft cluster may have any number of Nodes. Analysing our model using simulation (*i.e.*, stepping through the model interactively) or verification (*e.g.*, computing the validity of requirements fully automatically) to assess the correctness of (our model of) the algorithm, however, requires a fixed, concrete number of servers. Since the behaviour of the Raft algorithm crucially depends on the number of Nodes in the network, we model this number by means of a constant that all our processes can refer to. This is done as follows:

```
map NumberOfServers: Nat;
eqn NumberOfServers = 3;
```

This declares a constant `NumberOfServers` and sets it to 3; this constant should be the same as the number of `Node` processes running in parallel in the top-level process. Our model contains a few other constants which can be set similarly.

In the remainder of this section, we describe the `Client` process and the `Network` process (Section 2.1) and the `Node` process (Section 2.2).

2.1 The Raft Environment

Clients of the Raft algorithm can use it to store data and request commands that are to be executed on multiple interconnected Nodes. These Nodes operate independently and may hold different copies of the same data, with the consistency thereof being guaranteed by the Raft algorithm. For the purpose of analysing the algorithm, we introduce a simple client model: only a single client and, since we are not interested in the actual data or the commands issued by this client, we use unique ID's, modelled by natural numbers *`Nat`*, to abstract from the different messages of the client:

```
proc Client(clientCommandID: Nat) =
     (clientCommandID <= NumberOfClientRequests) ->
         sendClientRequest(clientCommandID) . Client(clientCommandID+1);
```

This defines a process `Client` that can be instantiated by passing a positive number as argument. Assuming that the constant `NumberOfClientRequests` is `3`, process `Client(1)` then executes the action `sendClientRequest(1)`, followed by `sendClientRequest(2)` and finally `sendClientRequest(3)`, after which the process is unable to perform any further actions. This behaviour is described compactly using the sequential composition operator '`.`' of mCRL2, and through the use of recursion.

We assume that communication between the client and the Raft cluster is synchronous, unlike the communication among the different nodes in a Raft cluster, which proceeds asynchronously. Raft claims to be correct even when network communication between nodes is unreliable, including delays, partitions, and packet loss, duplication, and reordering. As mentioned earlier, communication in our model of Raft happens via the communication of actions between the various `Node` processes and the `Network` process. If node *A* wants to send a message to node *B*, it sends the message to the network, which then sends it to node *B*. The network layer is introduced as an intermediary in message exchange between nodes to model message reordering. While we do not analyse the Raft algorithm in the presence of message loss or duplication, our network model can easily be modified to accommodate for these. Messages exchanged between nodes are essentially Remote Procedure Calls (RPCs). Raft utilises two distinct types of RPCs: *vote request/response* RPCs and *append entries request/response* RPCs.

```
sort RPC = struct RequestVoteRequest(currentTermRPC: Nat, endLogIndex: Nat, endLogTerm: Nat)
                    ?isRequestVoteRequest
                | RequestVoteResponse(currentTermRPC: Nat, isVoteGranted: Bool)
                    ?isRequestVoteResponse
                | ...
```

We model the messages exchanged between a node and a network using the data type `NetworkPayload`, which is a triple consisting of the ID of the sending node, a command of type `RPC` and the ID of the receiving node:

```
sort NetworkPayload = struct Message(senderID: Nat, rpc: RPC, receiverID: Nat);
```

Our network model allows a node to send a message to another nodes using a `SendToNetwork` action, which can then communicate with a `receiveFromServer` action offered by the network. Alternatively, a set of nodes can be addressed in one go, using a `sendToNetworkSet` action and which can communicate with a `receiveFromServerSet` action offered by the network. The network then takes care of dispatching the messages to these nodes using a `sendToServer` action. This is achieved by the following process:

```
proc Network(messageCollection: FSet(NetworkPayload)) =
     (# messageCollection < NetworkSize) ->
       sum msg: NetworkPayload . receiveFromServer(msg)
                               . Network(messageCollection = messageCollection + {msg})
  +
     (# messageCollection + NumberOfServers < NetworkSize + 1) ->
       sum msgs: FSet(NetworkPayload) . receiveFromServerSet(msgs)
                                      . Network(messageCollection = messageCollection + msgs)
  +
     sum msg: NetworkPayload .
       (msg in messageCollection) ->
         sendToServer(msg) . Network(messageCollection = messageCollection - {msg});
```

Informally, this process can, execute a `receiveFromServer` action carrying a (non-deterministically chosen) message of type `NetworkPayload`, as long as the network is not yet full, indicated by the condition `# messageCollection < NetworkSize`. Once the action was executed, the process again behaves as `Network`, but the parameter `messageCollection` has been updated to also contain the message `msg`. Alternatively, as indicated by the binary non-deterministic choice operator '`+`', the process may receive a message that is to be sent to all other nodes (second summand), or send a message that is in the set of messages `messageCollection`.

Note that the model depicted above models a 'perfect' network; however, using a minor adjustment, it can be turned into an unreliable network. For instance, by extending the third summand to include an option to lose the message instead of sending the message, message loss can be modelled as follows:

```
...
  +
     sum msg: NetworkPayload .
```

```
        (msg in messageCollection) ->
          (
            sendToServer(msg) . Network(messageCollection = messageCollection - {msg})
          +
            lose . Network(messageCollection = messageCollection - {msg})
           );
```

Here, `lose` is a new action indicating a message is lost, not revealing which message this is. By including this action in the set of actions that are in the *allow* set of the top-level process, the network can non-deterministically decide to drop messages.

2.2 Node

The core logic of the Raft algorithm is described by the `Node` processes. This process needs to deal with messages received from other `Node` processes, and, send messages (potentially received from a `Client` process) to other `Node` processes. Logical decisions are based on the local state of the process; this state is reflected in the parameters of the `Node` process:

```
proc Node(id: Nat, currentState: State, currentTerm: Nat, log: LogType,
          commitIndex: Nat, votedFor: Int, voterLog: FSet(Nat), nextIndex: List(Nat),
          matchIndex: List(Nat), replyToBeSent: replyHelper) =
     currentState != Crashed ->
     ( (IsNone(replyToBeSent)) ->
         Node_process_receiveFromNetwork(id, currentState, currentTerm, log,
                                         commitIndex, votedFor, voterLog, nextIndex,
                                         matchIndex, replyToBeSent)
       +
       Node_process_sendToNetwork(id, currentState, currentTerm, log, commitIndex,
                                  votedFor, voterLog, nextIndex, matchIndex, replyToBeSent)
       +
       (currentState != Leader  && currentTerm < MaxTerm) ->
         timeout . Node(currentState = Candidate, currentTerm = currentTerm + 1,
                        votedFor = id, voterLog = {id}, replyToBeSent = none)
    )
    +
    currentState == Crashed -> ...;
```

As can be seen, for the sake of readability we have split part of the `Node` process in two subprocesses, *viz.*, the process `Node_process_receiveFromNetwork` and `Node_process_sendToNetwork`. So long as the node has not crashed, it offers a non-deterministic choice between the behaviour described by these two processes and (conditionally) timing out (as described by the third summand). Subprocess `Node_process_receiveFromNetwork` handles all messages the node receives through the network, whereas `Node_process_sendToNetwork` takes care of sending messages, received from the client, or replies to previous messages, to other nodes. If the node has crashed, a recovery mechanism can be initiated (not depicted here).

The Raft algorithm divides time into terms of arbitrary length. The current term number is represented by the `currentTerm` parameter of type *Nat* of the `Node` process. Each node in a Raft cluster can be in one of three states: *Leader*, *Follower*, or *Candidate*; see also Fig. 1. This state is maintained by parameter `currentState` in process `Node`. In addition to these three possible states, we have introduced a fourth state to indicate that the node has crashed. The data type `State` thus is as follows:

```
sort State = struct Leader | Candidate | Follower | Crashed;
```

The way Raft (and algorithms like Raft) implements replicated state machines is by means of a replicated log. Each node in the Raft cluster stores a log consisting of entries that contain a state machine command and the term number that indicates when the entry was received by the *Leader*. In our log entries, the

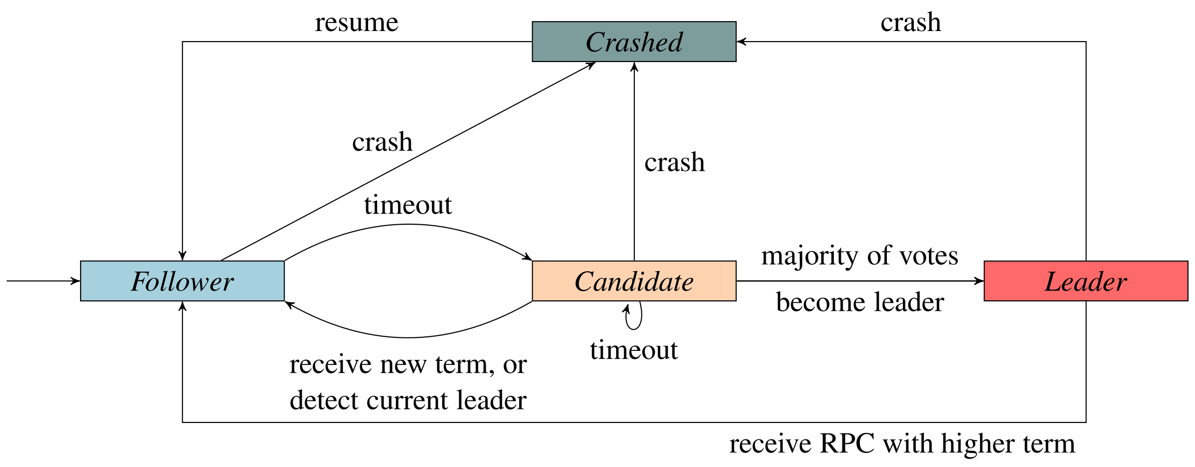


Figure 1: State transitions for a node in Raft.

state machine commands are represented by the command ID, since we are not interested in the actual command itself. The parameter `log` of process `Node` is thus basically a list of entries that contain a command ID and term:

```
sort logEntry = struct Command(term: Nat, commandID: Nat);
sort LogType  = List(logEntry);
```

A practical complication with our formalisation is that mCRL2 lists are zero-indexed, unlike the logs described in the original Raft paper [12, 11], which are one-indexed. While converting from one representation to the other is straightforward, it is equally easy to make mistakes. We have circumvented this by utilising helper functions that make the conversion less error-prone.

The Raft State Machine. All nodes start out as *Followers*. Depending on the state of the node, certain actions are permitted. Only when a node is in state *Leader*, it can start accepting messages from a client. A node that is *Leader* sends periodic heartbeats to followers to assert its presence. During an election, nodes that are *Candidates* send *vote request* RPCs to other nodes to garner votes. Subprocess `Node_process_sendToNetwork` takes care of these events. If a *Follower* does not receive either of these messages over a period of time, called the election timeout and modelled by means of the `timeout` action, it starts a new election by changing into a *Candidate* state and increments its term. Subsequently, it will vote for itself. We remark that we here closely follow the LNT model, allowing the *Candidate* to vote for itself rather than by sending a *vote request* RPC to itself, and by modelling the timeout by means of non-determinism rather than by imposing hard real-time requirements. Safety requirements should not be affected by modelling timeouts using non-determinism. However, due to this abstraction, we cannot analyse real-time requirements, nor the real-time performance of the algorithm. Also, when phrasing liveness requirements, the abstraction may require one to be explicit about the absence or occurrence of these timeouts.

After a node becomes a *Candidate*, it sends a *vote request* RPC to all other servers in the cluster. In our model, this is achieved using a `sendToNetworkSet` action, carrying a set of messages consisting of the RPC and a target node as its parameter; the `Network` process then relays the request to all targeted nodes. The set of messages is created using a recursive function `CreateRequestVoteSet` that builds the set by iterating over all possible node IDs that have not voted for the *Candidate* node yet; the latter is specified in an auxiliary function `CreateRequestVoteSetHelper`:

```
map CreateRequestVoteSet: Nat # Nat # Nat # Nat # FSet(Nat) -> FSet(NetworkPayload);
var sender, termNode, lengthLog, lastTermLog: Nat;
    voterLog: FSet(Nat);
eqn CreateRequestVoteSet(sender, termNode, lengthLog, lastTermLog, voterLog)
  =
    CreateRequestVoteSetHelper(sender, RequestVoteRequest(termNode, lengthLog, lastTermLog),
                               voterLog, 0);

map CreateRequestVoteSetHelper: Nat # RPC # FSet(Nat) # Nat -> FSet(NetworkPayload);
var sender, receiver: Nat;
    rvr: RPC;
    voterLog: FSet(Nat);
eqn (receiver==NumberOfServers) ->
      CreateRequestVoteSetHelper(sender, rvr, voterLog, receiver) = {};
    (receiver<NumberOfServers ) ->
      CreateRequestVoteSetHelper(sender, rvr, voterLog, receiver) =
        CreateRequestVoteSetHelper(sender, rvr, voterLog, receiver + 1 )
      + if(receiver!=sender && !(receiver in voterLog), {Message(sender, rvr, receiver)},{});
```

If a node receives a stale message, *i.e.*, a message with a term smaller than `currentTerm`, it immediately discards it. When it receives a message with a term greater than `currentTerm`, the node steps down to the *Follower* state and resets the `votedFor` parameter to `-1`, to indicate it has not voted for anyone in that term, and it sends a reply. The type of message received determines the type of reply sent by the node. This reply is then stored in the `replyToBeSent` parameter so that it can be sent out before the node engages in other interactions but potentially only after the node has updated its state. This allows for analysing the effects (in any) of nodes crashing random moments. In particular, when nodes crash, part of their state information is saved and restored, and, hence, the order of events might matter.

When a server receives a *vote request* RPC from a *Candidate*, it votes for them if it has not yet voted for any other node in that term previously. Additionally, to prevent a *Candidate* with an out-of-date log from becoming *Leader*, the node compares the index and term of the last entries in the logs of the voter and the *Candidate*. The Raft algorithm uses an ingeneous scheme—taking the type of RPC, the current term of the node and the message and the log of the *Candidate* into account—to decide whether the vote is granted to the *Candidate* or not; it then informs the *Candidate* of its decision. On receiving a reply from the node, the *Candidate* evaluates the number of votes it has received. If it successfully acquires votes from a majority of the nodes in the cluster, it becomes a *Leader* by changing to state `Leader`. While waiting for votes, if the *Candidate* receives a valid heartbeat from a *Leader*, it steps down to become a *Follower*. In case of a split vote, the *Candidate* can timeout again and start a new election.

Log Replication. When a *Leader* receives a request from the client, the request is appended to the log and all other nodes are informed using *append entries* request RPCs. In our model, this is achieved in a way that is similar to how *Candidate* nodes deal with *vote request* RPCs. A *Leader* sends only one log entry at a time; this is in line with the TLA+ specification, although the Raft algorithm supports sending multiple log entries at once.

In an *append entries* request RPC, the *Leader* includes the index and term of the log entry immediately preceding the new entries. If a *Follower* does not find a matching entry in its log with the same index and term, then it refuses the entry and sends back a negative response. The *Leader*, upon receiving a negative response, decrements the *Follower*'s `nextIndex`, which is a list where each index corresponds to the same `serverID` and which stores the index of the log entry the *Leader* will send to that node. The *Leader*, when first elected, initialises all `nextIndex` values to the index just after the last one in its log. After decrementing the `nextIndex`, the *append entries* RPC is retried. Eventually `nextIndex` will reach the point where the *Leader* and *Follower* logs match. When this happens, any subsequent conflicting entries in the *Follower*'s log are removed and entries from the *Leader*'s log are appended (if any). Consequently, a positive response is sent back to the *Leader* and log replication is successful.

Once the leader has sucessfully replicated a log entry on majority of the servers, the entry is deemed committed. We use the action `advanceCommitIndex` to model this. Moreover, we have introduced a function `MaxAgreeIndex` to find the highest possible index that can be committed. Once an entry has been committed, the *Leader* applies it to its state machine. The *Leader* keeps track of the highest index it knows to be committed, in parameter `commitIndex`, and includes this in *append entries* RPCs (heartbeat messages included) so other nodes can commit the entries, too. This method of counting successful replication on a majority is not used to commit entries from previous terms: only log entries from the *Leader*'s current term are committed by counting replicas. Once an entry from the current term has been committed in this way, then all prior entries are committed indirectly. The function `isAdvanceCommitIndexOk` is used to keep this in check.

Model Statistics. We have generated state spaces of various instances of the model as described (see also the Mars repository for the full model). The base case is a configuration in which there are 3 nodes, 2 commands from clients, 1 term, a network capacity of 3 messages, and no crashes and recovery of nodes. This basic configuration already leads to a rather large state space of over 200k states, which can be generated in slightly under a minute on a 2017 Macbook Pro. We do note that there is some redundancy in the model, since strong bisimilarity reduction manages to compress the state space with almost a factor of 5. Table 1 shows the statistics of all configurations we explored, including a configuration that shows the effect of using a lossy network. To give a rough indication of the time required to generate these state spaces, we have included the time it takes for a 2017 Macbook Pro with 16GB memory to generate these state spaces. This clearly shows the dramatic effect of nodes crashing and of increasing the number of possible terms.

#Nodes	#Commands	#Terms	#Network Capacity	Lossy Network	Crashing	Size	Time
3	2	1	3	no	no	$2.14\,10^5$	$\sim$ 1 min.
3	1	2	3	no	no	$1.17\,10^6$	$\sim$ 2 min.
3	1	3	3	no	no	$1.32\,10^7$	$\sim$ 13 min.
3	2	1	3	no	yes	$1.79\,10^7$	$\sim$ 5 min.
3	2	2	3	no	no	$2.25\,10^7$	$\sim$ 19 min.
3	2	1	3	yes	yes	$5.97\,10^7$	$\sim$ 105 min.
3	1	2	3	no	yes	$1.48\,10^8$	$\sim$ 24 min.
3	1	3	3	no	yes	$2.38\,10^9$	$\sim$ 820 min.†

Table 1: Some statistics for the state space sizes for various configurations of the Raft algorithm; (†) this experiment was conducted on a compute server so the runtime is not directly comparable to the other runtimes reported.

3 Raft Properties

The Raft algorithm is quite involved, and it is easy to make small mistakes when formalising the algorithm. A simple example of the subtleties include the aforementioned difference between the zero-indexed lists of mCRL2 and the one-indexed arrays used in the original description of the algorithm. As part of the original description of the algorithm, the authors also list several properties that the algorithm guarantees; such properties can be seen as partial specifications of the algorithm. We have taken some of these properties and formalised them as modal μ-calculus formulas. These formulas have been used throughout the model development to hunt for bugs in our formalisation, and provide an extra layer of validation in addition to manual simulation of the model, increasing our confidence in the model.

One central complication in formalising the original properties in the modal μ-calculus is the fact that the properties refer to the variables that span the state of each node; in our case, those are, for instance, the parameters of the `Node` process. Since the mCRL2 language is action-based rather than state-based, these parameters cannot be referred to in the modal μ-calculus formulas. We have sidestepped this issue by extending our model with auxiliary actions that expose the relevant information. For instance, for the purpose of verification, we have introduced self-loops labelled by actions such as `exposeLeader`, `exposeLogLeader` and `exposeLog`.

In what follows, we will briefly discuss four main properties that have been formalised and their modal μ-calculus formalisation next to it. The *Append Entries* property that is also mentioned in [12, 11] is omitted since it follows immediately from the operations a *Leader* can carry out in our model. All formulas happen to fall in a category of formulas that can be represented in a PDL-style language, called *regular formulae*, that abstracts from the fixed points one typically expects in modal μ-calculus formulas. We will explain the meaning of the formulas as we proceed. All main properties hold true for all configurations of Table 1; verification of each property takes roughly 2-3 times that of generating the state space using the symbolic technique described in [10]. We additionally verify a number of simple liveness properties that demonstrate that the main properties we verify do not hold true vacuously. Also these can be expressed in terms of regular formulae, save one.

Election Safety. One of the fundamental properties on which the Raft algorithm relies for its correct functioning is the property that at most one *Leader* can be elected in a term. This is called the *Election Safety* property in [12, 11]. Since the state of each node cannot be read directly, we use the `exposeLeader` action to expose that the state of a node is `Leader`. Then, the correctness property can be phrased as the inability for two distinct nodes to execute `exposeLeader` actions in the same term. The formula we use to express this is as follows:

```
[true*] forall id1, termx: Nat . [exposeLeader(id1, termx)]
  [true*] forall id2: Nat . val(id1!=id2) => [exposeLeader(id2, termx)] false
```

This formula should be read as follows. Invariantly (captured by the first occurrence of `[true*]`), executing any `exposeLeader` action, carrying some ID (here represented by `id1`) of a node and a term, will lead to a state from which invariantly (captured by the second `[true*]` formula) it is impossible to execute another `exposeLeader` action carrying an ID (represented by `id2`) of a different node and the same term. The latter part is captured by the `[exposeLeader(id2, termx)]false` subformula.

Note that this property can hold true trivially if no *Leader* is ever elected. To exclude this scenario, we have additionally phrased a liveness property that verifies that such actions can indeed take place:

```
<true*> exists id1, termx:Nat . <exposeLeader(id1, termx)>true
```

This property, which should be read as follows: following zero or more actions, it is possible that an `exposeLeader` action can be executed, carrying some ID (represented by `id1`) and some term (represented by `termx`). Also this property holds true for all the configurations of Table 1. To assess whether there is a scenario in which two different nodes can be a leader, we can check the following formula:

```
<true*> exists id1, termx:Nat . <exposeLeader(id1, termx)>
  <true*> exists id2, termy:Nat. val(id1 != id2) && <exposeLeader(id2,termy)>true
```

This formula only holds true in configurations in which there are at least two terms, as can be expected. Taking this into account, we can verify a formula that states that there is a sequence of events in which we see `MaxTerm` times a different leader announce itself. Such a formula requires us to explicitly use a least fixed point and keep track of the number of times we have witnessed the `exposeLeader` action and the ID of the leader that announced itself most recently.

```
mu X(id:Nat = 1, n:Nat = 0).
  (  val(n >= MaxTerm) || <true>X(id,n)
  || exists id2, termy:Nat. (val(id != id2) && <exposeLeader(id2,termy)>X(id2,n+1))
  )
```

This formula holds true in all configurations.

Log Matching. The log replication mechanism ensures that each node in the Raft cluster has the same view on the state of the cluster. In particular, if, for two nodes, their logs contain an entry with the same index and term, then these logs are identical in all entries up to (and including) the given index. This is called the *Log Matching* property in [12, 11]. Log information of nodes, which is tracked in the `log` parameter of the `Node` process cannot be inspected using the modal μ-calculus formula without exposing the information through actions. This is achieved by extending the model with self loops of `exposeLog` actions; the property can then be formalised as follows:

```
[true*] forall id1, term1, commitIndex1: Nat, log1: LogType .
  val(log1!=[]) => [exposeLog(id1, term1, commitIndex1, log1)]
    [true*] forall id2, term2, index, commitIndex2: Nat, log2: LogType .
      val(index<#log1 && index<#log2 && id1!=id2 && log2!=[] && log1.index == log2.index)
      =>
      ([exposeLog(id2, term2, commitIndex2, log2)]
        val(slice(log1, 1, index+1) == slice(log2, 1, index+1)))
```

Again, the `[true*]` should be read as ‘invariantly’. The formula can then be understood to state the following: invariantly, for any state in which there is a node (the ID of which is `id1`) with a non-empty log `log1` it is the case that invariantly from that moment onwards, any other state in which there is another node (the ID of which is `id2`) with a non-empty log `log2` that has an entry at position `index` in common, the slices of `log1` and `log2` coincide up to, and including position `index`.

Leader Completeness. Another aspect of the log replication mechanism is that log entries, committed in a given term, will persist in the logs of the *Leaders* in future terms; in [12, 11] this is referred to as the *Leader Completeness* property. This ensures that the logs are indeed a proper reflection of what has happened in the past. We modify the model to include `exposeLogLeader` self-loops that expose the log information of the node that is currently in state `Leader`. The `advanceCommitIndex` actions, already present in the model, are used as signals that information has been committed in the log up to, and including entry `currentCommitIndex`. Note that in our model, the `advanceCommitIndex` action also exposes the log of the leader through the `log1` parameter.

```
[true*] forall currentCommitIndex, nextCommitIndex, term1: Nat,  log1: LogType .
  [advanceCommitIndex(currentCommitIndex, nextCommitIndex, term1, log1)]
    [true*] forall term2, index: Nat, log2: LogType .
      val(term2>term1 && index>currentCommitIndex && index<=nextCommitIndex)
      =>
      [exposeLogLeader(term2, log2)] val((log1 . index) in log2)
```

This formula should be read as follows: invariantly, whenever a `advanceCommitIndex` action happens, exposing the current commit index `currentCommitIndex`, the next commit index `nextCommitIndex`, the term `term1` and the *Leader*'s log `log1`, then whenever we subsequently inspect the log of a *Leader* in a future term `term2`, then those log entries in `log1` that can be found at indices beyond `currentCommitIndex` and `nextCommitIndex` are contained in the log entries of `log2`.

State Machine Safety. The logs that appear in each node furthermore must provide a uniform, consistent view on the state of the cluster. That means that after a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index. This property is referred to as the *Sate Machine Safety* property in [12, 11]. In order to express this property, we again assume that the model has been extended with self-loops labelled with `exposeLog` actions. The property can then be formalised as follows:

```
[true*] forall id1, term1, commitIndex1: Nat, log1: LogType .
  val(commitIndex1 > 0)
  =>
  [exposeLog(id1, term1, commitIndex1, log1)]
    [true*] forall id2, term2, commitIndex2: Nat, log2: LogType .
      val(id1!=id2 && commitIndex2>=commitIndex1)
      =>
      [exposeLog(id2, term2, commitIndex2, log2)]
        val(slice(log1, 1, commitIndex1) == slice(log2, 1, commitIndex1))
```

This formula can be understood as follows: invariantly, whatever the log of a node is, given the commit index `commitIndex1` of the node at that time, the log will overlap up-to and including this index in any future moment in which the commit index `commitIndex2` of a node is equal or larger. This ensures consistency of the logs over time, meaning that the same entries have been applied to the state machine. Note that the condition `commitIndex1 > 0` ensures that an entry has been committed.

In order to assess whether or not the property holds true vacuously, we have phrased the following simple liveness requirement:

```
<true*> exists id1, term1, commitIndex1: Nat, log1: LogType .
  val(commitIndex1 > 0) && <exposeLog(id1, term1, commitIndex1, log1)>
    <true*> exists id2, term2, commitIndex2: Nat, log2: LogType .
      val(id1 != id2 && commitIndex2 >= commitIndex1)
      &&
      <exposeLog(id2, term2, commitIndex2, log2)> true
```

This property holds true for every model we have analysed.

4 Discussion

We briefly touch on a few observations related to modelling in mCRL2, but also related to how our model compares to existing formalisations of the Raft algorithm.

Modelling in mCRL2. The mCRL2 language has all the features that allow one to concisely describe the workings of complex distributed algorithms. Parallelism and message passing, both key ingredients in the Raft algorithm, are key concepts that allow the model to stay close to reality. Furthermore, parameterisation of processes and actions allows for reusing parts of the specifications, avoiding copy-paste mistakes and improving readability. Finally, the rich and expressive data language of mCRL2 is essential when describing the more complex operations on arrays that are part of the Raft algorithm. Built-in types such as lists, natural numbers, and sets, and the facility to specify custom data types and operations on these turned out to be essential for keeping the specification readable and its size to a minimum.

Modelling in the mCRL2 language does require experience, and there is not really a practical guidebook that explains how to use the language effectively. This can lead to sub-optimal ways of modelling. For instance, the initial model, which was created by the first two authors and who had no prior experience of using mCRL2, used a modelling style that is perfectly valid but that led to state spaces that were orders of magnitude larger than needed. To illustrate, consider the following mCRL2 specification:

```
act a:Bool;
proc X(c:Bool) = sum b:Bool. a(b). ( (b -> X(true) + !b -> X(false)));
init X(true);
```

The transition system that is generated for this specification has 3 states and 6 transitions. Based on the specification, one would expect at most 2 states: one representing `X(true)` and one representing `X(false)`. The third state is introduced by the pre-processing of the mCRL2 toolset, which rewrites the above specification to normal form and which may introduce extra process parameters. Now consider the following mCRL2 specification:

```
act a:Bool;
proc X(c:Bool) = sum b:Bool. ( b -> a(b).X(true) + !b -> a(b).X(false));
init X(true);
```

The state space generated for this specification has only 1 state and 2 transitions. It is strongly bisimilar to the state space of the previous example, and therefore, for all intents and purposes, equivalent to it. In this case, the pre-processing conducted by the mCRL2 toolset does not lead to an additional state because the process is already in a shape it wishes to produce. Even better, during the pre-processing it detects that parameter `c` of process `X` is irrelevant, which can easily be seen because it does not appear in the right-hand side.

Comparison to Other Formalisations. As we have indicated earlier, in constructing the mCRL2 model for the Raft algorithm, we have drawn inspiration from both the TLA+ and the LNT specifications. However, there are cases where the TLA+ and LNT specifications make different modelling choices, and, consequently, we have had to make a choice between the two. A case in point is the way the TLA+ specification deals with stale RPC messages: it drops stale responses but stale requests are replied to, to alert the sending party of the newer term. The LNT specification, on the other hand, discards stale requests as well. In our model, we chose to here follow the LNT specification. An example where we followed the TLA+ specification is where we send the minimum between `commitIndex` of the current node and the `nextIndex` of the receiver when sending the *append entries* request, rather than LNT's choice to send the `commitIndex` of the *Leader*. There are other places where our model deviates subtly from the LNT model, for instance in dealing with crashed nodes. In particular, the LNT model does not appear to allow for nodes to reboot, and the inclusion of rebooting nodes has had some implications on how we dealt with sending replies to requests.

Concerning the TLA+, LNT and mCRL2 modelling languages, we remark that due to LNT having many traits of an imperative language, unlike mCRL2 and TLA+, the LNT specification is in all likelihood more appealing to the average software engineer than the TLA+ or mCRL2 specifications. Also, the ability to specify crashing of nodes using LNT's disrupt operator is rather elegant; in our mCRL2 model, this requires hard-coding the option to crash. While in our model, the difference turns out to be minimal, this would not have been the case if our model had used large numbers of actions that could be executed sequentially. For instance, specifying that a `crash` action can interrupt the process `a.b.c` would require a specification of the form `crash + a. (crash + b. (crash + c))`.

5 Conclusions and Future Work

In this paper, we have highlighted and discussed several parts of our mCRL2 model of the Raft algorithm and the modal μ-calculus formulas capturing its properties. The full details of the model and the formulas can be found in the Mars repository. While our model shares many aspects with the TLA+ and LNT specification that have been published before, the formalisation of some of the key properties of the algorithm using a modal logic appear to be new. Note that only the simplest configurations can be verified in reasonable time, but it may still be interesting to verify the more complex configurations as well, including non-perfect network behaviour. We consider this part of future work.

Furthermore, it would be interesting to verify stronger liveness requirements. We have only covered a few very basic, weak liveness properties, asserting that it is possible to, *e.g.*, (repeatedly) become a leader. Stronger liveness requirements, asserting that always inevitably a leader *must* be elected, are simply not true in our model because in some configurations, messages are lost or nodes crash, but also due to us imposing limits on the maximum number of terms we consider. Phrasing the exact properties while taking all exceptions into account is non-trivial: for some properties, a counterexample may not simply be a run of the system but it can consist of an entire subgraph of the transition system [2, 3], consisting of a 1 000 or more states. In such cases, understanding the root cause of the violation can be virtually impossible. Proving liveness properties for the unrestricted model (*i.e.*, when not limiting the number of terms) can be even more challenging.

Furthermore, in the model, when a node receives a message, it computes the reply atomically. This simplifies the model but does not accurately reflect real-world scenarios where the computation of a reply would involve multiple steps and could be interrupted by other events. Refining these aspects would increase the applicability of the model to real-life scenarios but a careful tradeoff must be made between the level of abstraction and the granularity of the model to keep the state space from exploding.

Finally, like the TLA+ and LNT specifications, our model lacks real-time, even though the algorithm suggests typical timing intervals. For instance, Raft chooses election timeouts arbitrarily from a fixed interval (*e.g.*, 150–300ms), whereas in our model a timeout can happen non-deterministically. While mCRL2 has facilities to model real-time aspects, the current status of the tooling is not sufficiently powerful to deal with real-time systems with state spaces of this size. A real-time extension of our model could therefore serve as a challenging benchmark for real-time model checking tools.

Acknowledgements. We would like to thank Myrthe Spronck (TU/e) for discussions on the problem of consensus and the solution provided by Paxos.

References

[1] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, *Lecture Notes in Computer Science* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[2] Sjoerd Cranen, Bas Luttik & Tim A. C. Willemse (2013): *Proof Graphs for Parameterised Boolean Equation Systems*. In Pedro R. D'Argenio & Hernán C. Melgratti, editors: *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*,

Lecture Notes in Computer Science 8052, Springer, pp. 470–484, doi:`10.1007/978-3-642-40184-8_33`.

[3] Sjoerd Cranen, Bas Luttik & Tim A. C. Willemse (2015): *Evidence for Fixpoint Logic*. In Stephan Kreutzer, editor: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, *LIPIcs* 41, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 78–93, doi:`10.4230/LIPICS.CSL.2015.78`.

[4] Hugues Evrard (2020): *Modeling the Raft Distributed Consensus Protocol in LNT*. In Ansgar Fehnker & Hubert Garavel, editors: *Proceedings of the 4th Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2020, Dublin, Ireland, April 26, 2020*, *EPTCS* 316, pp. 15–39, doi:`10.4204/EPTCS.316.2`.

[5] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *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`.

[6] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. MIT Press, doi:`10.5555/2628007`.

[7] Fabrice Kordon, Hubert Garavel, Lom-Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, César Rodríguez & Francis Hulin-Hubard (2016): *MCC'2015 - The Fifth Model Checking Contest*. *Trans. Petri Nets Other Model. Concurr.* 11, pp. 262–273, doi:`10.1007/978-3-662-53401-4_12`.

[8] Leslie Lamport (1998): *The Part-Time Parliament*. *ACM Trans. Comput. Syst.* 16(2), pp. 133–169, doi:`10.1145/279227.279229`.

[9] Leslie Lamport (2002): *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, doi:`10.5555/579617`. Available at `http://research.microsoft.com/users/lamport/tla/book.html`.

[10] Maurice Laveaux, Wieger Wesselink & Tim A. C. Willemse (2022): *On-The-Fly Solving for Symbolic Parity Games*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, *Lecture Notes in Computer Science* 13244, Springer, pp. 137–155, doi:`10.1007/978-3-030-99527-0_8`.

[11] Diego Ongaro (2014): *Consensus: bridging theory and practice*. Ph.D. thesis, Stanford University, USA, doi:`10.5555/AAI28121474`.

[12] Diego Ongaro & John K. Ousterhout (2014): *In Search of an Understandable Consensus Algorithm*. In Garth Gibson & Nickolai Zeldovich, editors: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, USENIX Association, pp. 305–319, doi:`10.5555/2643634.2643666`.

[13] Fred B. Schneider (1990): *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. *ACM Comput. Surv.* 22(4), pp. 299–319, doi:`10.1145/98163.98167`.

[14] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst & Thomas Anderson (2016): *Planning for change in a formal verification of the raft consensus protocol*. *CPP 2016 - Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, co-located with POPL 2016*, pp. 154–165, doi:`10.1145/2854065.2854081`.

Four Formal Models of IEEE 1394 Link Layer

Hubert Garavel
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
hubert.garavel@inria.fr

Bas Luttik
Eindhoven University of Technology, The Netherlands
s.p.luttik@tue.nl

We revisit the IEEE 1394 high-performance serial bus ("FireWire"), which became a success story in formal methods after three PhD students, by using process algebra and model checking, detected a deadlock error in this IEEE standard. We present four formal models for the asynchronous mode of the Link Layer of IEEE 1394: the original model in μCRL, a simplified model in mCRL2, a revised model in LOTOS, and a novel model in LNT.

1 Introduction

IEEE 1394 (also called "FireWire") is an interface standard that specifies a serial bus architecture for high-speed communications. It can connect up to 63 peripherals in a tree or daisy-chain topology, and can perform both asynchronous and isochronous transfers simultaneously. It was developed between 1986 and 1995 by a large consortium gathering Apple, Panasonic, Philips, Sony, and many others contributors. This work resulted in an IEEE standard [43], followed by integration in many industrial products.

In the framework of the COST-247 action [22], a pan-European academic collaboration that took place between 1994 and 1997, the asynchronous mode of the link layer protocol of IEEE 1394 was selected as an interesting case study for formal methods. This protocol, which was close to being standardized, was thus studied by several young scientists at this time. At CWI Amsterdam, Bas Luttik developed a formal model [26, 27] in the μCRL language [19, 15] and stated five correctness properties that the protocol should satisfy. At INRIA Grenoble, Mihaela Sighireanu translated this model to LOTOS [20] and, using the XTL model checker [29] with the help of Radu Mateescu, discovered that the deadlock-freeness property did not hold, i.e., that the protocol could enter a deadlock state after a specific sequence of 50 transitions [37, 38, 39]. A detailed report about this bug, which would have been difficult to detect using step-by-step simulation or testing, can be found in [41]. The link layer protocol was also studied using theorem proving at the Universities of Kiel and Eindhoven by Lars Kühne, Jozef Hooman, and Willem-Paul de Roever [23].

Although the IEEE 1934 serial bus is no longer used today (deployed in Apple products from 1999 to 2016, it has been gradually replaced by USB 2, USB 3, and Gigabit Ethernet), it is an inspiring example for the formal methods community. From a historical perspective, it is a striking success story where three doctoral students discovered in a few weeks an unexpected deadlock in an IEEE standard designed and scrutinized over ten years by one hundred experts. Also, numerous research papers have been devoted to another aspect of IEEE 1934, its leader election algorithm ("root contention protocol"), the verification of which involves parameters, probabilities, and real time [35, 30, 33, 47, 7, 28, 48, 31, 34, 42, 4, 2, 24, 25, 32, 46, 49, 5].

Concerning the link layer protocol, formal methods evolved since 1997, as the μCRL and LOTOS languages have been replaced by newer languages, respectively mCRL2 [17, 16, 18, 1] and LNT [13, 10, 12, 36, 3], a descendent of the E-LOTOS standard [21]. Therefore, twenty-five years after, we revisit this case study to present, along with the original μCRL model, three companion models: a model written in mCRL2 by Jan Friso Groote, a recent revision of the LOTOS model developed by M. Sighireanu, and a novel model written in LNT.

The present article is organized as follows. Section 2 gives an overview of the IEEE 1394 architecture and explains the behaviour of the Link layer and neighbour layers. Section 3 presents four formal models in μCRL, mCRL2, LOTOS, and LNT, and discusses their main features from a modelling point of view — the models themselves being fully provided in Annexes A to D. Section 4 briefly reports about the verification (model checking and equivalence checking) done on these models. Finally, Section 5 gives a few concluding remarks.

2 IEEE 1394 bus

In this section, we present a description of IEEE 1394 that bridges the gap between the general description given in the IEEE standard [43] and the four formal models provided in the present article. The text in this section is based upon the technical report [26] in which the μCRL model first appeared — actually, this model was developed from a draft version [44] of the IEEE standard, but we believe that there is no significant difference between the draft and the standard in this respect.

First, we present the architecture as defined in the standard. Then, we focus our attention on the link layer of the protocol, the behaviour of which is our primary modelling purpose. To provide a comprehensive description of the link layer interacting with its environment, we will need to include the external functional behaviour of the physical layer, and so that is described too.

2.1 Architecture

The IEEE 1394 standard deals both with the physical requirements and the protocol of the bus. The main feature of the standard is that it supports two modes of transaction: an *asynchronous mode* and an *isochronous mode*.

In asynchronous mode, one party (the sender) can send a message of arbitrary length to some other party (the receiver). Such a message may be sent at an arbitrary moment after the sender has gained access to the bus; the only timing restriction is that the interval during which a node may have access to the bus is bounded. In this mode, the receiver must confirm the receipt of the message by sending an acknowledgement.

In isochronous mode the sender is obliged to send messages at fixed rates, and messages are not acknowledged. This service is useful for fast transmission of large amounts of data (e.g., audio/video streams), if certainty at the side of the sender about the receipt of the data by the receiver is not important, whereas the arrival of the data at a constant rate is.

The IEEE 1394 serial bus architecture is roughly as depicted in Figure 1. It consists of a number of nodes (addressable entities that run their own part of the protocol) connected by a serial cable.

The protocol describing the behaviour of a node in asynchronous mode distinguishes three layers:

1. The *transaction layer* (the upper layer, indicated by TRANS in Figure 1) offers three types of transactions to the application(s) running on the node: *read transactions* (read data from another node), *write transactions* (write data to another node), and *lock transactions* (have some of its own

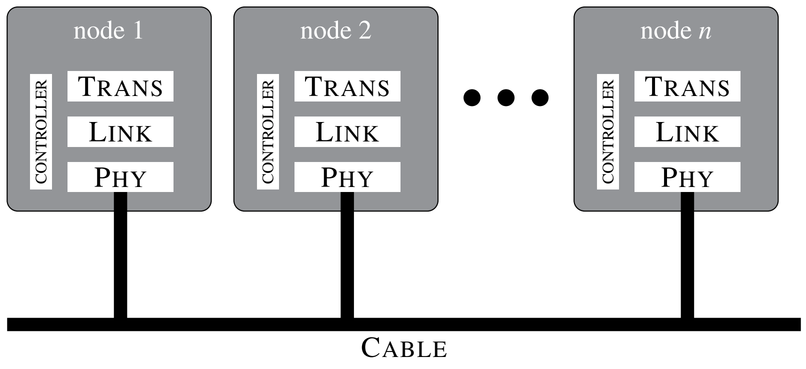


Figure 1: IEEE 1394 architecture

data processed by another node after which it is transferred back). Such transactions consist of a request and a response; the transaction layer can both handle *concatenated response* transactions (response follows request immediately) and *split* transactions (response not necessarily follows immediately on the request it belongs to).

2. The *link layer* (the middle layer, indicated by LINK in Figure 1) forms the interface between the transaction layer and the physical components of the bus (consisting of the physical layers, which are connected to each other by a serial cable). The link layer provides two types of services to the transaction layer:

Data request/response: By means of a LINK *data request*, the transaction layer instructs the link layer to send a packet to some particular node or to broadcast a packet to all other nodes. The transaction layer must react on a packet addressed to it by sending an acknowledge packet by means of a LINK *data response*.

Data indication/confirmation: By means of a LINK *data indication*, the link layer indicates the arrival of data (either request or response data). The receipt of an acknowledge packet is indicated to the transaction layer by means of a LINK *data confirmation*.

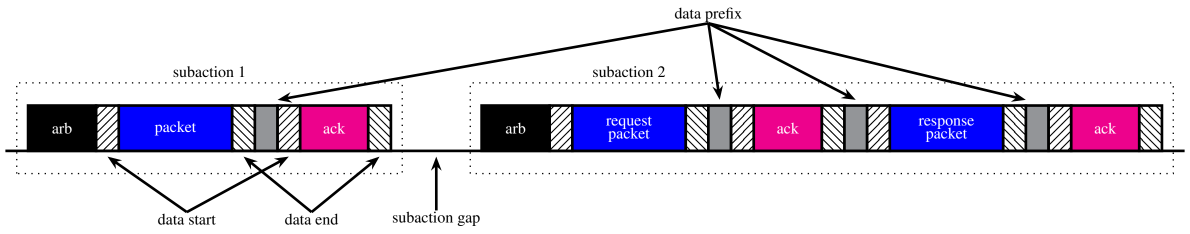


Figure 2: Subactions

The link layer divides the stream of data that it receives from the physical layer into an alternating sequence of *subactions* and *subaction gaps*, the latter being time intervals with a specified minimal length during which serial cable resides in an idle state (see Figure 2). A subaction either consists of a single packet (in case of a *split transaction*, see subaction 1) or of two packets (in case of a *concatenated response transaction*, see subaction 2). Within each subaction, a packet is delimited by special *data start* and *data end* signals[1]; the gap between two packets within a subaction must be filled with *data prefix* signals in order to distinguish these gaps from the subaction gaps.

Before a packet can be sent, the link layer must first gain access by issuing an arbitration procedure. Moreover, the link layer must transform the requests of the transaction layer into a certain packet format, computing and attaching checksums to parts of the data to be transmitted. It also decides whether incoming packets have been received properly by verifying the attached checksums. Every packet that is sent by any of the nodes is received by the link layer of every node. If a link layer determines that the packet was indeed addressed to the node it is part of, then it forwards the contents of the packet to the transaction layer. The link layer also handles the sending and receiving of acknowledgements.

3. The physical connection between a node and the serial line is called the physical layer (the lower layer, indicated by PHY in Figure 1). It listens to and puts signals on the serial cable, measures the lengths of the time intervals during which the cable resides in an idle state, and determines, together with the other physical layers, which node has control over the cable (arbitration). It provides the following services to the link layer:

Arbitration request/confirmation: The link layer instructs the physical layer to start an arbitration procedure by means of a PHY *arbitration request*. The result of this procedure (either `won` or `lost`) is communicated to the link layer by means of a PHY *arbitration confirmation*.

Data request/indication: The link layer instructs the physical layer to put some signal on the cable by means of a PHY *data request*. The physical layer indicates to the link layer the detection of a signal on the cable (or information about the status of the cable) by means of a PHY *data indication*.

Clock indication: To notify the link layer that it can (and should) put a signal on the cable, the physical layer communicates a PHY *clock indication*.

According to [43], there is also a so-called *node controller* that can influence each of the three layers. Since, in asynchronous mode, the role of this node controller is restricted to the ability to reset each of the three layers (force them into their initial state), we will not consider the node controller in this paper.

2.2 Link layer

We proceed to describe in more detail the behaviour in asynchronous mode of the link layer (the middle layer of the three-layered protocol), which is responsible for the construction of packets, the transmission of these over a serial (one-bit) line to other parties, and the computation and verification of checksums.

We model the process behaviour of the link layer according to the state machine depicted in [43, Figure 6-19, Page 170] and the accompanying informal explanation. The part of the state machine defining the behaviour in asynchronous mode has eight states Ln ($0 \leq n \leq 7$).

The link layer processes maintain a buffer (initially empty) to store a pending request from the transaction layer.

In its initial state, the link layer can either receive a data request from the transaction layer or a data indication from the PHY layer.

At a data request, a packet is constructed from the parameters that have been put into the buffer by the transaction layer. The link layer process then starts a fair arbitration procedure to gain access to the bus. If it wins the arbitration, then the underlying physical layer controls the cable and the link layer

[1]These and other "signals" of the link layer correspond to analog signals detected or emitted by the physical layer.

enters *send mode* (see below). However, it may also happen that the physical layer indicates the arrival of data: the packet to be sent is then stored in the buffer and the data is received first.

At a data indication, it must be checked whether the received signal is a `Start` signal. If so, this means that some other node has control over the cable and is sending a packet; the incoming packet must be received in *receive mode*. Otherwise, the signal (which is not a `Start` signal) can be ignored.

Send mode. As soon as a node has gained control over the cable, its physical layer starts emitting clock indications to inform the link layer that it should send a signal. The link layer must respond to every such clock indication and send the entire packet, one signal at a time, delimited by a `Start` and an `End` signal. The `End` signal also notifies the physical layer that the link layer is done sending the packet; it will cease to send clock indications. Depending on the value of the destination field, the link layer either informs the transaction layer that a broadcast packet was sent properly, or that it must wait for an acknowledge packet.

The acknowledge packet must arrive within some specific amount of time: if a subaction gap (`subactgap` signal) occurs before an acknowledgement with valid checksum has been received entirely (i.e., up to and including the terminating `End` signal), then the link layer will act as if the acknowledgement is missing (an acknowledge packet can be identified by its length; it consists of one signal). When a `Start` signal has been received, then the link layer expects to receive an acknowledge signal. If the next signal is indeed a data signal, then the link layer receives the terminating `End` signal, checks the validity of the received acknowledge signal, and sends an *acknowledgement received* (`ackrec`) to the transaction layer. If, instead, another data signal arrives, or if there is no terminating `End` signal, or if the acknowledge packet is invalid, then the link layer sends *acknowledgement missing* (`ackmiss`) to the transaction layer. Both in case of failure and in case of success, the link layer does wait for an indication of the physical layer that a subaction gap has occurred, before it returns to its initial state. Of course, if a subaction gap interferes in the above described behaviour, then the link layer should immediately send an `ackmiss` and return to its initial state.

Receive mode. If the link layer receives a `Start` signal, it enters *receive mode*, expecting to see a packet being put on the bus by some other link layer. Asynchronous packets consist of four signals. The link layer must receive at least two signals before it can determine whether the packet is addressed to it.

If it only receives one signal followed by a terminating `End`, this is an acknowledge packet, which should be ignored: the link layer will wait for the next subaction gap and return to the initial state.

If the second signal is indeed a destination signal, the link layer must check whether the incoming packet is either a packet addressed to it, or a broadcast packet, or a packet for some other node. In the first case, the link layer must notify the physical layer that it wants access to the bus as soon as the packet has been received entirely, in anticipation of sending an acknowledgement. This is done by means of an `immediate` arbitration request. Broadcast packets, however, are not acknowledged; so, in the second case, no such request is needed. In the third case, the link layer should completely ignore the packet and return to the initial state at the next subaction gap.

The third signal is expected to be a header signal, and the fourth signal should be a data signal. If the packet is correctly terminated by either an `End` signal or a `Prefix` signal, then the packet is forwarded to the transaction layer, either as a broadcast packet or as a packet that was addressed to this node. In both cases, the data checksum is verified. Observe that, in the broadcast case, a packet with an invalid data checksum is ignored. In the other case, the packet will have to be acknowledged, so upon winning a PHY *Arbitration confirmation*, the link layer continues in *send acknowledgement mode*.

Any deviation of the above described procedure will cause the link layer to ignore the packet; it will wait for a subaction gap and then returns to the initial state. Since an `immediate` arbitration request may have been dispatched, a PHY *Arbitration confirmation* of `won` may still arrive. In such a case, the link layer is granted access to the bus, but does not need to send an acknowledgement. Therefore, if the destination signal indicated that the packet was meant for this node, the arbitration confirmation must be received, and control over the cable must be terminated immediately by sending an `End` signal.

Send acknowledgement mode. While the link layer is waiting for the transaction layer to respond to a data indication with the proper acknowledgement code, it must keep the cable busy by sending a `Prefix` signal at every clock indication; this is to avoid the occurrence of a subaction gap. Depending on the type of the received packet, the transaction layer may need to issue a so-called *concatenated response* (for instance, the packet was a read request and the transaction layer immediately wants to send the requested data to the requesting node). By means of a data response, the transaction layer communicates the proper acknowledgement, as well as one of the values `release` or `hold`. The former means that no concatenated response is requested and that, after sending the acknowledgement, the link layer may release the bus and return to its initial state. The latter means that a concatenated response is requested and that the link layer should maintain control over the bus after sending the acknowledgement packet by responding to clock indications with `Prefix` signals. Upon a data request, the link layer can then go into *send mode* immediately.

2.3 Physical layer

To simulate and analyse the interaction of the link layers of *n* nodes, we need to model the external behaviour of underlying *n* physical layers connected by a cable, which, together, we shall refer to as the *bus*.

The bus needs to keep track of which of the *n* nodes have had control over the bus during a so-called *fairness interval*; to this aim, it maintains a table of *n* Booleans. During a fairness interval, each node is allowed to gain control over the bus at most once, by means of a `fair` arbitration request. It may also access the bus more than once as a consequence of an `immediate` arbitration request. As soon as the bus has been idle for some specified amount of time and at least one link layer has got access during the running fairness interval, an *arbitration reset gap* occurs to indicate that every node may, again, be granted access through `fair` arbitration. The time interval that the bus must idle before such an arbitration reset gap may occur should be longer than that of a subaction gap.

When the bus is in idle state and the link layer of some node requests arbitration, the bus enters *decision mode*: it checks whether the requesting node already got access during the present fairness interval. If not, the bus confirms the arbitration request by indicating that the node has `won` arbitration and evolves into a *busy* state; otherwise, the bus indicates that the arbitration is `lost`.

When the bus is in busy state, it records which node has control over the bus, and which nodes have requested immediate arbitration. In this state, the bus may still receive `fair` arbitration requests, but they will be confirmed by reporting that the arbitration was `lost`. The node that must send a response to the packet put on the bus will issue an `immediate` arbitration request. No confirmation is sent, however, until the busy node releases its control. Furthermore, as long as some link layer still needs to send signals, the appropriate clock indications must be generated and signals must be distributed.

In distribution mode, the bus delivers signals to all nodes except the one that dispatched it. To obtain a realistic model, the potential loss or corruption of signals is taken into account through a function that assigns an error value to the checksum field of header signals, data signals, and acknowledge signals.

Moreover, an extra dummy value will be used to describe the situation in which packets with a invalid length are delivered. The following transmission errors are modelled:

- If the signal is a destination signal, then this signal may be invalidated. However, if this happens, the header checksum (which comes with the next signal) is no longer valid. The bus should register of which nodes invalid destinations have been distributed.
- Any signal, except for header signals having a corrupted checksum according to the above, may be delivered correctly.
- If the signal to be delivered is a header signal, a data signal or an acknowledge signal, then it may be delivered corrupted, or it may not be delivered at all.
- If the signal to be delivered is a data signal, then the packet may be extended by sending a dummy signal immediately after the data signal.

When a signal has been distributed to every node, it is checked whether this signal was an `End` signal. If so, the current busy node no longer requires access to the bus. It is then checked whether some node has requested `immediate` arbitration. If not, a `subactgap` is distributed to all nodes and the bus returns to its idle state. Otherwise, if other nodes have requested access, control over the bus must go to one of those nodes. The bus then sends arbitration confirmations and a clock indication to all nodes that requested `immediate` arbitration.

It may happen that more than one node has control over the bus. To resolve such a conflict situation, the bus must wait for `End` signals from nodes, until only one node has access. Then, a data request is received from this node. If it is not an `End` signal, the node becomes the busy one and this signal is distributed to all other nodes. However, if the received signal is an `End` signal, no node has control over the bus anymore; a `subactgap` signal is then distributed to all nodes, after which the bus returns to its idle state.

2.4 Transaction and application layers

To precisely model the lower layers of IEEE 1394, it is sufficient to combine in parallel n LINK processes and one BUS process, which describes n PHY processes and a cable. The μCRL and mCRL2 models given in Annexes A and B follow this approach for $n = 2$, with a simple MAIN process gathering two link layers and a bus.

For model-checking verification (i.e., using a model checker to exhaustively explore and analyze the reachable state space), it is desirable to describe the upper layers as well, namely, the external behaviour of the transaction layer and of the application running on top of it. To this aim, M. Sighireanu introduced in her E-LOTOS model [37] two additional processes: TRANS, which represents a transaction layer, and `Application`, which describes the application and which we note APPLI.

TRANS process. As mentioned in Section 2.1, the transaction layer provides read, write, and lock transactions to the application. Transactions follow the traditional four-step connection establishment of the OSI model: request, indication, response, and confirmation. Inside the TRANS process, outgoing requests and incoming responses are handled by two sub-processes running in parallel and synchronized together. Both types of transactions (concatenated and split) are dealt with. Further details can be found in [37, Section 7].

The deadlock problem mentioned in Section 1 is caused by a missing transition in the packet transmit/receive state machine of the link layer (precisely, in the `Link4BRec` sub-process of the μCRL and mCRL2 models). To fix this bug, one option is to modify the behaviour of the link layer to insert the missing transition, as shown in [41]. Another option (adopted in the LOTOS and LNT models to preserve compatibility with the μCRL and mCRL2 models) is to keep the LINK process unchanged and modify instead the TRANS process by removing the transition (synchronized with the LINK process) that causes the deadlock; interestingly, the 2008 revision of IEEE 1394 also kept the link-layer state machine unchanged (see [45, Figure 6-21, Page 162]). Finally, to determine the behaviour of TRANS, a parameter `v` was added, which is equal either to `ok` (deadlock-free version) or to `ko` (original version).

APPLI process. M. Sighireanu designed 11 different applications, which differ by the scenario chosen among three possibilities (see [37, Section 9.2] for details), the maximal number of nodes connected to the bus, and the maximal number of requests sent to the link layer. Combined with both variants of the TRANS process, this led to 22 different MAIN processes, hence 22 models to be verified.

NODE process. To factorize the vast amount of duplicated code among these 22 MAIN processes, H. Garavel introduced a new NODE process that expresses the parallel composition of three processes: a LINK, a TRANS, and an APPLI. Notice that, unlike the approach of [37, Section 9.2], the APPLI process is no longer invoked from within the TRANS process.

3 Formal models

In this section, we present in more detail the four formal models of the IEEE 1394 link layer, following the chronological order of their development.

3.1 Formal model in μCRL

The first formal model of the link layer was written in 1997 by B. Luttik and circulated among the COST-247 community. It was reviewed by H. Garavel, J.F. Groote, and M. Sighireanu, who provided comments that led to improvements and simplifications. It was published as an annex (nicely compacted using mathematical symbols) in [26, 27] and, since then, has remained fairly stable. The μCRL model given in Annex A is close to this original model, with three enhancements:

- It is "machine-readable", meaning that it can be executed by the μCRL toolset.
- It uses the **map** keyword added in the 1997 version of μCRL [15] to declare non-constructors, whereas the original model [26, 27] used the 1995 version of μCRL [19], which does not distinguish between constructors and non-constructors.
- It introduces `tau` internal actions in the `Resolve` and `Distribute` sub-processes of the BUS process, in order to eliminate two unguarded recursive calls that existed in the original model and that the μCRL toolset cannot handle — even if the recursion is actually bounded by the fixed number of LINK processes.

Notice that the μCRL model is quite large (809 non-blank lines), as the `Bool` and `Nat` types with all their basic functions must be defined in extension. This verbosity issue was solved in the three other formal models.

3.2 Formal model in LOTOS

In 1997, M. Sighireanu wrote a LOTOS model of the IEEE 1394 link layer, based on the draft μCRL model of B. Luttik. The development of both models at the same time led to clarifications, enhancements, and simplifications in each of them. The LOTOS model aimed at using the existing CADP toolset [8] to perform model-checking verification, and became an official demo example [40] of CADP in 1997. The LOTOS code was similar in essence to the μCRL code, but with a few differences:

- As mentioned in Section 2.4, it introduced TRANS and APPLI processes to describe the upper layers of IEEE 1394, as well as various MAIN processes specifying 22 verification scenarios.
- The LOTOS model was shorter because it imported predefined libraries containing, e.g., the `Boolean` and `NaturalNumber` types.
- The LOTOS model uses conditional rewrite rules (e.g., $C_1, ..., C_n \implies L = R$) where the μCRL model needs to take a detour via user-defined `if`(C, E, E') functions to express conditional equalities.
- The μCRL rewriter does not consider a fixed ordering of the rewrite rules: it is the modeller's responsibility to define a confluent term rewrite system. On the contrary, the CÆSAR.ADT compiler [9] for LOTOS assumes that the rewrite rules defining each (non-constructor) function are ordered by decreasing priority; this allows more concise definitions of equality functions (e.g., the `eq` comparator for type `SIGNAL` has 16 rules in μCRL and 2 in LOTOS) and other functions (e.g., `is_dest`, `is_header`, `is_data`, and `is_ack` need 10 rules each in μCRL and 2 in LOTOS).
- The LOTOS model renames all local variables `i` to `j`, because the former is a reserved LOTOS keyword that denotes the internal action (i.e., Milner's τ action). Later versions of CADP lifted this restriction by making it possible to have LOTOS variables or functions named `i`.

This LOTOS model remained stable for many years with only, in 2005, a simplification of the handwritten C code used to iterate over data domains, which was reduced from 2134 to 156 lines by factorizing similar code fragments present in the various scenarios.

However, in 2023, H. Garavel did a full revision of the LOTOS model, prompted by the development of the LNT model in parallel. The volume of LOTOS code was reduced by one third (from 2091 to 1385 lines), without loss of functionality and still preserving strong bisimilarity. This was done by merging the two versions of the TRANS process into one parameterized process, by merging the five versions of the APPLI process into another parameterized process, and by introducing the NODE process to factorize duplicated LOTOS code. A few other changes were made to simplify the LOTOS code and make it closer to the μCRL code:

- Like in the μCRL model, two LOTOS processes `Link` and `Bus` have been added to serve as main entry points.
- The definitions of the LOTOS type `SIGNAL` and of its related types have been aligned on the μCRL ones by eliminating unnecessary auxiliary tuple types. Yet, to make the LOTOS model easier to understand, the four overloaded constructors `sig` of type `SIGNAL` have been renamed to `destsig`, `acksig`, `datasig`, and `headersig`, respectively (even if LOTOS and LNT also support overloading of constructor functions).
- To reflect the model-checking assumptions of [37, Section 9.2], each of the three types `DATA`, `HEADER`, and `ACK` is directly defined as a singleton (one-value) type, rather than defining it as a two-value type and later providing ad hoc C code that only enumerates one of these two values.

3.3 Formal model in mCRL2

In June 2005, the μCRL model was translated to mCRL2 by J.F. Groote and distributed as a demo example [14] in the mCRL2 toolset.

The mCRL2 spec is 60% shorter than the μCRL one (809 non-empty lines in μCRL vs 327 in mCRL2). Most of this reduction comes from data type definitions, the size of which was roughly divided by 6.4 in mCRL2. This is explained by two factors:

- Like LOTOS, mCRL2 benefits from built-in data types (e.g., `Bool`, `Nat`, etc.), together with their basic functions, which need not be defined in every model.
- Like functional languages (ML, Haskell, etc.) and E-LOTOS [21], mCRL2 types can be defined by their constructors. For instance, the `SIGNAL` type is defined using the **struct** construct of mCRL2 and the `BoolTABLE` type is concisely defined using the built-in `List` datatype. For such types, equality functions, recognizers (i.e., functions, such as `is_dest`, that check whether an expression matches a given constructor), and projections (i.e., functions, such as `first`, `second`, `third`, and `fourth` for type `quadruple`, that extract the various arguments of a constructor) are defined automatically.

The mCRL2 processes differ on minor points from the μCRL ones:

- The syntax of the "**if** *C* **then** *A* **else** *B*" construct has changed: it is noted "*C* `->` *A* `<>` *B*" in mCRL2 and "*A* `<|` *C* `|>` *B*" in μCRL.
- In the LINK process, the μCRL definitions of the `Link0` and `Link7` sub-processes contain summations (i.e., nondeterministic choices) ranging over natural numbers that are not restricted in any way. In the mCRL2 model, these summations are bounded by the number of LINK layers.
- In the mCRL2 model, each `tau` action introduced to guard recursion (see Section 3.1) is replaced by an action `internal`, which is later abstracted from.

3.4 Formal model in LNT

Besides developing a complete LOTOS model and using it for model-checking verification, M. Sighireanu also wrote an E-LOTOS model of the IEEE 1394 link layer that was, rather than the LOTOS model itself, presented in [37, 38, 39]. At this time, the E-LOTOS language was still being standardized and not finalized yet. In essence, the E-LOTOS model bears similarities with the mCRL2 model developed later, notwithstanding the syntactic differences between both languages.

The LNT model presented in Annex D does not derive from this E-LOTOS model, as its history is distinct. In 2022, the LOTOS model (taken in its original version) was partly translated to LNT by Oussama Oulkaid and Marck-Edward Kemeh, as part of an exercise for master students at the University of Grenoble. Their model was later reworked and reshaped by H. Garavel, in order to make it complete and strongly bisimilar to the LOTOS one. Because it had been obtained by systematic translation, this LNT model was very much in the same style as the μCRL, mCRL2, and LOTOS ones: namely, data types defined as term rewrite systems, and processes defined as state machines extended with local variables that can be read and modified on transitions.

Therefore, H. Garavel entirely revised this LNT model in order to obtain a "better" model that would exploit the characteristic features of LNT and demonstrate the full capabilities of this language. This revision was achieved by progressive transformations, checking at each step that strong bisimilarity is preserved. Concerning data specifications in the resulting LNT model, three main remarks can be made:

- The type definitions in LNT are similar (up to syntax) to mCRL2 ones, except that equality/inequality functions must be requested explicitly (using "**with** =" and "**with** <>" clauses) and that functions for extracting/updating constructor arguments must also be requested (using "**with** `get`" and "**with** `set`" clauses); this ensures that LNT models are self-contained and not cluttered with useless implicit functions.

- As regards function definitions, the LOTOS rewrite rules ordered by decreasing priority can be systematically translated to LNT pattern-matching **case** statements. However, this is not the only style permitted by LNT, and not necessarily the most concise and readable one. One can also define functions in a more imperative style, with the usual programming constructs (variable assignments, **if**-**then**-**else**, **return** statements, etc.), as shown, for instance, in the various functions manipulating values of type `BoolTABLE`.

- A salient difference between μCRL, LOTOS, and mCRL2, on the one hand, and LNT, on the other hand, concerns partial functions, i.e., functions that are not defined over the entire domain of their arguments (e.g., function `get` for the `BoolTABLE` type or functions `getdest`, `getdcrc`, `getdata`, `gethead`, `getadd`, and `corrupt` for the `SIGNAL` type). In μCRL, LOTOS, and mCRL2, partial definition is implicit, in the sense that some equations are not given, e.g., there is no equation to define "`get (`n`, empty)`". The LOTOS model of Annex C contains comments to warn about partial definitions, but this is left to the good will of the specifier.

  In LNT, the situation is different: any partial function triggers (based on control- and data-flow analysis) an error, which the specifier is expected to correct, either by properly dealing with the overlooked cases, or by explicitly inserting a "**raise** *E*" statement at each point where the function might terminate without returning a result — *E* being either an event declared as an exception that the function can raise, or the predefined event `UNEXPECTED` denoting an exception that cannot be caught and triggers a run-time error.

Concerning processes, the following five transformations have been repeatedly applied until an idiomatic LNT model was obtained:

- The guarded commands "$[C] \rightarrow A$ `[]` $[\mathtt{not}(C)] \rightarrow B$" present in the LOTOS model have been translated to "**if** *C* **then** *A* **else** *B* **end if**" statements of LNT. The **then** and **else** branches have been permuted, negating the Boolean condition *C*, when *B* was much shorter than *A*. Also, nested **if** statements have been flattened whenever possible by using the (Ada-like) **elsif** clause of LNT.

- When this was convenient, calls to recursive processes have been replaced by the **loop** statements of LNT, possibly with a **break** statement to exit the loop. For instance, the `Link3`, `Link5`, and `Link7` processes of the μCRL, mCRL2, and LOTOS models have been replaced, in the LNT model, by **loop** statements. Indeed, in μCRL, mCRL2, and LOTOS, (finite or infinite) iteration must always be expressed using recursion, with two main drawbacks: (i) the mandatory use of recursion obfuscates the flow of control by requiring the definition of auxiliary recursive processes and "goto-like" calls to these processes; (ii) it also obfuscates the flow of data by requiring, for such processes, as many parameters as there are live variables at the point where these processes are called. Using iteration rather than recursion often leads to simpler, more readable models.

- In some cases, finite loops can be further simplified by turning them into **while** or **for** loops. For instance, the sub-process `Resolve2` of the μCRL, mCRL2, and LOTOS models can be rephrased as a **while** loop, whereas the sub-processes `Resolve`, `SubactionGap`, and `Distribute` can be described using **for** loops, hereby getting rid of the extra parameters that store the loop variables.

Notice that such iterative behaviour was quite clear from the textual description of these processes in [26], but only LNT enables one to express it in natural way.

- Processes that are called only once (especially after recursion has been replaced by iteration) should be expanded in-line at the point where they are called. Doing so, the control flow becomes more readable (as each process call is similar to a "goto") and many process parameters are eliminated. M. Sighireanu applied this idea when designing her E-LOTOS model: the two μCRL sub-processes `DecideIdle` and `Link1` were expanded in-line [37, footnotes 7 and 8]. In the LNT model of Annex D, this idea was pushed beyond by also eliminating the sub-processes `Link3`, `Link3RA`, `Link3RE`, `Link4DH`, `Link4RH`, `Link4RD`, `Link4RE`, `Link4BRec`, `Link4DRec`, `Link5`, `Link6`, `Link7`, `Resolve`, and `Resolve2`. The sub-process `Link4`, although called only once, was not expanded in-line, because it is so large that its expansion would have increased the nesting depth too much. Also, a new `Link2` sub-process was added to factorize both sub-processes `Link2req` and `Link2resp` in a single one. As a result, the LINK process has only 6 (mutually recursive) processes in the LNT model, instead of 19 in the other models — maintaining an exact correspondence with the 8 states describing the asynchronous mode [43, Figure 6-19, Page 170] was not considered a requirement for the LNT model.
- Since the in-line expansion of processes often creates variables with nested scopes, three additional transformations may be suitable to keep the LNT model simple:
  - merging different variables that have the same type and are never used simultaneously, so as to decrease the number of variables.
  - enlarging the scope of nested variables by moving their declarations upward, so has to reduce the nesting depth of variable scopes;
  - renaming nested variables declared in the scope of another variable having the same name; for instance, after successively expanding the sub-process `Link7` in `Link6`, `Link6` in `Link5`, and `Link5` in `Link4DRec`, the `d` variable of `Link7` arrives in the scope of the `d` variable of `Link4DRec`; even if the innermost variable hides the outermost one in LNT (as in Algol-60), it may be suitable to give these variables different names to avoid confusion.

  These transformations sometimes conflict with each other, and their judicious application cannot be governed by strict laws: it is rather a matter of taste and circumstances.

4 Verification

The four formal models of the IEEE 1934 link layer have been checked by their respective compilers: the μCRL toolset, the mCRL2 toolset, and, for the LOTOS and LNT models, the CADP toolset.

The five correctness properties stated by B. Luttik [26, Section 4] have been formulated in the ACTL temporal logic [6] by R. Mateescu and M. Sighireanu [37, Section 10]. Using the XTL [29] model checker of CADP, these formulas have been checked on 16 out of 22 variants of the LOTOS model (totalling 80 model-checking jobs), the domains of the types `ACK`, `DATA`, and `HEADER` being limited to a single value. All the properties hold, except the first property (deadlock freeness), which is violated on the "original" models when the application layer executes its most complex scenarios.

The LNT model has been verified in two ways, using both model checking and equivalence checking. On the one hand, the ACTL formulas evaluate identically on the 16 variants of the LNT model. On the other hand, the labelled transition systems generated from 20 out of 22 variants of the LNT model are strongly bisimilar to those generated from the same variants of the LOTOS model. The labelled transition systems of the two remaining variants are too large for being generated directly, and would certainly benefit from compositional verification techniques [11]. In 14 cases out of 20, the labelled transition systems generated from LOTOS and LNT have the same size, whereas in 6 cases, those generated from LNT are slightly larger (+0.46% states, +0.43% transitions). Using version 2024-a "Eindhoven" of the CADP toolbox, these verifications were performed in less than 8 minutes on a Dell Latitude 5580 (Intel Core i5-7200U processor, 16 GB RAM) running Linux.

5 Conclusion

Revisiting the IEEE 1394 link layer problem, a true success story of formal methods, we presented and discussed four models written in μCRL, mCRL2, LOTOS, and LNT — the LOTOS model (revised in 2023) and the LNT model being novel contributions. In this respect, the present paper is a tentative "Rosetta stone" for comparing various modelling languages dedicated to communication protocols and concurrent systems. In a nutshell, our main findings are as follows:

- It appears that the three languages μCRL, mCRL2, and LOTOS are quite close, except that data type specifications are more concise in the latter two languages. Each of these three languages contains two separate sub-languages: one for specifying data types (using algebraic specifications or term rewrite rules), and another one for concurrent processes.

  These sub-languages sometimes use distinct symbols to express the same concept (e.g., **if-then-else** being noted differently in the data and process parts) and sometimes give the same symbol totally different meanings, e.g., in μCRL and mCRL2, the "$+$" operator (which denotes addition in the data part and nondeterministic choice in the process part), the "||" operator (which denotes logical disjunction in the data part and parallel composition in the process part), or closing parentheses (which denote the end of expressions in the data part and the end of a choice, a sequential composition, etc. in the process part).

  On the contrary, LNT is a unified language, without separate sub-languages: LNT functions and LNT processes are defined using the same notations (";" for sequential composition, **if-then-else** for conditionals, etc.), and LNT avoids, as much as possible, "overloaded" symbols.

- Although it has been argued that LOTOS supports very diverse "specification styles" [50], most LOTOS, μCRL, and mCRL2 models consist of a set of concurrent processes, each of which being specified using guarded commands and terminal recursion. Such a style is convenient for describing automata extended with state variables, but leads to models that are difficult to maintain when specifications evolve frequently, and does not scale well when automata complexity increases, resulting in large, poorly structured state machines scattered with "goto-like" transitions.

  In addition to supporting guarded commands and terminal recursion, LNT provides alternative specification styles suitable for the description of complex systems. In particular, LNT offers the classical primitives of structured programming, properly bracketed with an Ada-like syntax, which make large models easier to read and reduce the need for drawing state machines on paper.

To some extent, there is here a debate around the concept of minimality and how it should be interpreted. On the one hand, LOTOS, μCRL, and mCRL2 try to be minimal in the size of the language[2], the number of syntactic constructs, and the number of semantic rules. An explicit concern for μCRL and mCRL2 has been to ensure that the semantics are as simple and elegant as possible, only including constructs in

[2]The μ letter (which stands for "micro") in μCRL indeed expresses such a desire for minimality.

the language if they are needed for expressiveness; ease of modelling has been less of a concern so far. LNT also tries to be minimal, e.g., by unifying the sub-languages for functions and processes, the former being included in the latter, but it can be rightly argued that LNT is richer than the three other languages and requires more complex compilers that implement involved control- and data-flow analyses.

Perhaps the proper concept of minimality is not so much about the size of a language or of its compiler, but about the effort needed to learn the language, the time needed to write correct models, and the difficulty of understanding such models for engineers who do not have a strong background in formal methods. We hope that the present study will usefully contribute to this debate.

Acknowledgements

We are grateful to all those who contributed to the design and verification of the four formal models of the IEEE 1394 link layer, namely: Jan Friso Groote, Marck-Edward Kemeh, Radu Mateescu, Laurent Mounier, Oussama Oulkaid, Charles Pecheur, Judi Romijn, Mihaela Sighireanu, and Bruno Vivien. We also thank the anonymous reviewers for their constructive remarks.

References

[1] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems – Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *Proceedings (Part II) of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019), Prague, Czech Republic Proceedings*, *Lecture Notes in Computer Science* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[2] Vincenza Carchiolo, Michele Malgeri & Giuseppe Mangioni (2003): *Synthesis of LOTOS Specification of the IEEE-1394 Firewire Protocol*. In: *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03), San Diego, California, USA*, IEEE Computer Society Press, pp. 86–92, doi:10.1109/IWRSP.2003.1207034.

[3] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe & Gideon Smeding (2023): *Reference Manual of the LNT to LOTOS Translator (Version 7.1)*. Available at `http://cadp.inria.fr/publications/Champelovier-Clerc-Garavel-et-al-10.html`. INRIA, Grenoble, France.

[4] Conrado Daws, Marta Z. Kwiatkowska & Gethin Norman (2002): *Automatic Verification of the IEEE-1394 Root Contention Protocol with KRONOS and PRISM*. In Rance Cleaveland & Hubert Garavel, editors: *Proceedings of the 7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'02), Málaga, Spain*, *Electronic Notes in Theoretical Computer Science* 66, Elsevier, pp. 104–119, doi:10.1016/S1571-0661(04)80406-7.

[5] Conrado Daws, Marta Z. Kwiatkowska & Gethin Norman (2004): *Automatic Verification of the IEEE 1394 Root Contention Protocol with KRONOS and PRISM*. *International Journal on Software Tools for Technology Transfer (STTT)* 5(2–3), pp. 221–236, doi:10.1007/S10009-003-0118-5.

[6] Rocco De Nicola & Frits W. Vaandrager (1990): *Action versus State based Logics for Transition Systems*. In Irène Guessarian, editor: *Semantics of Systems of Concurrent Processes – Proceedings of the LITP Spring School on Theoretical Computer Science, La Roche Posay, France*, *Lecture Notes in Computer Science* 469, Springer, pp. 407–419, doi:10.1007/3-540-53479-2_17.

[7] Marco Devillers, W. O. David Griffioen, Judi Romijn & Frits W. Vaandrager (2000): *Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394*. *Formal Methods in System Design* 16(3), pp. 307–320, doi:10.1023/A:1008764923992.

[8] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier & Mihaela Sighireanu (1996): *CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox*. In Rajeev Alur & Thomas A. Henzinger, editors: *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96), New Brunswick, New Jersey, USA*, *Lecture Notes in Computer Science* 1102, Springer, pp. 437–440, doi:10.1007/3-540-61474-5_97.

[9] Hubert Garavel (1989): *Compilation of LOTOS Abstract Data Types*. In Son T. Vuong, editor: *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, North Holland, pp. 147–162. Available at `http://cadp.inria.fr/publications/Garavel-89-c.html`.

[10] Hubert Garavel (2008): *Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular*. In Catuscia Palamidessi & Frank D. Valencia, editors: *Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory, Ecole Polytechnique de Paris, France, November 13–15, 2006*, *Electronic Notes in Theoretical Computer Science* 209, Elsevier Science Publishers, pp. 149–164, doi:10.1016/J.ENTCS.2008.04.009. Also available as INRIA Research Report RR-6368.

[11] Hubert Garavel, Frédéric Lang & Laurent Mounier (2018): *Compositional Verification in Action*. In Falk Howar & Jiri Barnat, editors: *Proceedings of the 23rd International Conference on Formal Methods for Industrial Critical Systems (FMICS'18), Maynooth, Ireland – Essays Dedicated to Susanne Graf at the Occasion of Her 60th Birthday*, *Lecture Notes in Computer Science* 11119, Springer, pp. 189–210, doi:10.1007/978-3-030-00244-2_13.

[12] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *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.

[13] Hubert Garavel & Mihaela Sighireanu (1998): *Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS*. In Jan-Friso Groote, Bas Luttik & Jos van Wamel, editors: *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems (FMICS'98), Amsterdam, The Netherlands*, CWI, Amsterdam, pp. 187–230. Available at `http://cadp.inria.fr/publications/Garavel-Sighireanu-98-a.html`.

[14] Jan Friso Groote: *IEEE 1394 Link Layer in mCRL2*. Available at `https://github.com/mCRL2org/mCRL2/tree/master/examples/industrial/1394`.

[15] Jan Friso Groote (1997): *The Syntax and Semantics of Timed μCRL*. Technical Report SEN-R9709, CWI, Amsterdam, The Netherlands. Available at `https://ir.cwi.nl/pub/4746`.

[16] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko & Muck van Weerdenburg (2007): *The Formal Specification Language mCRL2*. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens & Roel Wieringa, editors: *Methods for Modelling Software Systems (MMOSS)*, *Dagstuhl Seminar Proceedings* 06351, Schloss Dagstuhl, Germany, pp. 1–34, doi:10.4230/DagSemProc.06351.12.

[17] Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg & Yaroslav S. Usenko (2006): *From μCRL to mCRL2: Motivation and Outline*. *Electronic Notes in Theoretical Computer Science* 162, pp. 191–196, doi:10.1016/j.entcs.2005.12.101.

[18] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. The MIT Press, doi:10.7551/mitpress/9946.001.0001.

[19] Jan Friso Groote & Alban Ponse (1995): *The Syntax and Semantics of μCRL*. In A. Ponse, C. Verhoef & S.F.M. van Vlijmen, editors: *Proceedings of the 1st Workshop on the Algebra of Communicating Processes (ACP'94), Utrecht, The Netherlands*, Workshops in Computing Series, Springer, pp. 26–62, doi:10.1007/978-1-4471-2120-6_2.

[20] ISO/IEC (1989): *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva. Available at `https://www.iso.org/standard/16258.html`.

[21] ISO/IEC (2001): *Enhancements to LOTOS (E-LOTOS).* International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva. Available at `https://www.iso.org/standard/27680.html`.

[22] Mark Jorgensen & Hubert Garavel (1997): *Final Report of the COST-247 Action.* Available at `https://vasy.inria.fr/COST247`.

[23] Lars Kühne, Jozef Hooman & Willem-Paul de Roever (1997): *Towards Mechanical Verification of Parts of the IEEE P1394 Serial Bus.* In Ignac Lovrek, editor: *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, pp. 73–85.

[24] Marta Z. Kwiatkowska, Gethin Norman & Jeremy Sproston (2003): *Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol.* *Formal Aspects of Computing* 14(3), pp. 295–318, doi:10.1007/S001650300007.

[25] Izak van Langevelde, Judi Romijn & Nicolae Goga (2003): *Founding FireWire Bridges through Promela Prototyping.* In: *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France*, IEEE Computer Society, p. 239, doi:10.1109/IPDPS.2003.1213434.

[26] Bas Luttik (1997): *Description and Formal Specification of the Link Layer of P1394.* Report SEN-R9706, CWI, Software Engineering (SEN), Amsterdam, The Netherlands. Available at `https://ir.cwi.nl/pub/4758`.

[27] Bas Luttik (1997): *Description and Formal Specification of the Link Layer of P1394.* In Ignac Lovrek, editor: *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, pp. 43–56.

[28] Savi Maharaj & Carron Shankland (2000): *A Survey of Formal Methods Applied to Leader Election in IEEE 1394.* *Journal of Universal Computer Science* 6(11), pp. 1145–1163. Available at `http://www.jucs.org/jucs_6_11/a_survey_of_formal`.

[29] Radu Mateescu & Hubert Garavel (1998): *XTL: A Meta-Language and Tool for Temporal Logic Model-Checking.* In Tiziana Margaria, editor: *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark*, BRICS, pp. 33–42. Available at `http://cadp.inria.fr/publications/Mateescu-Garavel-98.html`.

[30] Judi Romijn (1999): *A Timed Verification of the IEEE 1394 Leader Election Protocol.* In Stefania Gnesi & Diego Latella, editors: *Proceedings of the 4th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'99), Trento, Italy*, pp. 3–29.

[31] Judi Romijn (2001): *A Timed Verification of the IEEE 1394 Leader Election Protocol.* *Formal Methods in System Design* 19(2), pp. 165–194, doi:10.1023/A:1011284000753.

[32] Judi Romijn (2003): *False Loop Detection in the IEEE 1394 Tree Identify Phase.* *Formal Aspects of Computing* 14(3), pp. 319–327, doi:10.1007/S001650300008.

[33] Carron Shankland & Alberto Verdejo (1999): *Time, E-LOTOS, and the FireWire.* In Marco Ajmone Marsan, Juan Quemada, Tomás Robles & Manuel Silva, editors: *Proceedings of the Workshop on Formal Methods and Telecommunications (WFMT'99), Zaragoza, Spain*, Prensas Universitarias de Zaragoza, pp. 103–119. Available at `http://maude.sip.ucm.es/alberto-verdejo/papers/FireWire99.html`.

[34] Carron Shankland & Alberto Verdejo (2001): *A Case Study in Abstraction Using E-LOTOS and the FireWire.* *Computer Networks* 37(3/4), pp. 481–502, doi:10.1016/S1389-1286(01)00190-6.

[35] Carron Shankland & Mark van der Zwaag (1998): *The Tree Identify Protocol of IEEE 1394 in μCRL.* *Formal Aspects of Computing* 10(5-6), pp. 509–531, doi:10.1007/s001650050030.

[36] Mihaela Sighireanu, Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe & Jan Stoecker (2023): *LOTOS NT User's Manual (Version 3.12).* INRIA/CONVECS, Grenoble, France, `https://vasy.inria.fr/ftp/traian/manual.pdf`, 88 pages.

[37] Mihaela Sighireanu & Radu Mateescu (1997): *Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS.* Research Report RR-3172, INRIA, France. Available at `http://cadp.inria.fr/publications/Sighireanu-Mateescu-97.html`.

[38] Mihaela Sighireanu & Radu Mateescu (1997): *Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS*. In Ignac Lovrek, editor: *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, pp. 57–72. Full version available as INRIA Research Report RR-3172.

[39] Mihaela Sighireanu & Radu Mateescu (1998): *Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): an Experiment with E-LOTOS*. *Springer International Journal on Software Tools for Technology Transfer (STTT)* 2(1), pp. 68–88, doi:10.1007/S100090050018.

[40] Mihaela Sighireanu, Radu Mateescu & Hubert Garavel: *CADP Demo № 23*. Available at `http://cadp.inria.fr/ftp/demos/demo_23`.

[41] Mihaela Sighireanu, Radu Mateescu & Hubert Garavel (1998): *VASY Reports a Deadlock in the IEEE 1394 "Firewire" Standard*. Available at `https://vasy.inria.fr/press/firewire.html`.

[42] David P. L. Simons & Mariëlle Stoelinga (2001): *Mechanical Verification of the IEEE 1394a Root Contention Protocol Using Uppaal2k*. *International Journal on Software Tools for Technology Transfer (STTT)* 3(4), pp. 469–485, doi:10.1007/S100090100059.

[43] IEEE Computer Society (1995): *IEEE Standard for a High Performance Serial Bus*. IEEE Standard 1394-1995, Institution of Electrical and Electronic Engineers, doi:10.1109/IEEESTD.1996.81049.

[44] IEEE Computer Society (1995): *P1394 Standard for a High Performance Serial Bus*. Technical Report, Institution of Electrical and Electronic Engineers. Draft 8.0v2.

[45] IEEE Computer Society (2008): *IEEE Standard for a High Performance Serial Bus*. IEEE Standard 1394-2008, Institution of Electrical and Electronic Engineers, doi:10.1109/IEEESTD.2008.4659233.

[46] Mariëlle Stoelinga (2003): *Fun with FireWire: A Comparative Study of Formal Verification Methods Applied to the IEEE 1394 Root Contention Protocol*. *Formal Aspects of Computing* 14(3), pp. 328–337, doi:10.1007/S001650300009.

[47] Mariëlle Stoelinga & Frits W. Vaandrager (1999): *Root Contention in IEEE 1394*. In Joost-Pieter Katoen, editor: *Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99), Bamberg, Germany*, *Lecture Notes in Computer Science* 1601, Springer, pp. 53–74, doi:10.1007/3-540-48778-6_4.

[48] Alberto Verdejo, Isabel Pita & Narciso Martí-Oliet (2000): *The Leader Election Protocol of IEEE 1394 in Maude*. In Kokichi Futatsugi, editor: *Proceedings of the 3rd International Workshop on Rewriting Logic and its Applications (WRLA 2000), Kanzawa, Japan*, *Electronic Notes in Theoretical Computer Science* 36, Elsevier, pp. 383–404, doi:10.1016/S1571-0661(05)80133-1.

[49] Alberto Verdejo, Isabel Pita & Narciso Martí-Oliet (2003): *Specification and Verification of the Tree Identify Protocol of IEEE 1394 in Rewriting Logic*. *Formal Aspects of Computing* 14(3), pp. 228–246, doi:10.1007/S001650300003.

[50] C. Vissers, G. Scollo, M. van Sinderen & E. Brinksma (1991): *Specification Styles in Distributed Systems Design and Verification*. *Theoretical Computer Science* 89(1), pp. 179–206, doi:10.1016/0304-3975(90)90111-T.

A Formal model in μCRL

A.1 Types and functions in μCRL

% Boolean type

sort `Bool`
func

```
  T,F: -> Bool
```

map
```
  eq: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  eq(T,b)=b
  eq(b,T)=b
  eq(b,F)=not(b)
  eq(F,b)=not(b)
```

map
```
  and: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  and(T,b)=b
  and(b,T)=b
  and(b,F)=F
  and(F,b)=F
```

map
```
  or: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  or(T,b)=T
  or(b,T)=T
  or(b,F)=b
  or(F,b)=b
```

map
```
  not: Bool -> Bool
  if: Bool#Bool#Bool -> Bool
```
var
```
  b1,b2: Bool
```
rew
```
  not(F)=T
  not(T)=F
  if(T,b1,b2)=b1
  if(F,b1,b2)=b2
```

% Natural number type

sort `NAT`
func
```
  0,1,2: -> NAT
```
% 0,1,2,3,4,5,6,7,8,9: −> NAT
map `succ: NAT -> NAT`

map

```
  eq: NAT#NAT -> Bool
var
  n,m: NAT
rew
  1=succ(0)
  2=succ(1)
  eq(0,0)=T
  eq(succ(n),0)=F
  eq(0,succ(n))=F
  eq(succ(n),succ(m))=eq(n,m)

map
  lt: NAT#NAT -> Bool
var
  n,m: NAT
rew
  lt(0,0)=F
  lt(succ(n),0)=F
  lt(0,succ(n))=T
  lt(succ(n),succ(m))=lt(n,m)

% Data/Control/Acknowledge elemens and their CRC computation

sort CHECK
func
  bottom,check: -> CHECK
map
  eq: CHECK#CHECK -> Bool
rew
  eq(bottom,bottom)=T
  eq(check,check)=T
  eq(check,bottom)=F
  eq(bottom,check)=F

sort DATA
func
  d1,d2: -> DATA
map
  crc: DATA -> CHECK
  eq: DATA#DATA -> Bool
rew
  crc(d1)=check
  crc(d2)=check
  eq(d1,d1)=T
  eq(d1,d2)=F
  eq(d2,d1)=F
  eq(d2,d2)=T

sort HEADER
func
  h1,h2: -> HEADER
map
```

```
  crc: HEADER -> CHECK
  eq: HEADER # HEADER -> Bool
rew
  crc(h1)=check
  crc(h2)=check
  eq(h1,h1)=T
  eq(h1,h2)=F
  eq(h2,h1)=F
  eq(h2,h2)=T

sort ACK
func
  a1,a2: -> ACK
map
  crc: ACK -> CHECK
  eq : ACK # ACK -> Bool
rew
  crc(a1)=check
  crc(a2)=check
  eq(a1,a1)=T
  eq(a1,a2)=F
  eq(a2,a1)=F
  eq(a2,a2)=T

sort SIGNAL
func
  sig: NAT -> SIGNAL
  sig: HEADER#CHECK -> SIGNAL
  sig: DATA#CHECK -> SIGNAL
  sig: ACK#CHECK -> SIGNAL

  Start,End: -> SIGNAL
  Prefix,subactgap: -> SIGNAL
  dhead,Dummy: -> SIGNAL
map
  is_start,is_end: SIGNAL -> Bool
  is_prefix,is_sagap: SIGNAL -> Bool
  is_dummy,is_dhead: SIGNAL -> Bool
  eq: SIGNAL#SIGNAL -> Bool
var
  n,n' : NAT
  h,h' : HEADER
  d,d' : DATA
  a,a' : ACK
  c,c' : CHECK
  s : SIGNAL
rew
  is_start(Start)=T
  is_start(End)=F
  is_start(Prefix)=F
  is_start(subactgap)=F
  is_start(dhead)=F
  is_start(Dummy)=F
  is_start(sig(n))=F
  is_start(sig(h,c))=F
  is_start(sig(d,c))=F
  is_start(sig(a,c))=F
  eq(Start,s)=is_start(s)
  eq(s,Start)=is_start(s)

  is_end(End)=T
  is_end(Start)=F
  is_end(Prefix)=F
  is_end(subactgap)=F
  is_end(dhead)=F
  is_end(Dummy)=F
  is_end(sig(n))=F
  is_end(sig(h,c))=F
  is_end(sig(d,c))=F
  is_end(sig(a,c))=F
  eq(End,s)=is_end(s)
  eq(s,End)=is_end(s)

  is_prefix(Prefix)=T
  is_prefix(Start)=F
  is_prefix(End)=F
  is_prefix(subactgap)=F
  is_prefix(dhead)=F
  is_prefix(Dummy)=F
  is_prefix(sig(n))=F
  is_prefix(sig(h,c))=F
  is_prefix(sig(d,c))=F
  is_prefix(sig(a,c))=F
  eq(Prefix,s)=is_prefix(s)
  eq(s,Prefix)=is_prefix(s)

  is_sagap(subactgap)=T
  is_sagap(Start)=F
  is_sagap(End)=F
  is_sagap(Prefix)=F
  is_sagap(dhead)=F
  is_sagap(Dummy)=F
  is_sagap(sig(n))=F
  is_sagap(sig(h,c))=F
  is_sagap(sig(d,c))=F
  is_sagap(sig(a,c))=F
  eq(subactgap,s)=is_sagap(s)
  eq(s,subactgap)=is_sagap(s)

  is_dhead(subactgap)=F
  is_dhead(Start)=F
  is_dhead(End)=F
  is_dhead(Prefix)=F
  is_dhead(dhead)=T
```

```
  is_dhead(Dummy)=F
  is_dhead(sig(n))=F
  is_dhead(sig(h,c))=F
  is_dhead(sig(d,c))=F
  is_dhead(sig(a,c))=F
  eq(dhead,s)=is_dhead(s)
  eq(s,dhead)=is_dhead(s)

  is_dummy(subactgap)=F
  is_dummy(Start)=F
  is_dummy(End)=F
  is_dummy(Prefix)=F
  is_dummy(dhead)=F
  is_dummy(Dummy)=T
  is_dummy(sig(n))=F
  is_dummy(sig(h,c))=F
  is_dummy(sig(d,c))=F
  is_dummy(sig(a,c))=F
  eq(Dummy,s)=is_dummy(s)
  eq(s,Dummy)=is_dummy(s)

  eq(sig(n),sig(n'))=eq(n,n')
  eq(sig(n),sig(h,c))=F
  eq(sig(n),sig(d,c))=F
  eq(sig(n),sig(a,c))=F
  eq(sig(h,c),sig(n'))=F
  eq(sig(h,c),sig(h',c'))=and(eq(h,h'),eq(c,c'))
  eq(sig(h,c),sig(d,c'))=F
  eq(sig(h,c),sig(a,c'))=F
  eq(sig(d,c),sig(n))=F
  eq(sig(d,c),sig(h,c'))=F
  eq(sig(d,c),sig(d',c'))=and(eq(d,d'),eq(c,c'))
  eq(sig(d,c),sig(a,c'))=F
  eq(sig(a,c),sig(n))=F
  eq(sig(a,c),sig(h,c'))=F
  eq(sig(a,c),sig(d,c'))=F
  eq(sig(a,c),sig(a',c'))=and(eq(a,a'),eq(c,c'))
```

map

```
  is_dest,is_header: SIGNAL -> Bool
  is_data,is_ack: SIGNAL -> Bool
```

var

```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```

rew

```
  is_dest(sig(n))=T
  is_dest(sig(h,c))=F
  is_dest(sig(d,c))=F
  is_dest(sig(a,c))=F
  is_dest(Start)=F
  is_dest(End)=F
  is_dest(Prefix)=F
  is_dest(subactgap)=F
  is_dest(dhead)=F
  is_dest(Dummy)=F

  is_header(sig(h,c))=T
  is_header(sig(n))=F
  is_header(sig(d,c))=F
  is_header(sig(a,c))=F
  is_header(Start)=F
  is_header(End)=F
  is_header(Prefix)=F
  is_header(subactgap)=F
  is_header(dhead)=F
  is_header(Dummy)=F

  is_data(sig(d,c))=T
  is_data(sig(n))=F
  is_data(sig(h,c))=F
  is_data(sig(a,c))=F
  is_data(Start)=F
  is_data(End)=F
  is_data(Prefix)=F
  is_data(subactgap)=F
  is_data(dhead)=F
  is_data(Dummy)=F

  is_ack(sig(a,c))=T
  is_ack(sig(n))=F
  is_ack(sig(h,c))=F
  is_ack(sig(d,c))=F
  is_ack(Start)=F
  is_ack(End)=F
  is_ack(Prefix)=F
  is_ack(subactgap)=F
  is_ack(dhead)=F
  is_ack(Dummy)=F
```

map
```
  is_physig,is_terminator: SIGNAL -> Bool
```
var
```
  s : SIGNAL
```
rew
```
  is_physig(s)=or(is_start(s),or(is_end(s),or(is_prefix(s),is_sagap(s))))
  is_terminator(s)=or(is_end(s),is_prefix(s))
```

map
```
  is_hda: SIGNAL -> Bool
```
var
```
  s : SIGNAL
```

rew
```
  is_hda(s)=or(is_header(s),or(is_data(s),is_ack(s)))
```

map
```
  valid_hpart, valid_ack: SIGNAL -> Bool
```
var
```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```
rew
```
  valid_ack(sig(a,c))=eq(c,check)
  valid_ack(sig(h,c))=F
  valid_ack(sig(d,c))=F
  valid_ack(sig(n))=F
  valid_ack(Start)=F
  valid_ack(End)=F
  valid_ack(Prefix)=F
  valid_ack(subactgap)=F
  valid_ack(Dummy)=F
  valid_ack(dhead)=F

  valid_hpart(sig(h,c))=eq(c,check)
  valid_hpart(sig(n))=F
  valid_hpart(sig(d,c))=F
  valid_hpart(sig(a,c))=F
  valid_hpart(Start)=F
  valid_hpart(End)=F
  valid_hpart(Prefix)=F
  valid_hpart(subactgap)=F
  valid_hpart(Dummy)=F
  valid_hpart(dhead)=F
```

map
```
  getdest: SIGNAL -> NAT
  getdcrc: SIGNAL -> CHECK
  getdata: SIGNAL -> DATA
  gethead: SIGNAL -> HEADER
  getack: SIGNAL -> ACK
  corrupt: SIGNAL -> SIGNAL
```
var
```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```
rew
```
  getdest(sig(n)) = n
  gethead(sig(h,c)) = h
  getdcrc(sig(d,c)) = c
  getdata(sig(d,c)) = d
  getack (sig(a,c)) = a

  corrupt(sig(h,c)) = sig(h,bottom)
  corrupt(sig(d,c)) = sig(d,bottom)
  corrupt(sig(a,c)) = sig(a,bottom)
```

sort `SIG_TUPLE`
func
```
  quadruple: SIGNAL#SIGNAL#SIGNAL#SIGNAL -> SIG_TUPLE
  void: -> SIG_TUPLE
```
map
```
  first,second,third,fourth: SIG_TUPLE -> SIGNAL
  is_void: SIG_TUPLE -> Bool
```
var
```
  x1,x2,x3,x4: SIGNAL
```
rew
```
  first(quadruple(x1,x2,x3,x4))=x1
  second(quadruple(x1,x2,x3,x4))=x2
  third(quadruple(x1,x2,x3,x4))=x3
  fourth(quadruple(x1,x2,x3,x4))=x4

  is_void(void)=T
  is_void(quadruple(x1,x2,x3,x4))=F
```

sort `PAR`
func
```
  fair,immediate: -> PAR
```
map
```
  eq: PAR#PAR -> Bool
```
rew
```
  eq(fair,fair)=T
  eq(immediate,immediate)=T
  eq(fair,immediate)=F
  eq(immediate,fair)=F
```

sort `PAC`
func
```
  won,lost: -> PAC
```
map
```
  eq: PAC#PAC -> Bool
```
rew
```
  eq(won,won)=T
  eq(lost,lost)=T
  eq(won,lost)=F
  eq(lost,won)=F
```

sort `LDC`
func
```
  ackrec: ACK -> LDC
  ackmiss,broadsent: -> LDC
```

sort `LDI`

```
func
  good,broadrec: HEADER#DATA -> LDI
  dcrc_err: HEADER -> LDI

sort BOC
func
  release,hold: -> BOC
map
  eq: BOC#BOC -> Bool
rew
  eq(release,release)=T
  eq(hold,hold)=T
  eq(release,hold)=F
  eq(hold,release)=F
```

A.2 The LINK process in μCRL

```
act
  LDreq: NAT#NAT#HEADER#DATA
  LDcon: NAT#LDC
  LDind: NAT#LDI
  LDres: NAT#ACK#BOC

  sPDreq,rPDind: NAT#SIGNAL
  sPAreq: NAT#PAR
  rPAcon: NAT#PAC
  rPCind: NAT

proc

LINK(n:NAT,i:NAT)=
   ( Link0(n,i,void) )

Link0(n:NAT,id:NAT,buffer:SIG_TUPLE)=
(
    sum(dest:NAT,
      sum(h:HEADER,
        sum(d:DATA,
          LDreq(id,dest,h,d).
            Link0(n,id,quadruple(dhead,
                                 sig(dest),
                                 sig(h,crc(h)),
                                 sig(d,crc(d))))
        )
      )
    )
  <| is_void(buffer) |>
    sPAreq(id,fair).Link1(n,id,buffer)
)
+
  sum(p:SIGNAL,
    rPDind(id,p).
      ( Link4(n,id,buffer) <| is_start(p) |> Link0(n,id,buffer) )
  )

Link1(n:NAT,id:NAT,p:SIG_TUPLE)=
  rPAcon(id,won).Link2req(n,id,p)
+
  rPAcon(id,lost).Link0(n,id,p)

Link2req(n:NAT,id:NAT,p:SIG_TUPLE)=
   ( rPCind(id).sPDreq(id,Start).
     rPCind(id).sPDreq(id,first(p)).
     rPCind(id).sPDreq(id,second(p)) ) .
   ( rPCind(id).sPDreq(id,third(p)).
     rPCind(id).sPDreq(id,fourth(p)).
     rPCind(id).sPDreq(id,End) ).
    (
      LDcon(id,broadsent).Link0(n,id,void)
     <| eq(getdest(second(p)),n) |>
      Link3(n,id,void)
    )

Link3(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(p:SIGNAL,
    rPDind(id,p).
      (
        Link3(n,id,buffer)
      <| is_prefix(p) |>
        (
          Link3RA(n,id,buffer)
        <| is_start(p) |>
          (
            LDcon(id,ackmiss).Link0(n,id,buffer)
          <| is_sagap(p) |>
            LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
          )
        )
      )
  )

Link3RA(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(a:SIGNAL,
    rPDind(id,a).
      (
        (
          LDcon(id,ackmiss).Link0(n,id,buffer)
        <| is_sagap(a) |>
          LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
        )
      <| is_physig(a) |>
        Link3RE(n,id,buffer,a)
      )
  )

Link3RE(n:NAT,id:NAT,buffer:SIG_TUPLE,a:SIGNAL)=
  sum(e:SIGNAL,
    rPDind(id,e).
      (
        LDcon(id,ackrec(getack(a))).LinkWSA(n,id,buffer,n)
      <| and(valid_ack(a),is_terminator(e)) |>
        (
          LDcon(id,ackmiss).Link0(n,id,buffer)
        <| is_sagap(e) |>
          LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
        )
      )
  )

Link4(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(dh:SIGNAL,
    rPDind(id,dh).
      (
        (
          Link0(n,id,buffer)
        <| is_sagap(dh) |>
          LinkWSA(n,id,buffer,n)
        )
      <| is_physig(dh) |>
        Link4DH(n,id,buffer)
      )
  )

Link4DH(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(dest:SIGNAL,
    rPDind(id,dest).
      (
        (
          sPAreq(id,immediate).Link4RH(n,id,buffer,id)
        <| eq(getdest(dest),id) |>
          (
            Link4RH(n,id,buffer,n)
          <| eq(getdest(dest),n) |>
            LinkWSA(n,id,buffer,n)
          )
        )
      <| is_dest(dest) |>
        (
          Link0(n,id,buffer)
        <| is_sagap(dest) |>
          LinkWSA(n,id,buffer,n)
        )
      )
  )
```

```
Link4RH(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT)=
  sum(h:SIGNAL,
    rPDind(id,h).
      (
        Link4RD(n,id,buffer,dest,h)
      <| valid_hpart(h) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4RD(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT,h:SIGNAL)=
  sum(d:SIGNAL,
    rPDind(id,d).
      (
        Link4RE(n,id,buffer,dest,h,d)
      <| is_data(d) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4RE(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT,h:SIGNAL,d:SIGNAL)=
  sum(e:SIGNAL,
    rPDind(id,e).
      (
        (
          Link4DRec(n,id,buffer,h,d)
        <| eq(dest,id) |>
          Link4BRec(n,id,buffer,h,d)
        )
      <| is_terminator(e) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4DRec(n:NAT,id:NAT,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
  LDind(id,good(gethead(h),getdata(d))).rPAcon(id,won).Link5(n,id,buffer)
<| eq(getdcrc(d),check) |>
  LDind(id,dcrc_err(gethead(h))).rPAcon(id,won).Link5(n,id,buffer)

Link4BRec(n:NAT,id:NAT,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
  LDind(id,broadrec(gethead(h),getdata(d))).Link0(n,id,buffer)
<| eq(getdcrc(d),check) |>
  Link0(n,id,buffer)

Link5(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(a:ACK,
    sum(b:BOC,
      LDres(id,a,b).Link6(n,id,buffer,sig(a,crc(a)),b)
    )
  )
+
  rPCind(id).sPDreq(id,Prefix).Link5(n,id,buffer)
```

```
Link6(n:NAT,id:NAT,buffer:SIG_TUPLE,p:SIGNAL,b:BOC)=
  ( rPCind(id).sPDreq(id,Start).rPCind(id).sPDreq(id,p) ) .
      ( rPCind(id).
        (
          sPDreq(id,End).Link0(n,id,buffer)
        <| eq(b,release) |>
          sPDreq(id,Prefix).Link7(n,id,buffer)
        )
      )

Link7(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  rPCind(id).sPDreq(id,Prefix).Link7(n,id,buffer)
+
  sum(dest:NAT,
    sum(h:HEADER,
      sum(d:DATA,
        LDreq(id,dest,h,d).
          Link2resp(n,id,buffer,quadruple(dhead,
                                          sig(dest),
                                          sig(h,crc(h)),
                                          sig(d,crc(d))))
      )
    )
  )

Link2resp(n:NAT,id:NAT,buffer:SIG_TUPLE,p:SIG_TUPLE)=
  ( rPCind(id).sPDreq(id,Start).
    rPCind(id).sPDreq(id,first(p)).
    rPCind(id).sPDreq(id,second(p)) ).
    ( rPCind(id).sPDreq(id,third(p)).
      rPCind(id).sPDreq(id,fourth(p)).
      rPCind(id).sPDreq(id,End)).
        ( LDcon(id,broadsent).Link0(n,id,buffer)
             <| eq(getdest(second(p)),n) |>
              Link3(n,id,buffer)
        )

LinkWSA(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT)=
  sum(p:SIGNAL,
    rPDind(id,p).
      (
        Link0(n,id,buffer)
      <| is_sagap(p) |>
        LinkWSA(n,id,buffer,dest)
      )
  )
+
  (
    rPAcon(id,won).rPCind(id).sPDreq(id,End).Link0(n,id,buffer)
  <| eq(dest,id) |>
    delta
  )
```

A.3 The BUS process in μCRL

sort `BoolTABLE`
func
```
  empty: -> BoolTABLE
  btable: NAT#Bool#BoolTABLE -> BoolTABLE
```

map
```
  inita: NAT -> BoolTABLE
  invert: NAT#BoolTABLE -> BoolTABLE
  get: NAT#BoolTABLE -> Bool
  if: Bool#BoolTABLE#BoolTABLE -> BoolTABLE
  eq:BoolTABLE#BoolTABLE->Bool
```
var
```
  n,m : NAT
  b : Bool
bt1,bt2 : BoolTABLE
```
rew
```
  eq(bt1, bt1)=T
  inita(0)=empty
  inita(succ(n))=btable(n,F,inita(n))

  invert(n,empty)=empty
  invert(n,btable(m,b,bt1))=
    if(eq(n,m),
      btable(m,not(b),bt1),
      btable(m,b,invert(n,bt1))
    )
  get(n,btable(m,b,bt1))=if(eq(n,m),b,get(n,bt1))
  get(n,empty)=F
  if(T,bt1,bt2)=bt1
  if(F,bt1,bt2)=bt2
```

map
```
  zero,one,more: BoolTABLE -> Bool
```
var
```
  n : NAT
  bt : BoolTABLE
```
rew
```
  zero(empty)=T
  zero(btable(n,T,bt))=F
  zero(btable(n,F,bt))=zero(bt)
  one(empty)=F
  one(btable(n,T,bt))=zero(bt)
  one(btable(n,F,bt))=one(bt)
  more(bt)=and(not(zero(bt)),not(one(bt)))
```

act

```
  rPAreq: NAT#PAR
  rPDreq,sPDind: NAT#SIGNAL
  sPAcon: NAT#PAC
  sPCind: NAT
  arbresgap
  losesignal
```

proc

```
BUS(n:NAT)=
  BusIdle(n, inita(n))

BusIdle(n:NAT,t:BoolTABLE)=
  sum(id:NAT,
    sum(astat:PAR,
      rPAreq(id,astat).DecideIdle(n,t,id,astat)))
+
  arbresgap.BusIdle(n,inita(n)) <| not(zero(t)) |> delta

DecideIdle(n:NAT,t:BoolTABLE,id:NAT,astat:PAR)=
  ( sPAcon(id,won).BusBusy(n,invert(id,t),inita(n),inita(n),id) )
  <| not(get(id,t)) |>
  ( sPAcon(id,lost).BusIdle(n,t) )

BusBusy(n:NAT,
        t:BoolTABLE,
        next:BoolTABLE,
        destfault:BoolTABLE,
        busy:NAT)=
(
  (
    sPCind(busy).
      sum(p:SIGNAL,
        rPDreq(busy,p).Distribute(n,t,next,destfault,busy,p,0)
      )
  )
  <| lt(busy,n) |>
  (
      SubactionGap(n,t,0)
    <| zero(next) |>
      Resolve(n,t,next,0)
  )
)
+
  sum(j:NAT,
    rPAreq(j,fair).sPAcon(j,lost).BusBusy(n,t,next,destfault,busy)
  )
+
  sum(j:NAT,
    rPAreq(j,immediate).
      ( BusBusy(n,t,invert(j,next),destfault,busy)
          <| not(get(j,next)) |> delta )
  )

SubactionGap(n:NAT,t:BoolTABLE,i:NAT)=
  BusIdle(n,t)
<| eq(i,n) |>
  sPDind(i,subactgap).SubactionGap(n,t,succ(i))

Resolve(n:NAT,t:BoolTABLE,next:BoolTABLE,i:NAT)=
(
  (
    ( sPAcon(i,won).sPCind(i).Resolve(n,t,next,succ(i)) )
  <| get(i,next) |>
    ( tau.Resolve(n,t,next,succ(i)) )
  )
<| lt(i,n) |>
  Resolve2(n,t,next)
)

Resolve2(n:NAT,t:BoolTABLE,next:BoolTABLE)=
(
  sum(j:NAT,
    rPDreq(j,End).
      (
        Resolve2(n,t,invert(j,next))
      <| get(j,next) |>
        delta
      )
  )
<| more(next) |>
  sum(j:NAT,
    sum(p:SIGNAL,
      rPDreq(j,p).
        (
          SubactionGap(n,t,0)
        <| is_end(p) |>
          Distribute(n,t,inita(n),inita(n),j,p,0)
        )
    )
  )
)

Distribute(n:NAT,
           t:BoolTABLE,
           next:BoolTABLE,
           destfault:BoolTABLE,
           busy:NAT,
           p:SIGNAL,
           i:NAT)=
(
  (
    (
      %% Signals can be handed over correctly
      ( sPDind(i,p).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| or(not(is_header(p)),not(get(i,destfault))) |>
              delta )
    +
      %% Destination signals may be corrupted
      ( sum(dest:NAT,
          sPDind(i,sig(dest)).
            Distribute(n,t,next,invert(i,destfault),busy,p,succ(i))
        ) <| is_dest(p) |> delta )
    +
      %% Headers/Data/Acks may be corrupted
      ( sPDind(i,corrupt(p)).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| is_hda(p) |> delta )
    +
      %% Headers/Data/Acks may get lost
      ( losesignal.Distribute(n,t,next,destfault,busy,p,succ(i))
          <| is_hda(p) |> delta )
    +
      %% Packets may be too large
      ( sPDind(i,p).sPDind(i,Dummy).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| is_data(p) |> delta )
    +
      ( rPAreq(i,immediate).
          ( Distribute(n,t,invert(i,next),destfault,busy,p,i)
              <| not(get(i,next)) |> delta ) )
    )
  <| not(eq(i,busy)) |>
    tau.Distribute(n,t,next,destfault,busy,p,succ(i))
  )
<| lt(i,n) |>
  (
    BusBusy(n,t,next,destfault,n)
  <| is_end(p) |>
    BusBusy(n,t,next,destfault,busy)
  )
)
```

A.4 The MAIN process in μCRL

act

```
  PDind,PDreq: NAT#SIGNAL
  PAcon: NAT#PAC
  PAreq: NAT#PAR
  PCind: NAT
```

comm

```
  rPDind|sPDind=PDind
  rPDreq|sPDreq=PDreq
  rPAcon|sPAcon=PAcon
  rPAreq|sPAreq=PAreq
  rPCind|sPCind=PCind

proc

 P1394(n:NAT)=
   hide({PDind, PDreq, PAcon, PAreq, PCind, arbresgap,losesignal},
     encap( {rPDind, sPDind, rPDreq, sPDreq, rPAcon,
             sPAcon, rPAreq, sPAreq, rPCind, sPCind},
         BUS(2) || LINK(2,0) || LINK(2,1)
       )
     )
```

% note: for 3 links, use BUS(3) || LINK(3,0) || LINK(3,1) || LINK(3,2), etc.

```
init P1394(2)
```

B Formal model in mCRL2

B.1 Types and functions in mCRL2

```
sort CHECK = struct bottom | check;

sort DATA = struct d1 | d2;

map crc : DATA -> CHECK;
eqn crc(d1)=check;
     crc(d2)=check;

sort HEADER = struct h1 | h2;

map crc : HEADER -> CHECK;
eqn crc(h1)=check;
     crc(h2)=check;

sort ACK = struct a1 | a2;

map crc : ACK -> CHECK;
eqn crc(a1)=check;
     crc(a2)=check;

sort SIGNAL = struct sig(getdest:Nat) ? is_dest |
                     sig(gethead:HEADER,gethcrc:CHECK) ? is_header |
                     sig(getdata:DATA,getdcrc:CHECK) ? is_data |
                     sig(getack:ACK,getacrc:CHECK) ? is_ack |
                     Start ? is_start |
                     End ? is_end |
                     Prefix ? is_prefix |
                     subactgap ? is_sagap |
                     dhead ? is_dhead |
                   Dummy ? is_dummy;

map is_physig,is_terminator : SIGNAL -> Bool;
    getcrc : SIGNAL -> CHECK;
var s : SIGNAL;
eqn is_physig(s) = is_start(s) || is_end(s) || is_prefix(s) || is_sagap(s);
    is_terminator(s)=is_end(s) || is_prefix(s);
    getcrc(s)=if(is_header(s),gethcrc(s),
              if(is_data(s),getdcrc(s),
              if(is_ack(s),getacrc(s),
                         bottom)));

map is_hda : SIGNAL -> Bool;
    valid_hpart, valid_ack : SIGNAL -> Bool;
var s : SIGNAL;
eqn is_hda(s)=is_header(s) || is_data(s) || is_ack(s);
    valid_ack(s)=if(is_ack(s),getacrc(s)==check,false);
    valid_hpart(s)=if(is_header(s),gethcrc(s)==check,false);

map corrupt : SIGNAL -> SIGNAL;
var h : HEADER;
    d : DATA;
    a : ACK;
    c : CHECK;
eqn corrupt(sig(h,c)) = sig(h,bottom);
    corrupt(sig(d,c)) = sig(d,bottom);
    corrupt(sig(a,c)) = sig(a,bottom);

sort SIG_TUPLE =
      struct quadruple (first:SIGNAL,
                        second:SIGNAL,
                        third:SIGNAL,
                        fourth:SIGNAL)
       | void ? is_void;

sort PAR = struct fair | immediate;

sort PAC = struct won | lost;

sort LDC = struct ackrec(ACK)
           | ackmiss
           | broadsent;

sort LDI = struct good (HEADER,DATA)
           | broadrec (HEADER,DATA)
           | dcrc_err (HEADER);

sort BOC = struct release | hold;
```

B.2 The LINK process in mCRL2

```
act
  LDreq : Nat#Nat#HEADER#DATA;
  LDcon : Nat#LDC;
  LDind : Nat#LDI;
  LDres : Nat#ACK#BOC;

  sPDreq,rPDind : Nat#SIGNAL;
  sPAreq : Nat#PAR;
  rPAcon : Nat#PAC;
  rPCind : Nat;

proc LINK(n:Nat,i:Nat)=Link0(n,i,void);

     Link0(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        is_void(buffer) ->
           ( sum dest:Nat,h:HEADER,d:DATA.
                ( dest<=n) -> LDreq(id,dest,h,d).
                     Link0(n,id,quadruple(dhead,
                                 sig(dest),
                                 sig(h,crc(h)),
                                 sig(d,crc(d))))<>delta) <>
             sPAreq(id,fair).Link1(n,id,buffer) +
        sum p:SIGNAL.
           rPDind(id,p).
              (is_start(p) -> Link4(n,id,buffer) <> Link0(n,id,buffer));

     Link1(n:Nat,id:Nat,p:SIG_TUPLE)=
        rPAcon(id,won).Link2req(n,id,p) +
        rPAcon(id,lost).Link0(n,id,p);

     Link2req(n:Nat,id:Nat,p:SIG_TUPLE)=
        rPCind(id).sPDreq(id,Start).
        rPCind(id).sPDreq(id,first(p)).
        rPCind(id).sPDreq(id,second(p)) .
        rPCind(id).sPDreq(id,third(p)).
        rPCind(id).sPDreq(id,fourth(p)).
        rPCind(id).sPDreq(id,End).
        ( (getdest(second(p))==n) ->
            LDcon(id,broadsent).Link0(n,id,void) <>
            Link3(n,id,void));

     Link3(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        sum p:SIGNAL.
           rPDind(id,p).
           ( is_prefix(p) -> Link3(n,id,buffer) <>
           ( is_start(p) -> Link3RA(n,id,buffer) <>
           ( is_sagap(p) -> LDcon(id,ackmiss).Link0(n,id,buffer) <>
                            LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
           )));

     Link3RA(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        sum a:SIGNAL.
```

```
      rPDind(id,a).
      ( is_sagap(a) -> LDcon(id,ackmiss).Link0(n,id,buffer) <>
        ( is_physig(a) -> LDcon(id,ackmiss).LinkWSA(n,id,buffer,n) <>
                         Link3RE(n,id,buffer,a)));

Link3RE(n:Nat,id:Nat,buffer:SIG_TUPLE,a:SIGNAL)=
   sum e:SIGNAL.
      rPDind(id,e).
      ((valid_ack(a) && is_terminator(e)) ->
              LDcon(id,ackrec(getack(a))).LinkWSA(n,id,buffer,n) <>
        ( is_sagap(e) ->
              LDcon(id,ackmiss).Link0(n,id,buffer) <>
              LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
      ) );

Link4(n:Nat,id:Nat,buffer:SIG_TUPLE)=
   sum dh:SIGNAL.
      rPDind(id,dh).
      ( is_physig(dh) ->
        ( is_sagap(dh) ->
           Link0(n,id,buffer) <>
           LinkWSA(n,id,buffer,n)) <>
        Link4DH(n,id,buffer));

Link4DH(n:Nat,id:Nat,buffer:SIG_TUPLE)=
   sum dest:SIGNAL.rPDind(id,dest).
      ( is_dest(dest) ->
        ( (getdest(dest)==id) ->
             sPAreq(id,immediate).Link4RH(n,id,buffer,id) <>
             ( (getdest(dest)==n) ->
                 Link4RH(n,id,buffer,n) <>
                 LinkWSA(n,id,buffer,n)
             )
        ) <>
        ( is_sagap(dest) ->
             Link0(n,id,buffer) <>
             LinkWSA(n,id,buffer,n)
      ) );

Link4RH(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat)=
   sum h:SIGNAL.rPDind(id,h).
      ( valid_hpart(h) ->
           Link4RD(n,id,buffer,dest,h) <>
           LinkWSA(n,id,buffer,dest)
      );

Link4RD(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat,h:SIGNAL)=
   sum d:SIGNAL.
      rPDind(id,d).
        ( is_data(d) ->
             Link4RE(n,id,buffer,dest,h,d) <>
             LinkWSA(n,id,buffer,dest)
        );

Link4RE(n,id:Nat,buffer:SIG_TUPLE,dest:Nat,h:SIGNAL,d:SIGNAL)=
   sum e:SIGNAL.
      rPDind(id,e).
        ( is_terminator(e) ->
             ( (dest==id) ->
                  Link4DRec(n,id,buffer,h,d) <>
                  Link4BRec(n,id,buffer,h,d)
             ) <>
             LinkWSA(n,id,buffer,dest)
        );

Link4DRec(n:Nat,id:Nat,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
   (getcrc(d)==check) ->
      LDind(id,good(gethead(h),getdata(d))).rPAcon(id,won).Link5(n,id,buffer)
      <>
      LDind(id,dcrc_err(gethead(h))).rPAcon(id,won).Link5(n,id,buffer);

Link4BRec(n:Nat,id:Nat,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
   (getcrc(d)==check) ->
      LDind(id,broadrec(gethead(h),getdata(d))).Link0(n,id,buffer) <>
      Link0(n,id,buffer);

Link5(n,id:Nat,buffer:SIG_TUPLE)=
   sum a:ACK,b:BOC.LDres(id,a,b).Link6(n,id,buffer,sig(a,crc(a)),b) +
   rPCind(id).sPDreq(id,Prefix).Link5(n,id,buffer);

Link6(n:Nat,id:Nat,buffer:SIG_TUPLE,p:SIGNAL,b:BOC)=
   rPCind(id).sPDreq(id,Start).rPCind(id).sPDreq(id,p).rPCind(id).
      ( (b==release) ->
           sPDreq(id,End).Link0(n,id,buffer) <>
           sPDreq(id,Prefix).Link7(n,id,buffer)
      );

Link7(n,id:Nat,buffer:SIG_TUPLE)=
   rPCind(id).sPDreq(id,Prefix).Link7(n,id,buffer) +
   sum dest:Nat,h:HEADER,d:DATA. (dest<=n) ->
        LDreq(id,dest,h,d). Link2resp(n,id,buffer,
                 quadruple(dhead,sig(dest),sig(h,crc(h)),sig(d,crc(d))))<>delta;

Link2resp(n:Nat,id:Nat,buffer:SIG_TUPLE,p:SIG_TUPLE)=
   rPCind(id).sPDreq(id,Start).
   rPCind(id).sPDreq(id,first(p)).
   rPCind(id).sPDreq(id,second(p)).
   rPCind(id).sPDreq(id,third(p)).
   rPCind(id).sPDreq(id,fourth(p)).
   rPCind(id).sPDreq(id,End).
   ( (getdest(second(p))==n) ->
        LDcon(id,broadsent).Link0(n,id,buffer) <>
        Link3(n,id,buffer)
   );
```

```
    LinkWSA(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat)=
       sum p:SIGNAL.rPDind(id,p).
          ( is_sagap(p) ->
              Link0(n,id,buffer) <>
              LinkWSA(n,id,buffer,dest)
          ) +
       (dest==id) -> rPAcon(id,won).rPCind(id).sPDreq(id,End).Link0(n,id,buffer)<>delta;
```

B.3 The BUS process in mCRL2

```
sort BoolTABLE = List(struct pair(Nat,getbool:Bool));

map inita : Nat -> BoolTABLE;
    invert : Nat#BoolTABLE -> BoolTABLE;
    get : Nat#BoolTABLE -> Bool;
var n,m : Nat;
    b : Bool;
    bt1,bt2 : BoolTABLE;
eqn inita(0)=[];
    n>0 -> inita(n)=pair(Int2Nat(n-1),false)|>inita(Int2Nat(n-1));

    invert(n,[])=[];
    invert(n,pair(m,b)|>bt1)=
       if(n==m,pair(m,!b)|>bt1,pair(m,b)|>invert(n,bt1));
    get(n,[])=false;
    get(n,pair(m,b)|>bt1)=if(n==m,b,get(n,bt1));

map zero,one,more: BoolTABLE -> Bool;
var n : Nat;
    bt : BoolTABLE;
eqn zero([])=true;
    zero(pair(n,true)|>bt)=false;
    zero(pair(n,false)|>bt)=zero(bt);
    one([])=false;
    one(pair(n,true)|>bt)=zero(bt);
    one(pair(n,false)|>bt)=one(bt);
    more(bt)=!zero(bt) && !one(bt);

act rPAreq: Nat#PAR;
    rPDreq,sPDind: Nat#SIGNAL;
    sPAcon: Nat#PAC;
    sPCind: Nat;
    arbresgap;
    losesignal;
    internal;

proc BUS(n:Nat)=BusIdle(n, inita(n));

    BusIdle(n:Nat,t:BoolTABLE)=
       sum id:Nat,astat:PAR.(id<=n) ->
```

```
      rPAreq(id,astat).DecideIdle(n,t,id,astat)<>delta +
      !zero(t)->arbresgap.BusIdle(n,inita(n))<>delta;

DecideIdle(n:Nat,t:BoolTABLE,id:Nat,astat:PAR)=
   (!get(id,t)) ->
     sPAcon(id,won).BusBusy(n,invert(id,t),inita(n),inita(n),id) <>
     sPAcon(id,lost).BusIdle(n,t);

BusBusy(n:Nat,t,next,destfault:BoolTABLE,busy:Nat)=
   (busy<n) ->
      ( sPCind(busy).
           (sum p:SIGNAL.rPDreq(busy,p).Distribute(n,t,next,destfault,busy,p,0))
      ) <>
      ( zero(next) ->
           SubactionGap(n,t,0) <>
             Resolve(n,t,next,0)
      ) +
  sum j:Nat.(j<=n) ->
     rPAreq(j,fair).sPAcon(j,lost).BusBusy(n,t,next,destfault,busy)<>delta +
  sum j:Nat.(j<=n) -> rPAreq(j,immediate).
     (!get(j,next) -> BusBusy(n,t,invert(j,next),destfault,busy)<>delta)<>delta;

SubactionGap(n:Nat,t:BoolTABLE,i:Nat)=
   (i==n) ->
      BusIdle(n,t) <>
      sPDind(i,subactgap).SubactionGap(n,t,i+1);

Resolve(n:Nat,t,next:BoolTABLE,i:Nat)=
   (i<n) ->
   (get(i,next) ->
       sPAcon(i,won).sPCind(i).Resolve(n,t,next,i+1) <>
       internal.Resolve(n,t,next,i+1)
   ) <>
   Resolve2(n,t,next);

Resolve2(n:Nat,t:BoolTABLE,next:BoolTABLE)=
   more(next) ->
      (sum j:Nat.(j<=n) -> rPDreq(j,End).(get(j,next) ->
      Resolve2(n,t,invert(j,next))<>delta)<>delta) <>
      (sum j:Nat,p:SIGNAL.(j<=n) ->
         rPDreq(j,p).
            (is_end(p) ->
                SubactionGap(n,t,0) <>
                Distribute(n,t,inita(n),inita(n),j,p,0)
      )<>delta);

Distribute(n:Nat,t,next,destfault:BoolTABLE,busy:Nat,p:SIGNAL,i:Nat)=
   (i<n) ->
   ( (i!=busy) ->
     ( %% Signals can be handed over correctly
       (!is_header(p) || !get(i,destfault)) ->
          sPDind(i,p).Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Destination signals may be corrupted
       sum dest:Nat.(is_dest(p) && dest<=n) ->
          sPDind(i,sig(dest)).
             Distribute(n,t,next,invert(i,destfault),busy,p,i+1)<>delta +
       %% Headers/Data/Acks may be corrupted
       is_hda(p) ->
          sPDind(i,corrupt(p)).
             Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Headers/Data/Acks may get lost
       is_hda(p) ->
          losesignal.Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Packets may be too large
       is_data(p) ->
          sPDind(i,p).sPDind(i,Dummy).
             Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       (!get(i,next)) ->
          rPAreq(i,immediate).
             Distribute(n,t,invert(i,next),destfault,busy,p,i)<>delta
    ) <>
    %% i==busy
    internal.Distribute(n,t,next,destfault,busy,p,i+1)
  ) <>
  %% i>=n
  ( is_end(p) ->
       BusBusy(n,t,next,destfault,n) <>
       BusBusy(n,t,next,destfault,busy)
  );
```

B.4 The MAIN process in mCRL2

```
act
  cPDreq,cPDind : Nat#SIGNAL;
  cPAreq : Nat#PAR;
  cPAcon : Nat#PAC;
  cPCind : Nat;

proc P1394(n:Nat)=
        allow({LDreq,LDcon,LDind,LDres},
           hide({arbresgap,losesignal,internal,cPDind,cPDreq,cPAcon,cPAreq,cPCind},
              comm({rPDind|sPDind->cPDind,rPDreq|sPDreq->cPDreq,rPAcon|sPAcon->cPAcon,
                    rPAreq|sPAreq->cPAreq,rPCind|sPCind->cPCind},
                  allow({LDreq,LDcon,LDind,LDres,arbresgap,losesignal,internal,
                              rPDind|sPDind,rPDreq|sPDreq,rPAcon|sPAcon,
                    rPAreq|sPAreq,rPCind|sPCind},
                          BUS(2) || LINK(2,0) || LINK(2,1)))));
```

% note: for 3 links, use BUS(3) || LINK(3,0) || LINK(3,1) || LINK(3,2), etc.

```
init P1394(2);
```

C Formal model in LOTOS

C.1 Types and functions in LOTOS

```
type CHECK is Boolean
   sorts
      CHECK
   opns
      bottom (*! constructor *) : -> CHECK
      check (*! constructor *) : -> CHECK
      eq : CHECK, CHECK -> Bool
   eqns
      forall x, y : CHECK
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
endtype

(*------------------------------------------------------------------------*)

type DATA is CHECK
   sorts
      DATA
   opns
      d1 (*! constructor *) : -> DATA
      (* for verification, this type is restricted to a single value *)
      (* d2 {*! constructor *} : -> DATA *)
      crc : DATA -> CHECK
      eq : DATA, DATA -> Bool
   eqns
      forall x, y : DATA
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*------------------------------------------------------------------------*)

type HEADER is CHECK
   sorts
      HEADER
   opns
      h1 (*! constructor *) : -> HEADER
      (* for verification, this type is restricted to a single value *)
      (* h2 {*! constructor *} : -> HEADER *)
      crc : HEADER -> CHECK
      eq : HEADER, HEADER -> Bool
   eqns
      forall x, y : HEADER
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*------------------------------------------------------------------------*)

type ACK is CHECK
   sorts
      ACK
   opns
      a1 (*! constructor *) : -> ACK
      (* for verification, this type is restricted to a single value *)
      (* a2 {*! constructor *} : -> ACK *)
      crc : ACK -> CHECK
      eq : ACK, ACK -> Bool
   eqns
      forall x, y : ACK
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*------------------------------------------------------------------------*)

type BOC is CHECK
   sorts
      BOC
   opns
      release (*! constructor *),
      hold (*! constructor *),
      no_op (*! constructor *) : -> BOC
      eq : BOC, BOC -> Bool
   eqns
      forall x, y : BOC
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
endtype

(*------------------------------------------------------------------------*)

type PHY_AREQ is CHECK
   sorts
      PHY_AREQ
   opns
      fair (*! constructor *),
      immediate (*! constructor *) : -> PHY_AREQ
      eq : PHY_AREQ, PHY_AREQ -> Bool
   eqns
     forall x, y : PHY_AREQ
     ofsort Bool
        eq (x, x) = true;
        (* otherwise *) eq (x, y) = false;
endtype

(*------------------------------------------------------------------------*)

type PHY_ACONF is CHECK
   sorts
      PHY_ACONF
   opns
      won (*! constructor *),
      lost (*! constructor *) : -> PHY_ACONF
      eq : PHY_ACONF, PHY_ACONF -> Bool
   eqns
     forall x, y : PHY_ACONF
     ofsort Bool
        eq (x, x) = true;
        (* otherwise *) eq (x, y) = false;
endtype

(*------------------------------------------------------------------------*)

type SIGNAL is ACK, CHECK, DATA, HEADER, NaturalNumber
   sorts
      SIGNAL
   opns
      destsig (*! constructor *) : Nat -> SIGNAL
      headsig (*! constructor *) : HEADER, CHECK -> SIGNAL
      datasig (*! constructor *) : DATA, CHECK -> SIGNAL
      acksig (*! constructor *) : ACK, CHECK -> SIGNAL
      dhead (*! constructor *) : -> SIGNAL
      Start (*! constructor *) : -> SIGNAL
      End (*! constructor *) : -> SIGNAL
      Prefix (*! constructor *) : -> SIGNAL
      subactgap (*! constructor *) : -> SIGNAL
      Dummy (*! constructor *) : -> SIGNAL
      is_dest, is_header, is_data, is_ack, is_physig : SIGNAL -> Bool
      valid_hpart, valid_ack : SIGNAL -> Bool
      getdest : SIGNAL -> Nat
      getdcrc : SIGNAL -> CHECK
      getdata : SIGNAL -> DATA
      gethead : SIGNAL -> HEADER
      getack : SIGNAL -> ACK
      corrupt : SIGNAL -> SIGNAL
      eq : SIGNAL, SIGNAL -> Bool
   eqns
      forall n : Nat, c : CHECK, h : HEADER, d : DATA, a : ACK, s, s1, s2 : SIGNAL
      ofsort Bool
```

```
         is_dest (destsig (n)) = true;
         (* otherwise *) is_dest (s) = false;
         is_header (headsig (h, c)) = true;
         (* otherwise *) is_header (s) = false;
         is_data (datasig (d, c)) = true;
         (* otherwise *) is_data (s) = false;
         is_ack (acksig (a, c)) = true;
         (* otherwise *) is_ack (s) = false;
         is_physig (Start) = true;
         is_physig (End) = true;
         is_physig (Prefix) = true;
         is_physig (subactgap) = true;
         (* otherwise *) is_physig (s) = false;
         valid_ack (acksig (a, c)) = eq (c, check);
         (* otherwise *) valid_ack (s) = false;
         valid_hpart (headsig (h, c)) = eq (c, check);
         (* otherwise *) valid_hpart (s) = false;
      ofsort Nat
         getdest (destsig (n)) = n;
         (* otherwise getdest (s) is undefined *)
      ofsort HEADER
         gethead (headsig (h, c)) = h;
         (* otherwise gethead (s) is undefined *)
      ofsort CHECK
         getdcrc (datasig (d, c)) = c;
         (* otherwise getdcrc (s) is undefined *)
      ofsort DATA
         getdata (datasig (d, c)) = d;
         (* otherwise getdata (s) is undefined *)
      ofsort ACK
         getack (acksig (a, c)) = a;
         (* otherwise getack (s) is undefined *)
      ofsort SIGNAL
         corrupt (headsig (h, c)) = headsig (h, bottom);
         corrupt (datasig (d, c)) = datasig (d, bottom);
         corrupt (acksig (a, c)) = acksig (a, bottom);
      ofsort Bool
         eq (s1, s1) = true;
         (* otherwise *) eq (s1, s2) = false;
endtype

(*------------------------------------------------------------------------*)

type SIG_TUPLE is Boolean, SIGNAL
   sorts
      SIG_TUPLE
   opns
      quadruple (*! constructor *) : SIGNAL, SIGNAL, SIGNAL, SIGNAL -> SIG_TUPLE
      void (*! constructor *) : -> SIG_TUPLE
      first, second, third, fourth : SIG_TUPLE -> SIGNAL
      is_void : SIG_TUPLE -> Bool
   eqns
```

```
      forall s1, s2, s3, s4 : SIGNAL
      ofsort SIGNAL
         first (quadruple (s1, s2, s3, s4)) = s1;
         second (quadruple (s1, s2, s3, s4)) = s2;
         third (quadruple (s1, s2, s3, s4)) = s3;
         fourth (quadruple (s1, s2, s3, s4)) = s4;
      ofsort Bool
         is_void (void) = true;
         is_void (quadruple (s1, s2, s3, s4)) = false;
endtype

(*----------------------------------------------------------------------*)

type LIN_DCONF is ACK
   sorts
      LIN_DCONF
   opns
      ackrec (*! constructor *) : ACK -> LIN_DCONF
      ackmiss (*! constructor *),
      broadsent (*! constructor *) : -> LIN_DCONF
endtype

(*----------------------------------------------------------------------*)

type LIN_DIND is Boolean, DATA, HEADER
   sorts
      LIN_DIND
   opns
      good (*! constructor *),
      broadrec (*! constructor *) : HEADER, DATA -> LIN_DIND
      dcrc_err (*! constructor *) : HEADER -> LIN_DIND
      is_broadrec : LIN_DIND -> Bool
   eqns
      forall h: HEADER, d: DATA, xind: LIN_DIND
      ofsort Bool
         is_broadrec (broadrec (h, d)) = true;
         (* otherwise *) is_broadrec (xind) = false;
endtype

(*----------------------------------------------------------------------*)

type BoolTABLE is Boolean, NaturalNumber
   sorts
      BoolTABLE
   opns
      empty (*! constructor *) : -> BoolTABLE
      btable (*! constructor *) : Nat, Bool, BoolTABLE -> BoolTABLE
      init : Nat -> BoolTABLE
      invert : Nat, BoolTABLE -> BoolTABLE
      get : Nat, BoolTABLE -> Bool
      zero, one, more : BoolTABLE -> Bool
   eqns
```

```
      forall n, n1, n2 : Nat, b : Bool, t : BoolTABLE
      ofsort BoolTABLE
         init (0) = empty;
         init (Succ (n)) = btable (n, false, init (n));
         invert (n, empty) = empty;
         n1 eq n2 => invert (n1, btable (n2, b, t)) = btable (n2, not (b), t);
         n1 ne n2 => invert (n1, btable (n2, b, t)) = btable (n2, b, invert (n1, t));
      ofsort Bool
         (* get (n, empty) is undefined *)
         n1 eq n2 => get (n1, btable (n2, b, t)) = b;
         n1 ne n2 => get (n1, btable (n2, b, t)) = get (n1, t);
      ofsort Bool
         zero (empty) = true;
         zero (btable (n, true, t)) = false;
         zero (btable (n, false, t)) = zero (t);
         one (empty) = false;
         one (btable (n, true, t)) = zero (t);
         one (btable (n, false, t)) = one (t);
         more (t) = not (zero (t)) and not (one (t));
endtype

(*------------------------------------------------------------------------*)

type Version is
   sorts
      Version
   opns
      ko (*! constructor *),
      ok (*! constructor *) : -> Version
endtype

(*------------------------------------------------------------------------*)

type Scenario is Boolean, Natural
   sorts
      Scenario
   opns
      scenario_1 (*! constructor *),
      scenario_2 (*! constructor *),
      scenario_3_2 (*! constructor *),
      scenario_3_3 (*! constructor *),
      scenario_3_4 (*! constructor *) : -> Scenario
      _eq_ : Scenario, Scenario -> Bool
   eqns
      forall s1, s2: Scenario
      ofsort Bool
         s1 eq s1 = true;
         (* otherwise *) s1 eq s2 = false;
endtype
```

C.2 The LINK process in LOTOS

```
process Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat) : noexit :=
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id, void)
endproc

(* ---------------------------------------------------------------------- *)

process Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   [is_void (buffer)] ->
       LDreq !id ?dest: Nat ?h: HEADER ?d: DATA;
       Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, quadruple (dhead,
                                 destsig (dest),
                                 headsig (h, crc (h)),
                                 datasig (d, crc (d))))
   []
   [not (is_void (buffer))] ->
       PAreq !id !fair;
       Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   []
   PDind !id ?p: SIGNAL;
   (
   [eq (p, Start)] ->
      Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [not (eq (p, Start))] ->
       Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   )
endproc

(* ---------------------------------------------------------------------- *)

process Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PAcon !id !won;
   Link2req [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   PAcon !id !lost;
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
          (n, id, buffer)
endproc

(* ---------------------------------------------------------------------- *)

process Link2req [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
```

```
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !first (buffer);
   PCind !id;
   PDreq !id !second (buffer);
   PCind !id;
   PDreq !id !third (buffer);
   PCind !id;
   PDreq !id !fourth (buffer);
   PCind !id;
   PDreq !id !End;
   (
   [getdest (second (buffer)) eq n] ->
      LDcon !id !broadsent;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, void)
   []
   [getdest (second (buffer)) ne n] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, void)
   )
endproc

(* ---------------------------------------------------------------------- *)

process Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?p: SIGNAL;
   (
   [eq (p, Prefix)] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [eq (p, Start)] ->
      Link3RA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   []
   [eq (p, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (p, Prefix) or eq (p, Start) or eq (p, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
   )
endproc

(* ---------------------------------------------------------------------- *)
```

```
process Link3RA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
           (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
  PDind !id ?a: SIGNAL;
  (
  [is_physig (a)] ->
    (
    [eq (a, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
    []
    [not (eq (a, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
    )
  []
  [not (is_physig (a))] ->
    Link3RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, a)
  )
endproc

(* ---------------------------------------------------------------------- *)

process Link3RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
           (n, id: Nat, buffer: SIG_TUPLE, a: SIGNAL) : noexit :=
  PDind !id ?e: SIGNAL;
  (
  [valid_ack (a) and (eq (e, End) or eq (e, Prefix))] ->
    LDcon !id !ackrec (getack (a));
    LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, n)
  []
  [not (valid_ack (a) and (eq (e, End) or eq (e, Prefix)))] ->
    (
    [eq (e, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
    []
    [not (eq (e, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
    )
  )
endproc

(* ---------------------------------------------------------------------- *)
```

```
process Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?dh: SIGNAL;
   (
   [is_physig (dh)] ->
      (
      [eq (dh, subactgap)] ->
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      []
      [not (eq (dh, subactgap))] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   []
   [not (is_physig (dh))] ->
      Link4DH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   )
endproc

(* ------------------------------------------------------------------------ *)

process Link4DH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?dest: SIGNAL;
   (
   [is_dest (dest)] ->
      (
      [getdest (dest) eq id] ->
         PAreq !id !immediate;
         Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, id)
      []
      [getdest (dest) eq n] ->
         Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      []
      [(getdest (dest) ne n) and (getdest (dest) ne id)] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   []
   [not (is_dest (dest))] ->
      (
      [eq (dest, subactgap)] ->
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      []
      [not (eq (dest, subactgap))] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   )
endproc

(*------------------------------------------------------------------------*)

process Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) : noexit :=
   PDind !id ?h: SIGNAL;
   (
   [valid_hpart (h)] ->
      Link4RD [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest, h)
   []
   [not (valid_hpart (h))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest)
   )
endproc

(*------------------------------------------------------------------------*)

process Link4RD [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE, dest: Nat, h: SIGNAL) : noexit :=
   PDind !id ?d: SIGNAL;
   (
   [is_data (d)] ->
      Link4RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest, h, d)
   []
   [not (is_data (d))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest)
   )
endproc

(*------------------------------------------------------------------------*)

process Link4RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE, dest: Nat, h: SIGNAL, d: SIGNAL)
              : noexit :=
   PDind !id ?e: SIGNAL;
   (
   [eq (e, End) or eq (e, Prefix)] ->
      (
      [dest eq id] ->
         Link4DRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, h, d)
      []
      [dest ne id] ->
         Link4BRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, h, d)
      )
   []
   [not (eq (e, End) or eq (e, Prefix))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer, dest)
   )
endproc

(* ------------------------------------------------------------------------ *)

process Link4DRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id: Nat, buffer: SIG_TUPLE, h: SIGNAL, d: SIGNAL) : noexit :=
   [eq (getdcrc (d), check)]->
      LDind !id !good (gethead (h), getdata (d));
      PAcon !id !won;
      Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [not (eq (getdcrc (d), check))] ->
      LDind !id !dcrc_err (gethead (h));
      PAcon !id !won;
      Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
endproc

(* ------------------------------------------------------------------------ *)

process Link4BRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id: Nat, buffer: SIG_TUPLE, h: SIGNAL, d: SIGNAL) : noexit :=
   [eq (getdcrc (d), check)] ->
      LDind !id !broadrec (gethead (h), getdata (d));
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [not (eq (getdcrc (d), check))] ->
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
endproc

(* ------------------------------------------------------------------------ *)

process Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   LDres !id ?a: ACK ?b: BOC;
   Link6 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer, acksig (a, crc (a)), b)
   []
   PCind !id;
   PDreq !id !Prefix;
   Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer)
endproc
```

```
(* ------------------------------------------------------------------------ *)

process Link6 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id: Nat, buffer: SIG_TUPLE, p: SIGNAL, b: BOC) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !p;
   PCind !id;
   (
   [eq (b, release)] ->
      PDreq !id !End;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (b, release))] ->
      PDreq !id !Prefix;
      Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   )
endproc

(* ------------------------------------------------------------------------ *)

process Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Prefix;
   Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer)
   []
   LDreq !id ?dest: Nat ?h: HEADER ?d: DATA;
   Link2resp [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, quadruple (dhead,
                                       destsig (dest),
                                       headsig (h, crc (h)),
                                       datasig (d, crc (d))))
endproc

(* ------------------------------------------------------------------------ *)

process Link2resp [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, p: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !first (p);
   PCind !id;
   PDreq !id !second (p);
   PCind !id;
   PDreq !id !third (p);
   PCind !id;
   PDreq !id !fourth (p);
   PCind !id;
   PDreq !id !End;
   (
   [getdest (second (p)) eq n] ->
      LDcon !id !broadsent;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [getdest (second (p)) ne n] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   )
endproc

(* ---------------------------------------------------------------------- *)

process LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) : noexit :=
   PDind !id ?p: SIGNAL;
   (
   [eq (p, subactgap)] ->
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [not (eq (p, subactgap))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer, dest)
   )
   []
   [dest eq id] ->
      PAcon !id !won;
      PCind !id;
      PDreq !id !End;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
endproc
```

C.3 The BUS process in LOTOS

```
process Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
            (n: Nat) : noexit :=
   BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
endproc

(* ---------------------------------------------------------------------- *)

process BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n: Nat, t: BoolTABLE) : noexit :=
   PAreq ?id: Nat ?astat: PHY_AREQ [id lt n];
   DecideIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t, id,
              astat)
   []
   [not (zero(t))] ->
      arbresgap;
      BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
endproc

(* ------------------------------------------------------------------------ *)

process DecideIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n: Nat, t: BoolTABLE, id: Nat, astat: PHY_AREQ) : noexit :=
   [get (id, t) eq false] ->
      PAcon !id !won;
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n,
              invert (id, t), init (n), init (n), id)
   []
   [get (id, t) eq true] ->
      PAcon !id !lost;
      BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
endproc

(* ------------------------------------------------------------------------ *)

process BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n: Nat, t: BoolTABLE, next: BoolTABLE, destfault: BoolTABLE,
                busy: Nat) : noexit :=
   [busy lt n] ->
      PCind !busy;
      PDreq !busy ?p: SIGNAL;
      Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, next, destfault, busy, p, 0)
   []
   [not (busy lt n)] ->
      (
      [zero (next)] ->
         SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, 0)
      []
      [not (zero (next))] ->
          Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                  (n, t, next, 0)
      )
   []
   PAreq ?j: Nat !fair [j lt n];
   PAcon !j !lost;
   BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
           (n, t, next, destfault, busy)
   []
   PAreq ?j: Nat !immediate [not (get (j, next)) and (j lt n)];
   BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
```

```
          (n, t, invert (j, next), destfault, busy)
endproc

(* ---------------------------------------------------------------------- *)

process SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                     (n: Nat, t: BoolTABLE, j: Nat) : noexit :=
   [j eq n] ->
     BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
   []
   [j ne n] ->
     PDind !j !subactgap;
     SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                  (n, t, succ (j))
endproc

(* ---------------------------------------------------------------------- *)

process Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n: Nat, t: BoolTABLE, next: BoolTABLE, j: Nat) : noexit :=
   [j lt n] ->
     (
     [get (j, next) eq true] ->
       PAcon !j !won;
       PCind !j;
       Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
               (n, t, next, succ (j))
     []
     [get (j, next) eq false] ->
       Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
               (n, t, next, succ (j))
     )
   []
   [not (j lt n)] ->
     Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next)
endproc

(* ---------------------------------------------------------------------- *)

process Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n: Nat, t: BoolTABLE, next: BoolTABLE) : noexit :=
   [more (next)] ->
     PDreq ?j: Nat !End [get (j, next) and (j lt n)];
     Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, invert (j, next))
   []
   [not (more (next))] ->
     PDreq ?j: Nat ?p: SIGNAL [j lt n];
     (
     [eq (p, End)] ->
       SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
```

```
                    (n, t, 0)
     []
     [not (eq (p, End))] ->
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, init (n), init (n), j, p, 0)
     )
endproc

(* ------------------------------------------------------------------------ *)

process Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n: Nat, t, next, destfault: BoolTABLE, busy: Nat, p: SIGNAL,
                   j: Nat) : noexit :=
   [j lt n] ->
     (
     [j ne busy] ->
        (
        [not (is_header (p)) or not (get (j, destfault))] ->
           PDind !j !p;
           Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, next, destfault, busy, p, succ (j))
        []
        [is_dest (p)] ->
           (
           choice dest: Nat []
              PDind !j !destsig (dest);
              Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                         (n, t, next, invert (j, destfault), busy, p, succ (j))
           )
        []
        [is_header (p) or (is_data (p) or is_ack (p))] ->
           PDind !j !corrupt (p);
           Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, next, destfault, busy, p, succ (j))
        []
        [is_header (p) or (is_data (p) or is_ack (p))] ->
           losesignal;
           Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, next, destfault, busy, p, succ (j))
        []
        [is_data (p)] ->
           PDind !j !p;
           PDind !j !Dummy;
           Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, next, destfault, busy, p, succ (j))
        []
        PAreq !j !immediate [not (get (j, next))];
           Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, invert (j, next), destfault, busy, p, j)
        )
     []
     [j eq busy] ->
```

```
      Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n, t, next, destfault, busy, p, succ (j))
    )
  []
  [not (j lt n)] ->
    (
    [eq (p, End)] ->
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, destfault, n)
    []
    [not (eq (p, End))] ->
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, destfault, busy)
    )
endproc
```

C.4 The TRANS process in LOTOS

```
process Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id: Nat, v: Version) : noexit :=
  hide TX0 in
    (
    TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
    |[TX0]|
    TransRes [LDind, LDres, TX0] (id, v)
    )
endproc

(*------------------------------------------------------------------------*)

process TransReq [LDreq, LDcon, TDreq, TX0] (n, id: Nat) : noexit :=
  TDreq !id ?dest: Nat ?h: HEADER ?d: DATA [dest le n];
  (
  TX0;
  exit (dest, h, d)
  []
  exit (dest, h, d)
  ) >> accept dest: Nat, h: HEADER, d: DATA in
  (
  LDreq !id !dest !h !d;
    (
    [dest eq n] ->
      LDcon !id !broadsent;
      TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
    []
    [dest ne n] ->
      (
      choice a: ACK []
        LDcon !id !ackrec (a);
        TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
      )
    []
```

```
        LDcon !id !ackmiss;
        TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
      )
   )
endproc

(*------------------------------------------------------------------------*)

process TransRes [LDind, LDres, TX0] (id: Nat, v: Version) : noexit :=
   LDind !id ?l: LIN_DIND;
   (
   [is_broadrec (l)] ->
      (
      [v = ko] ->
         (* original (incorrect) specification *)
         LDres !id !a1 !no_op;
         TransRes [LDind, LDres, TX0] (id, v)
      []
      [v = ok] ->
         (* correct specification *)
         TransRes [LDind, LDres, TX0] (id, v)
      )
   []
   [not (is_broadrec (l))] ->
      (
      choice a: ACK []
         (
         (* concatenated response = lock transaction *)
         TX0;
         LDres !id !a !hold;
         TransRes [LDind, LDres, TX0] (id, v)
         []
         (* split response *)
         LDres !id !a !release;
         TransRes [LDind, LDres, TX0] (id, v)
         )
      )
   )
endproc
```

C.5 The APPLI process in LOTOS

```
process Application [TDreq] (n: Nat, id: Nat, s: Scenario) : noexit :=
   [s eq scenario_1] ->
      [id eq 0] ->
         (
         (* send a request for transaction with a *different* node *)
         choice dest: Nat, h: HEADER, d: DATA []
            [(dest le n) and (dest ne id)] ->
               TDreq !id !dest !h !d;
               stop
```

```
        )
   []
   [s eq scenario_2] ->
      (
      (* send a request for transaction with a *different* node *)
      choice dest: Nat, h: HEADER, d: DATA []
         [(dest le n) and (dest ne id)] ->
            TDreq !id !dest !h !d;
            stop
      )
   []
   [(s eq scenario_3_2) or (s eq scenario_3_3) or (s eq scenario_3_4)] ->
      [id eq 0] ->
         (
         (* 2, 3 or 4 requests in sequence *)
         choice h: HEADER, d: DATA []
            TDreq !id !n !h !d;
            TDreq !id !n !h !d;
            (
            [s eq scenario_3_2] ->
               stop
            []
            [s eq scenario_3_3] ->
               TDreq !id !n !h !d;
               stop
            []
            [s eq scenario_3_4] ->
               TDreq !id !n !h !d;
               TDreq !id !n !h !d;
               stop
            )
        )
endproc
```

C.6 The NODE process in LOTOS

```
process Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n: Nat, id: Nat, v: Version, s: Scenario) : noexit :=
   hide TDreq in
      (
      Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id)
      |[LDreq, LDcon, LDind, LDres]|
      Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id, v)
      |[TDreq]|
      Application [TDreq] (n, id, s)
      )
endproc
```

C.7 The MAIN process in LOTOS

```
specification P1394 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind,
                 arbresgap, losesignal] : noexit

library
   BOOLEAN, NATURAL, DATA
endlib

behaviour

   (
   Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (2, 0, ko,
        scenario_3_4)
   |||
   Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (2, 1, ko,
        scenario_3_4)
   )
   |[PDreq, PDind, PAreq, PAcon, PCind]|
   Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (2)

where

   library
      APPLI, TRANS, LINK, BUS, NODE
   endlib

endspec
```

For model-checking purposes, a complementary file restricts the set of natural numbers, e.g., to the finite range $\{0,...,2\}$ in the above example.

D Formal model in LNT

D.1 Types and functions in LNT

```
module DATA is

type CHECK is
   bottom, check
   with =, <>
end type

------------------------------------------------------------------

type DATA is
   d1 -- , d2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (d: DATA): CHECK is
   use d; -- this parameter was not used in the LOTOS specification
   return check
end function
```

```
-----------------------------------------------------------------

type HEADER is
   h1 -- , h2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (h: HEADER): CHECK is
   use h; -- this parameter was not used in the LOTOS specification
   return check
end function

-----------------------------------------------------------------

type ACK is
   a1 -- , a2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (a: ACK): CHECK is
   use a; -- this parameter was not used in the LOTOS specification
   return check
end function

-----------------------------------------------------------------

type BOC is
   release, hold, no_op
   with =
end type

type PHY_AREQ is
   fair, immediate
   with =
end type

type PHY_ACONF is
   won, lost
   with =
end type

-----------------------------------------------------------------

type SIGNAL is
   destsig (dest: Nat),
   headsig (head: HEADER, crc: CHECK),
   datasig (data: DATA, crc: CHECK),
   acksig (ack: ACK, crc: CHECK),
   dhead,
   Start,
   End,
   Prefix,
   subactgap,
   Dummy
   with =, <>, get, set
end type

function is_dest (s: SIGNAL) : Bool is
   case s in
      destsig (any nat) -> return true
   | any -> return false
   end case
end function

function is_header (s: SIGNAL) : Bool is
   case s in
      headsig (any HEADER, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_data (s: SIGNAL) : Bool is
   case s in
      datasig (any DATA, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_ack (s: SIGNAL) : Bool is
   case s in
      acksig (any ACK, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_physig (s: SIGNAL) : Bool is
   case s in
      Start | End | Prefix | subactgap -> return true
   | any -> return false
   end case
end function

function valid_hpart (s: SIGNAL) : Bool is
   return is_header (s) and then (s.crc = check)
end function

function valid_ack (s: SIGNAL) : Bool is
   return is_ack (s) and then (s.crc = check)
end function

function getdest (s: SIGNAL) : Nat is
   return s .[UNEXPECTED] dest
end function
```

```
function getdcrc (s: SIGNAL) : CHECK is
   assert is_data (s);
   return s .[UNEXPECTED] crc
end function

function getdata (s: SIGNAL) : DATA is
   return s .[UNEXPECTED] data
end function

function gethead (s: SIGNAL) : HEADER is
   return s .[UNEXPECTED] head
end function

function getack (s: SIGNAL) : ACK is
   return s .[UNEXPECTED] ack
end function

function corrupt (s: SIGNAL) : SIGNAL is
   case s in
      headsig (any HEADER, any CHECK) -> return s.{crc -> bottom}
   | datasig (any DATA, any CHECK) -> return s.{crc -> bottom}
   | acksig (any ACK, any CHECK) -> return s.{crc -> bottom}
   | any -> raise UNEXPECTED
   end case
end function

----------------------------------------------------------------------

type SIG_TUPLE is
   quadruple (dh, dest, header, data: SIGNAL),
   void
   with get
end type

function is_void (s: SIG_TUPLE) : Bool is
   case s in
      void -> return true
   | any -> return false
   end case
end function

----------------------------------------------------------------------

type LIN_DCONF is
   ackrec (a: ACK),
   ackmiss,
   broadsent
end type

----------------------------------------------------------------------
```

```
type LIN_DIND is
   good (h: HEADER, d: DATA),
   broadrec (h: HEADER, d: DATA),
   dcrc_err (h: HEADER)
end type

function is_broadrec (x: LIN_DIND) : Bool is
   case x in
      broadrec (any HEADER, any DATA) -> return true
   | any -> return false
   end case
end function

------------------------------------------------------------------------

type BoolTABLE is
   empty,
   btable (index: Nat, value: Bool, next: BoolTABLE)
   with =, get
end type

function init (n: Nat) : BoolTABLE is
   -- returns a table of size n initialized to false
   if n = 0 then
      return empty
   else
      return btable (n - 1, false, init (n - 1))
   end if
end function

function zero (t: BoolTABLE) : Bool is
   -- returns true iff no value in t is true
   if t = empty then
      return true
   elsif t.value then
      return false
   else
      return zero (t.next)
   end if
end function

function one (t: BoolTABLE) : Bool is
   -- returns true iff exactly one value in t is true
   if t = empty then
      return false
   elsif t.value then
      return zero (t.next)
   else
      return one (t.next)
   end if
end function
```

```
function more (t: BoolTABLE) : Bool is
   -- returns true iff more than one value in t is true
   return not (zero (t)) and not (one (t))
end function

function get (n: Nat, t: BoolTABLE) : Bool is
   -- returns the value associated with index n in t
   if t = empty then
      raise UNEXPECTED
   elsif t.index = n then
      return t.value
   else
      return get (n, t.next)
   end if
end function

function invert (n: Nat, t: BoolTABLE) : BoolTABLE is
   -- returns in which the value associated with index n is negated
   if t = empty then
      return empty
   elsif t.index = n then
      return btable (t.index, not (t.value), t.next)
   else
      return btable (t.index, t.value, invert (n, t.next))
   end if
end function

------------------------------------------------------------------------

type Version is
   ko, ok
end type

type Scenario is
   scenario_1, scenario_2, scenario_3_2, scenario_3_3, scenario_3_4
   with =
end type

function requests (s: Scenario): Nat is
   case s in
      scenario_3_2 -> return 2
   | scenario_3_3 -> return 3
   | scenario_3_4 -> return 4
   | any -> raise UNEXPECTED
   end case
end function

end module
```

D.2 Channels in LNT

```
module CHANNELS (DATA) is

channel Id is
   (n: Nat)
end channel

channel Sig is
   (id: Nat, flag: SIGNAL)
end channel

channel Areq is
   (id: Nat, flag: PHY_AREQ)
end channel

channel Acon is
   (id: Nat, flag: PHY_ACONF)
end channel

channel Ack is
   (id: Nat, a: ACK, b: BOC)
end channel

channel Dreq is
   (id: Nat, dest: Nat, h: HEADER, d: DATA)
end channel

channel Dind is
   (id: Nat, l: LIN_DIND)
end channel

channel Dcon is
   (id: Nat, l: LIN_DCONF)
end channel

end module
```

D.3 The LINK process in LNT

```
module LINK (DATA, CHANNELS) is

process Link [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
             PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat) is
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id, void)
end process

-------------------------------------------------------------------------------

process Link0 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
              PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE) is
   select
      if is_void (buffer) then
```

```
        var dest: Nat, h: HEADER, d: DATA, b: SIG_TUPLE in
           LDreq (id, ?dest, ?h, ?d);
           b := quadruple (dhead,
                           destsig (dest),
                           headsig (h, crc (h)),
                           datasig (d, crc (d)));
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, b)
        end var
     else
        PAreq (id, fair);
        -- here, the LOTOS process Link1 was expanded in-line
        -- (see footnote 8 in the research report [Sighireanu-Mateescu-97])
        select
           PAcon (id, won);
           -- here, Link2 represents the LOTOS process Link2req
           Link2 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, void, buffer)
        []
           PAcon (id, lost);
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        end select
     end if
  []
     var p: SIGNAL in
        PDind (id, ?p);
        if p = Start then
           Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        else
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        end if
     end var
  end select
end process

----------------------------------------------------------------------

process Link1 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
  PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, p: SIGNAL) is
  -- process Link1 factors code repeated thrice in process Link3 below
  LDcon (id, ackmiss);
  if p = subactgap then
     Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
           (n, id, buffer)
  else
     LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer, n)
  end if
end process
```

```
-------------------------------------------------------------------

process Link2 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
  PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, p: SIG_TUPLE) is
   -- process Link2 unifies the two LOTOS processes Link2req and Link2resp
   PCind (id);
   PDreq (id, Start);
   PCind (id);
   PDreq (id, p.dh);
   PCind (id);
   PDreq (id, p.dest);
   PCind (id);
   PDreq (id, p.header);
   PCind (id);
   PDreq (id, p.data);
   PCind (id);
   PDreq (id, End);
   if getdest (p.dest) = n then
      LDcon (id, broadsent);
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   else
      -- here, the LOTOS process Link3 was expanded in-line (called only once)
      var p, a, e: SIGNAL in
         loop L in
            PDind (id, ?p);
            if p <> Prefix then
               break L
            end if
         end loop;
         if p <> Start then
            Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, p)
         else
            -- here, the LOTOS process Link3RA was expanded (called only once)
            PDind (id, ?a);
            if is_physig (a) then
               Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                      (n, id, buffer, a)
            else
               -- here, the LOTOS process Link3RE was expanded (called only once)
               PDind (id, ?e);
               if valid_ack (a) and ((e = End) or (e = Prefix)) then
                  LDcon (id, ackrec (getack (a)));
                  LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                           PCind] (n, id, buffer, n)
               else
                  Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                         (n, id, buffer, e)
               end if
            end if
         end if
      end var
   end if
end process

----------------------------------------------------------------

process Link4 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
              PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE) is
   var s1, s2, s3, s4, s5: SIGNAL, dest: Nat in
      PDind (id, ?s1);
      if s1 = subactgap then
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      elsif is_physig (s1) then
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      else
         -- here, the LOTOS process Link4DH was expanded in-line (called only once)
         PDind (id, ?s2);
         if s2 = subactgap then
            Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id, buffer)
         elsif not (is_dest (s2)) or else
               ((getdest (s2) <> id) and (getdest (s2) <> n)) then
            LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                    (n, id, buffer, n)
         else
            dest := getdest (s2);
            if dest = id then
               PAreq (id, immediate)
            end if;
            -- here, the LOTOS process Link4RH was expanded (called only once)
            PDind (id, ?s3);
            if not (valid_hpart (s3)) then
               -- here, the LOTOS process Link4RD was expanded (called only once)
               LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                       (n, id, buffer, dest)
            else
               PDind (id, ?s4);
               if not (is_data (s4)) then
                  LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                          PCind] (n, id, buffer, dest)
               else
                  -- here, the LOTOS process Link4RE was expanded (called only once)
                  PDind (id, ?s5);
                  if (s5 <> End) and (s5 <> Prefix) then
                     LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                             PCind] (n, id, buffer, dest)
                  elsif dest <> id then
                     -- here, the LOTOS process Link4BRec was expanded (called only once)
                     if getdcrc (s4) = check then
                        LDind (id, broadrec (gethead (s3), getdata (s4)))
                     end if;
                     Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                            PCind] (n, id, buffer)
                  else
                     -- here, the LOTOS process Link4DRec was expanded (called only once)
                     if getdcrc (s4) = check then
                        LDind (id, good (gethead (s3), getdata (s4)))
                     else
                        LDind (id, dcrc_err (gethead (s3)))
                     end if;
                     PAcon (id, won);
                     -- here, the LOTOS process Link5 was expanded (called only once)
                     loop L in
                        select
                          PCind (id);
                          PDreq (id, Prefix)
                        []
                          break L
                        end select
                     end loop;
                     var a: ACK, b: BOC, p: SIGNAL in
                        LDres (id, ?a, ?b);
                        p := acksig (a, crc (a));
                        -- here, the LOTOS process Link6 was expanded (called only once)
                        PCind (id);
                        PDreq (id, Start);
                        PCind (id);
                        PDreq (id, p);
                        PCind (id);
                        if b = release then
                          PDreq (id, End);
                          Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                                 PCind] (n, id, buffer)
                        else
                          PDreq (id, Prefix);
                          -- here, the LOTOS process Link7 was expanded (called only once)
                          loop L in
                             select
                                PCind (id);
                                PDreq (id, Prefix)
                             []
                                break L
                             end select
                          end loop;
                          var dest: Nat, h: HEADER, d: DATA, t: SIG_TUPLE in
                             LDreq (id, ?dest, ?h, ?d);
                             t := quadruple (dhead,
                                             destsig (dest),
                                             headsig (h, crc (h)),
                                             datasig (d, crc (d)));
                             -- here, Link2 represents the LOTOS process Link2resp
```

```
                        Link2 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq,
                              PAcon, PCind] (n, id, buffer, t)
                      end var
                    end if
                  end var
                end if
              end if
            end if
          end if
        end if
    end var
end process

-----------------------------------------------------------------

process LinkWSA [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
    PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) is
    select
      var p: SIGNAL in
        PDind (id, ?p);
        if p = subactgap then
            Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id, buffer)
        else
            LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                    (n, id, buffer, dest)
        end if
      end var
    []
      only if dest = id then
          PAcon (id, won);
          PCind (id);
          PDreq (id, End);
          Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id, buffer)
      end if
    end select
end process

end module
```

D.4 The BUS process in LNT

```
module BUS (DATA, CHANNELS) is

process Bus [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
             arbresgap, losesignal: none] (n: Nat) is
    BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
end process

-----------------------------------------------------------------
```

```
process BusIdle [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                 arbresgap, losesignal: none] (n: Nat, t: BoolTABLE) is
   select
      var id: Nat in
         PAreq (?id, ?any PHY_AREQ) where id < n;
         -- here, the LOTOS process DecideIdle was expanded in-line
         -- (see footnote 7 in the research report [Sighireanu-Mateescu-97])
         if get (id, t) = false then
            PAcon (id, won);
            BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                    (n, invert (id, t), init (n), init (n), id)
         else
            PAcon (id, lost);
            BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
         end if
      end var
   []
      only if not (zero (t)) then
         arbresgap;
         BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, init (n))
      end if
   end select
end process

------------------------------------------------------------------------

process BusBusy [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                 arbresgap, losesignal: none] (n: Nat, t: BoolTABLE,
                 in var next: BoolTABLE, destfault: BoolTABLE, busy: Nat) is
   select
      var j: Nat in
         PAreq (?j, fair) where j < n;
         PAcon (j, lost);
         BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, next, destfault, busy)
      end var
   []
      var j: Nat in
         PAreq (?j, immediate) where not (get (j, next)) and (j < n);
         BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, invert (j, next), destfault, busy)
      end var
   []
      if busy < n then
         var p: SIGNAL in
            PCind (busy);
            PDreq (busy, ?p);
            Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                       (n, t, next, destfault, busy, p)
         end var
      elsif zero (next) then
         SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
      else
         -- here, the LOTOS process Resolve was expanded (called only once)
         var j: Nat, p: SIGNAL in
            for j := 0 while j < n by j := j + 1 loop
               if get (j, next) then
                  PAcon (j, won);
                  PCind (j)
               end if
            end loop;
            -- here, the LOTOS process Resolve2 was expanded (called only once)
            while more (next) loop
               PDreq (?j, End) where get (j, next) and (j < n);
               next := invert (j, next)
            end loop;
            PDreq (?j, ?p) where j < n;
            if p = End then
               SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                            (n, t)
            else
               Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                          (n, t, init (n), init (n), j, p)
            end if
         end var
      end if
   end select
end process

----------------------------------------------------------------------

process SubactionGap [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                      arbresgap, losesignal: none] (n: Nat, t: BoolTABLE) is
   var j: Nat in
      for j := 0 while j < n by j := j + 1 loop
         PDind (j, subactgap)
      end loop;
      BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
   end var
end process

----------------------------------------------------------------------

process Distribute [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                    arbresgap, losesignal: none] (n: Nat, t: BoolTABLE,
                    in var next, destfault: BoolTABLE, busy: Nat, p: SIGNAL) is
   var j, incr: Nat in
      for j := 0 while j < n by j := j + incr loop
         incr := 1;
         if j <> busy then
            select
               only if not (is_header (p) and get (j, destfault)) then
```

```
                PDind (j, p)
             end if
          []
             only if is_dest (p) then
                var dest: Nat in
                   dest := any Nat;
                   PDind (j, destsig (dest));
                   destfault := invert (j, destfault)
                end var
             end if
          []
             only if is_header (p) or is_data (p) or is_ack (p) then
                select
                   PDind (j, corrupt (p))
                []
                   losesignal
                end select
             end if
          []
             only if is_data (p) then
                PDind (j, p);
                PDind (j, Dummy)
             end if
          []
             PAreq (j, immediate) where not (get (j, next));
             incr := 0; -- instead of 1, here
             next := invert (j, next)
          end select
       end if
    end loop;
    if p = End then
       j := n
    else
       j := busy
    end if;
    BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
            (n, t, next, destfault, j)
 end var
end process

end module
```

D.5 The TRANS process in LNT

```
module TRANS (DATA, CHANNELS) is

process Trans [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, TDreq: Dreq]
              (n, id: Nat, v: Version) is
   hide TX0: none in
      par TX0 in
         TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
      ||
         TransRes [LDind, LDres, TX0] (id, v)
      end par
   end hide
end process

-----------------------------------------------------------------

process TransReq [LDreq: Dreq, LDcon: Dcon, TDreq: Dreq, TX0: none] (n, id: Nat) is
   var dest: Nat, h: HEADER, d: DATA, a: ACK in
      loop
         TDreq (id, ?dest, ?h, ?d) where dest <= n;
         select
            TX0
         []
            null
         end select;
         i; -- this "i" corresponds to the ">>" operator in the LOTOS specification
         LDreq (id, dest, h, d);
         select
            if dest = n then
               LDcon (id, broadsent)
            else
               a := any ACK;
               LDcon (id, ackrec (a))
            end if
         []
            LDcon (id, ackmiss)
         end select
      end loop
   end var
end process

-----------------------------------------------------------------

process TransRes [LDind: Dind, LDres: Ack, TX0: none] (id: Nat, v: Version) is
   var l: LIN_DIND, a: ACK in
      loop
         LDind (id, ?l);
         if is_broadrec (l) then
            case v in
               ko ->
                  -- original (incorrect) specification
                  LDres (id, a1, no_op)
            | ok ->
                  -- correct specification
                  null
            end case
         else
            a := any ACK;
            select
               -- concatenated response = lock transaction
               TXO;
               LDres (id, a, hold)
            []
               -- split response
               LDres (id, a, release)
            end select
         end if
      end loop
   end var
end process

end module
```

D.6 The APPLI process in LNT

```
module APPLI (DATA, CHANNELS) is

process Application [TDreq: Dreq] (n: Nat, id: Nat, s: Scenario) is
   var dest: Nat, h: HEADER, d: DATA, r: Nat in
      case s in
         scenario_1 ->
            only if id == 0 then
               -- send a request for transaction with a *different* node
               dest := any Nat where (dest <= n) and (dest <> id);
               h := any HEADER;
               d := any DATA;
               TDreq (id, dest, h, d);
               stop
            end if
      | scenario_2 ->
            -- send a request for transaction with a *different* node
            dest := any Nat where (dest <= n) and (dest <> id);
            h := any HEADER;
            d := any DATA;
            TDreq (id, dest, h, d);
            stop
      | scenario_3_2 | scenario_3_3 | scenario_3_4 ->
            only if id == 0 then
               h := any HEADER;
               d := any DATA;
               for r := requests (s) while r > 0 by r := r - 1 loop
                  TDreq (id, n, h, d)
               end loop;
               stop
            end if
      end case
   end var
end process

end module
```

D.7 The NODE process in LNT

```
module NODE (DATA, CHANNELS, APPLI, TRANS, LINK) is

process Node [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
          PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, v: Version, s: Scenario) is
   hide TDreq: Dreq in
      par
         TDreq ->
            Application [TDreq] (n, id, s)
      ||
         TDreq, LDreq, LDcon, LDind, LDres ->
            Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id, v)
      ||
         LDreq, LDcon, LDind, LDres ->
            Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id)
      end par
   end hide
end process

end module
```

D.8 The MAIN process in LNT

```
module scen3_orig_2_4 (APPLI, TRANS, LINK, NODE, BUS) is

!nat_sup 2

process MAIN [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind:
   Sig, PAreq: Areq, PAcon: Acon, PCind: Id, arbresgap, losesignal: none] is
   par PDreq, PDind, PAreq, PAcon, PCind in
      par
         Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (2, 0, ko, scenario_3_4)
      ||
         Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (2, 1, ko, scenario_3_4)
      end par
   ||
      Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (2)
   end par
end process

end module
```

Formally Modelling the Rijkswaterstaat Tunnel Control Systems in a Constrained Industrial Environment

Kevin H.J. Jilissen
Rijkswaterstaat
Utrecht, the Netherlands
Eindhoven University of Technology
Eindhoven, the Netherlands
kevin.jilissen@rws.nl

Peter Dieleman
Rijkswaterstaat
Utrecht, the Netherlands
peter.dieleman@rws.nl

Jan Friso Groote
Eindhoven University of Technology
Eindhoven, the Netherlands
j.f.groote@tue.nl

Rijkswaterstaat, the National Dutch body responsible for infrastructure, recognised the importance of formal modelling and set up a program to model the control of road tunnels. This is done to improve the standardisation of tunnel control and make communication with suppliers smoother. A subset of SysML is used to formulate the models, which are substantial. In an earlier paper we have shown that these models can be used to prove behavioural properties by manually translating the models to mCRL2. In this paper we report on an automatic translation to mCRL2. As the results of the translation became unwieldy, we also investigated modelling tunnel control in the specification language Dezyne which has built-in verification capabilities and compared the results.

1 Introduction

Over the last few years, Rijkswaterstaat (RWS, the Dutch body responsible for road and water infrastructure in the Netherlands) has created SysML models of all system parts of the tunnel control systems describing the functionality these systems should perform based on a functional decomposition. They use SysML as within RWS there is a preference to use modelling tools with commercial support and industry acceptance. These models have been created as generic blueprints for the construction of several road tunnels. The behaviour of the systems is modelled using a functional decomposition in nested Activity Diagrams. There is no formal semantics to which the models adhere and essential parts are denoted in ‘structured natural language’.

Formal methods could be applied to reduce the chance of system failures and the chance that there are design flaws in the systems. RWS has shown interest in this approach, and in [11] it is already shown that a structured but largely manual translation of these SysML models to the formal mCRL2 [4] specification language is possible and very beneficial. The goal of this translation is to improve the quality of the models by formal verification of both safety and liveness properties of both the overall system as well as its individual components.

However, a manual translation to formal models is not deemed as the desirable solution by RWS as the set of models is substantial and they are regularly revised and changed. Manual translation is time consuming and relatively error prone. This may lead to the situation that changes in the SysML models quickly become out of sync with the verifiable mCRL2 models leading to a reduced benefit of verification.

For this reason, we investigate if the existing SysML specifications can automatically be translated to mCRL2. The first major obstacle is that SysML has no formal semantics — but see for instance [15] on how this can be remedied — and in particular the ‘structured natural language’ in the SysML models cannot be formally interpreted and translated. Therefore, we first systematically translate this to SysML activity diagrams. Subsequently, based on the XML interchange format, we constructed a translator in Spoofax [13], which is a language translation workbench with built-in support for variable declarations with local scope.

The communication scheme employed in the SysML models of tunnels is that all components simultaneously read their input and deliver their output. In [11] it was already observed that this leads to mCRL2 models with relatively few states but with a huge number of outgoing transitions in each state, sometimes more than 10^8. The automatic translation showed that this problem was exacerbated with the larger models making the verification of properties cumbersome.

Therefore, we also investigated modelling tunnels in the Dezyne specification language [3], which is commercially available and has very effective built-in verification. Following guidelines in [7], we provide both push and pull models for certain tunnel control components, and compared the behaviour with those generated from the SysML description, which was still possible despite the very different styles of modelling.

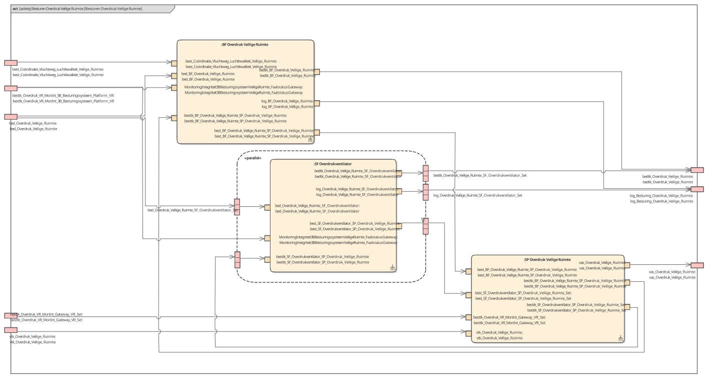
This paper presents the journey to develop support to improve the quality of the semi-formally specified tunnel control systems, which is a project largely running within Rijkswaterstaat. The results are in no way as clear cut as we hoped, and this work led to some conclusions, quite different from what we originally anticipated. We believe that these conclusions can be generalised to other behavioural specifications in graphical languages without a proper formal semantics. Besides this project, the enhancement of safety and reliability using formal methods is also being investigated by employing Synthesis Based Engineering (SBE) [16]. In this approach formal safety properties are used to automatically generate the control models, which is quite different from our approach to verify liveness and safety properties on explicitly specified control systems.

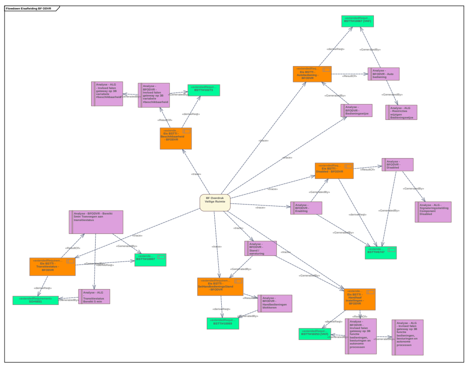
2 Existing SysML model structure

The complete tunnel installation has been generically modelled in Enterprise Architect [22] using SysML version 1.5 [18] and documented respecting J-STD-16 [9]. The genericity is introduced by a parameterised description that can be instantiated for any specific tunnel configuration. Examples of such parameters include the number of traffic tubes, the number of lanes, and the number and configuration of ventilators.

The permitted model elements and relations are specified using SysML in the meta-model of the model. This meta-model prescribes that all functionalities defined in the Landelijke Tunnel Standaard (translated: Dutch National Tunnel Standard) [20] must be modelled as behaviour using Activity Diagrams (ADs). In these ADs, the repeated token flow simulates the computation cycles of the Programmable Logic Controllers (PLCs) deployed in tunnel installations.

The behaviour of the tunnel control system is modelled as a functional decomposition of Activity Diagrams. The root AD contains one component encapsulating all behaviour of the system, with the environmental readings as input and the actuator control as output. Every behavioural component in the system contains subcomponents until every technical or system task is described by exactly one leaf subcomponent. An example of such leaf AD containing elementary tasks, or activities, regarding controlling the overpressure of safe spaces, in Dutch ‘overdruk veilige ruimte’, is depicted in Figure 1a. The elementary tasks are subject to various further descriptions in SysML, as depicted in Figure 1b. Important sub-descriptions are how actions, consisting of value assignments, must be carried out subject to certain conditions when the system receives messages, see Figure 1c. These descriptions consist of a curious mixture of Dutch and programming notation, to which we refer in this document as ‘structured natural language', which has no formal syntax and semantics. Note that the diagrams are often small and cluttered, and require the zoom feature of pdf to be readable.


(a) The leaf AD of the overpressure sub-system.


(b) Additional descriptions for Figure 1a.

```
EnableOverpressure()
Condition: *
Actions: #enabled := yes
```

(c) An elementary task description translated from Dutch.

```
DisableOverpressure()
Condition: *
Actions: #enabled := no
```

(d) Another elementary task description.

Figure 1: An example of a leaf AD taken from the SysML description of tunnel control systems.

Basic functionalities in the model are grouped together in so-called vertical slices. In total, the engineers identified 35 vertical slices in the systems. These slices together contain 52 standalone sub-systems responsible for some facility. The sub-systems are decomposed in a Base Functionality (BF) responsible for the overall management of that sub-system. They can contain zero or more Sub-Functionalities (SF) responsible for a single, possibly instantiated, entity, such as a single ventilation group or a single ventilator. Finally, the sub-systems can contain Drivers (SP, abbreviation of 'Stuurprogramma' in Dutch) which translate the generic control commands and status information to and from the vendor-specific implementations.

For this paper, the vertical slice and equally named sub-system for the overpressure of the safe space is used as the system we apply our techniques on. It consists of a Base Functionality and one Sub-Functionality which can be instantiated for (possibly redundant) overpressure ventilators. This sub-system was chosen as it is one of the investigated systems in [11]. Additional details on translations and models are provided in the appendices. In Appendix A a non traceable link is given to a zip file containing all artefacts belonging to this paper.

3 Model adaptions for formal analysis

In this section we sketch how the SysML models are translated to mCRL2. We only allow a restricted use of SysML activity diagrams, and this is enforced by the type checking in the translation. The transformation is implemented using the Spoofax Language Workbench [13]. Syntax definitions for parsing XMI are described in Spoofax using the SDF3 meta-language [21]. Static analysis rules on the parsed Abstract Syntax Tree (AST) are formulated in Statix [1], the meta-language for the specification of static semantics included in Spoofax 3. The analysed input AST is transformed using Stratego [12] to an AST of the resulting mCRL2 specification and exported. Figure 2 gives an overview of this workflow.

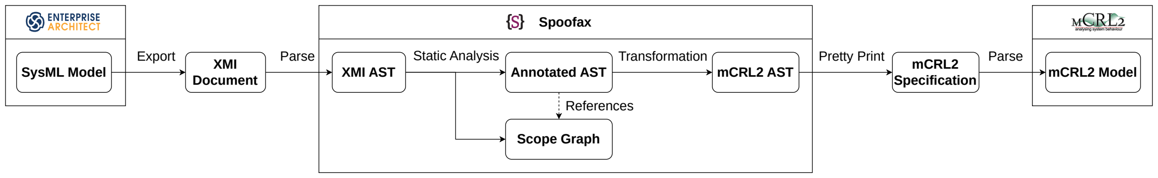


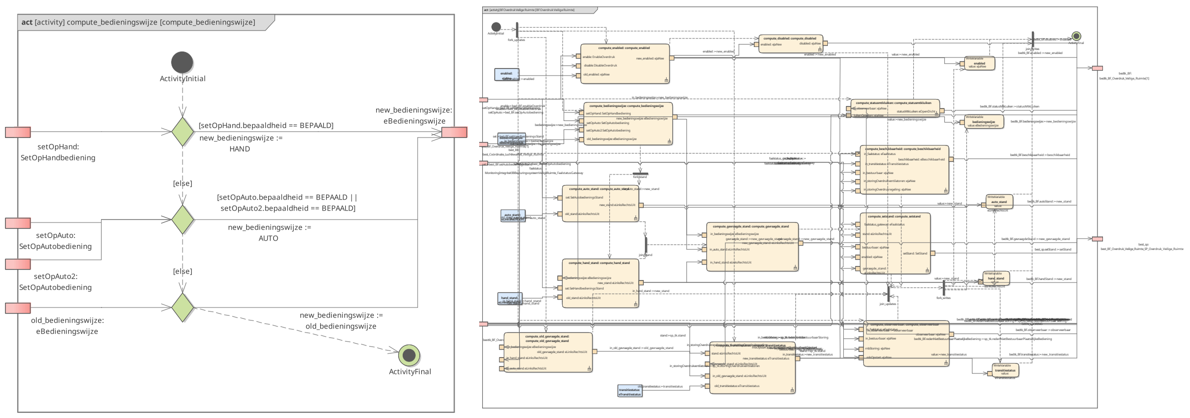
Figure 2: An overview of the workflow.

3.1 Formalising 'structured natural language'

The first step towards the translation is to replace the structured natural language descriptions by a notation that can be formally interpreted and translated. We formalise the structured natural language descriptions using activity diagrams annotated with simple conditions and assignments in SysML, as this deviates the least from the framework Rijkswaterstaat is using.

To create this description, two additional decomposition layers must be added. These manually created layers embed the structured natural language with a fixed semantics within the SysML model, such that these can be constructed and maintained by the SysML engineers. A new bottom leaf layer consists of ADs, in which each AD formally describes how several task descriptions such as in Figures 1c and 1d combined compute the values of the assigned variables. An example of an AD in the leaf layer is given in Figure 3a.

In between this newly created leaf layer and the ADs of the existing SysML specification, such as Figure 1a, a glue layer is added. For each activity in the existing AD, an AD in the glue layer is added which formally describes how the variables in the textual descriptions are bound to input, output, and state variables. The description in Figure 1a requires three ADs in the glue layer of which one is depicted in Figure 3b.


act [activity] compute_bedieningswijze [compute_bedieningswijze]
ActivityInitial
[setOpHand.bepaaldheid == BEPAALD]
new_bedieningswijze := HAND
new_bedieningswijze: eBedieningswijze
setOpHand: SetOpHandbediening
[else]
[setOpAuto.bepaaldheid == BEPAALD || setOpAuto2.bepaaldheid == BEPAALD]
new_bedieningswijze := AUTO
setOpAuto: SetOpAutobediening
setOpAuto2: SetOpAutobediening
[else]
old_bedieningswijze: eBedieningswijze
new_bedieningswijze := old_bedieningswijze
ActivityFinal

(a) Example AD in the leaf layer.

(b) Example AD in the glue layer.

Figure 3: Example Activity Diagrams in the newly introduced layers.

3.2 Assignment-based language in SysML

SysML allows modellers to define their own language to be used as names and guards on flows in diagrams. In the ADs in Figure 3a, a simple conditional assignment-based language is used which is just expressive enough to capture the natural language encountered in the specifications. The design for this language is guided by minimalism, which is trivial to transform to mCRL2.

The goal of the language is to assign values to all outgoing activity parameters and to the pins of all actions that write a new value to a state variable. For this purpose, flow names become assignments of the shape $e_1 := e_2$ and flow guards are boolean expressions with connectives $\neg$, $\vee$ and $\wedge$, and basic propositions of the shape $e_1 = e_2$ and $e_1 \neq e_2$. Within a sensible name resolution scope of the flow, e_1 and e_2 must refer to named elements and values of equivalent types in the SysML model.

Defining a sensible name resolution scheme and determining what equally typed values are is the most complex task of the transformation from XMI to mCRL2. For this we use the tool Statix, in which type checking is reduced to a constraint solving problem. If a solution is derived for the constraints on the root in the AST, the provided AST is well-typed. A scope graph is constructed in Statix while solving this constraint problem [1, 17].

The scope graph framework consists of a graph, which represents the naming structure of the AST, and a resolution calculus, which describes how to resolve references to declarations within a scope graph. A scope graph $\mathcal{G}$ connects *scopes* $s \in \mathcal{S}$, containing *data terms* $d \in \mathcal{D}$ bound by *relations* $r \in \mathcal{R}$, using directed *edges* labelled with *labels* $l \in \mathcal{L}$. A scoped datum $s \xrightarrow{r} d$ associates a data term d with scope s under relation r. Variable declarations in scope s are represented by $s \xrightarrow{:} (n, T)$, shortened by $n\colon T$ for name n with type T. Resolution between scopes is governed by path queries. The query can be read as a regular expression with the edge labels as alphabet. For example P^* matches paths with zero or more edges labelled P, $P?$ matches paths with zero or one P label, and ε the empty path. Edges $s_1 \xrightarrow{P} s_2$ are used to denote that s_2 is the parent scope of s_1. Due to the structure of XMI, $\mathcal{G}$ is structured as a tree with respect to label P such that from every scope s, root node s_r is reachable using path P^*. Notation ∇s is used to indicate a fresh scope not part of $\mathcal{G}$.

3.3 Abstract notation for XMI

The SysML specification can be exported to XML Metadata Interchange (XMI) version 2.1 [19]. This textual and computer-interpretable format is used as input to our transformation framework and parsed by Spoofax to an AST.

```
<node xmi:type="uml:InitialNode"
      xmi:id="EAID_1"
      name="ActivityInitial"
      visibility="public">
   <outgoing xmi:idref="EAID_2"/>
</node>
```

(a) The XMI document snipped.

```
UmlActivityElement.UmlActivityInitial =
  [<node xmi:type="uml:InitialNode"
         xmi:id="[XML-STRING]"
         name="[LANG-ID]"
         visibility="[UmlVisibility]">
     [UmlOutgoing]
  </node>]
```

(b) The SDF3 production rule used for parsing.

```
UmlActivityInitial("EAID_1", "ActivityInitial", UmlPublic(), UmlOutgoing("EAID_2"))
```

(c) The parsed AST subtree with syntactic constructors.

Figure 4: AST construction from XMI document in Spoofax.

To represent the parsed document in the AST, Spoofax generates syntactical constructor type definitions based on the SDF3 syntax specification. These syntax specifications can be specified using string templates and non-terminal symbols. Consider the representation of an Activity Initial, a start point of the control flow of an AD, in an XMI document. Assuming that the referenced symbols between [and] brackets are defined, a SDF3 production rule for symbol UmlActivityElement and constructor name UmlActivityInitial is given in Figure 4.

In the remainder of this paper, abstract versions of these syntactic constructors of the parsed XMI format are used to describe the transformation and semantical elements textually. Some SysML constructs are mapped to the same XMI elements. All relevant SysML elements with their syntactic constructor notation are listed in Table 1 where each element gets a unique ID i. In flows the directed edge relations in the ADs are drawn dashed for the control flows and solid for object flows. These are used to determine the flow type.

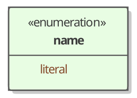
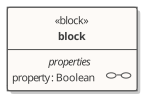
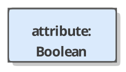
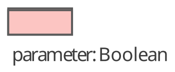
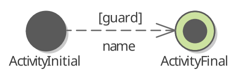
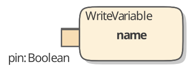
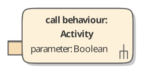
Abstract constructor	**Constructor parameter description**	**Visualisation**
EnumerationLiteral(i,n)	ID i, name n	
Enumeration(i,n,L)	ID i, name n, set of enumeration literals L	
Property(i,n,t)	ID i, name n, ID reference of property type t	
Block(i,n,P)	ID i, name n, set of block properties P	
Attribute(i,n,t,d)	ID i, name n, ID reference of attribute type t, default value d of referenced type	
ActivityParameter(i,n,t)	ID i, name n, ID reference of parameter type t	
ActivityInitial(i)	ID i	
ActivityFinal(i)	ID i	
Flow(i,f,n,g,s,t)	ID i, flow type $f \in \{\text{control},\text{object}\}$, name n or empty name ε, guard g or empty guard δ, source ID reference s, target ID reference t	
Pin(i,n,t)	ID i, name n, ID reference of parameter type t	
WriteVariable(i,n,p)	ID i, variable name n, input pin p	
CallBehaviour(i,n,b,P)	ID i, name n, ID reference of behaviour b, set of parameters P	
DecisionNode(i,n)	ID i, name n	

Table 1: Table with abstract syntactic constructors for SysML XMI elements.

3.4 Well-typedness using constraint solving in Statix

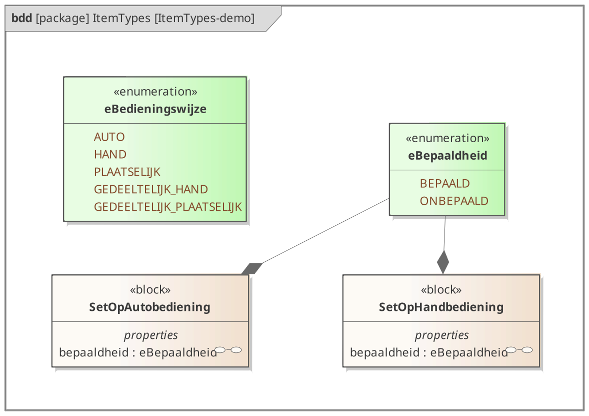
The SysML models are structured in a tree of packages. The root of the XMI document contains the root package. The packages help to guide humans in navigating the model. The document is well-typed if and only if the root is well-typed, and the root, or for that matter any node, is well-typed iff all its children are well-typed, although for each node extra typing constraints may be required.

These extra typing constraints are formulated in Statix using inference rules. As an example we give the rule for an enumeration that typically belongs to an enumeration declaration as in Figure 5a.

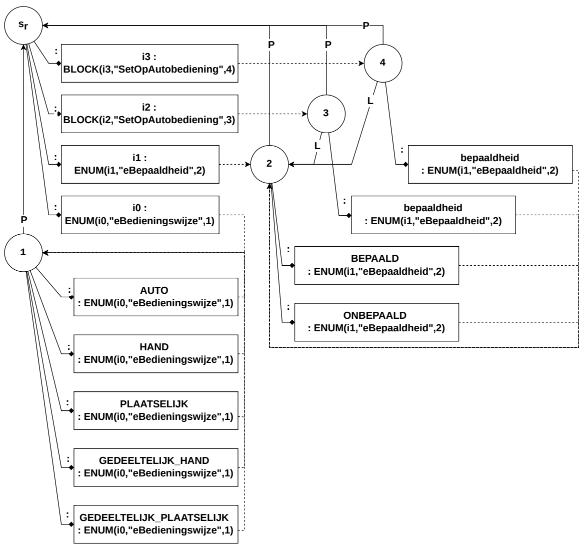
$$\frac{\nabla s_e \quad s_e \xrightarrow{P} s_r \quad T \equiv \text{ENUM}(i,n,s_e) \quad s_r \dot{\rightarrow} (i,T) \quad \forall_{\text{EnumerationLiteral}(i_l,n_l)\in L}\, s_e \dot{\rightarrow} (n_l,T)}{\text{Enumeration}(i,n,L)_{\textbf{OK}}}$$

In order to understand this rule it is important to know that these rules use scope graphs, see for instance Figure 5b. A scope graph is a directed graph of scopes with labelled links in which objects are declared. By searching the scope graph the nearest scope can be found in which an object is declared to determine its type and other properties. In the rule above ∇s_e says that a new scope s_e is added to the scope graph, and $s_e \xrightarrow{P} s_r$ says that this new scope is linked the existing scope s_r.

The rule above now expresses that an enumeration is well-typed, Enumeration$(i,n,L)_{\textbf{OK}}$, if the enumeration with unique identifier i is added to scope s_r with type $T = \text{ENUM}(i,n,s_e)$. Furthermore, all elements n_l of this enumeration, which occur in list L, are added with the same type T to the new scope s_e.



(a) The SysML BDD snippet.


s_r
i3 : BLOCK(i3,"SetOpAutobediening",4)
i2 : BLOCK(i2,"SetOpAutobediening",3)
i1 : ENUM(i1,"eBepaaldheid",2)
i0 : ENUM(i0,"eBedieningswijze",1)
1
2
3
4
P
L
bepaaldheid : ENUM(i1,"eBepaaldheid",2)
bepaaldheid : ENUM(i1,"eBepaaldheid",2)
BEPAALD : ENUM(i1,"eBepaaldheid",2)
ONBEPAALD : ENUM(i1,"eBepaaldheid",2)
AUTO : ENUM(i0,"eBedieningswijze",1)
HAND : ENUM(i0,"eBedieningswijze",1)
PLAATSELIJK : ENUM(i0,"eBedieningswijze",1)
GEDEELTELIJK_HAND : ENUM(i0,"eBedieningswijze",1)
GEDEELTELIJK_PLAATSELIJK : ENUM(i0,"eBedieningswijze",1)

(b) The corresponding scope graph of Figure 5a.

Figure 5: A snippet of enum and block definitions in a Block Definition Diagram (BDD) of the model.

More examples of well-typedness rules are given in Appendix C. Besides the links labelled with P for name resolution, additional edges in the scope graph occur labelled with T and L to connect the scope of the block to the scopes of the referenced semantic type BLOCK and ENUM respectively. Figure 5 shows an example snippet of enum and block definitions, together with the constructed scope graph. Visually, referenced scopes in semantic type constructors are connected with a dashed line. For the translation to mCRL2, every declaration of semantical type ENUM or BLOCK in s_r is directly translated to a sort specification.

In the next sections, AD elements that represent typed values are declared as variables of the referenced type. As semantic types are declared using their unique XMI ID in the root scope, they can always be resolved by following a P^* path from the scope to s_r. Flows that connect elements in the diagram also connect the nodes in the scope graph with labelled edges. References to block properties or enumeration literals can be accessed using a dot notation. Judgement $s \vdash e\colon T$ denotes that in the context of scope s, expression e has type T. Judgement $s \vdash p \overset{r}{\mapsto} d$ states that data term d is visible through path query p from scope s under relation r. Trivially, $s \vdash n\colon T$ if $s \vdash \varepsilon \overset{\cdot}{\mapsto} n\colon T$.

We show the typing rules to refer to a property of a block $n_1.n_2$, to equality $\square \in \{\equiv, \not\equiv\}$, to negation $\neg$, and to binary boolean operators $\Delta \in \{\wedge, \vee\}$.

$$\frac{s \vdash n_1\colon T_1 \quad \text{scope}(T_1) \vdash \varepsilon \overset{\cdot}{\mapsto} n_2\colon T_2}{s \vdash n_1.n_2\colon T_2} \qquad \frac{s \vdash n_1\colon T \quad s \vdash n_2\colon T}{s \vdash n_1 \square n_2\colon \mathbb{B}} \qquad \frac{s \vdash n\colon \mathbb{B}}{s \vdash \neg n\colon \mathbb{B}} \qquad \frac{s \vdash n_1\colon \mathbb{B} \quad s \vdash n_2\colon \mathbb{B}}{s \vdash n_1 \Delta n_2\colon \mathbb{B}}$$

3.5 Leaf Decomposition Layer

The elements in Activity Diagrams representing typed values are declared in the scope graph while establishing the well-typedness of the AST elements. To demonstrate how variables are declared and their linked types resolved, consider the activity parameters of the diagram in Figure 3a. Let ∇s_a with $s_a \xrightarrow{P} s_r$ and $s_r \overset{\cdot}{-\!\!\bullet}\, i\colon \text{ACT}(i, s_a)$ be the scope and semantic type declaration of the AD with id i. For every such parameter ActivityParameter(i,n,t), the well-typedness conclusion is established, making sure the referenced type exists, and a declaration of the semantic type PARAM in scope s_a is created.

$$\frac{\nabla s_p \quad s_p \xrightarrow{P} s_a \quad s_p \xrightarrow{T} \mathit{scope}(T) \quad s_p \vdash P^* \overset{\cdot}{\mapsto} t\colon T \quad s_p \overset{\cdot}{\rightarrow} n\colon T \quad s_a \overset{\cdot}{-\!\!\bullet}\, (i, \text{PARAM}(i,n,T,s_p))}{\text{ActivityParameter}(i,n,t)_{\mathbf{OK}}}$$

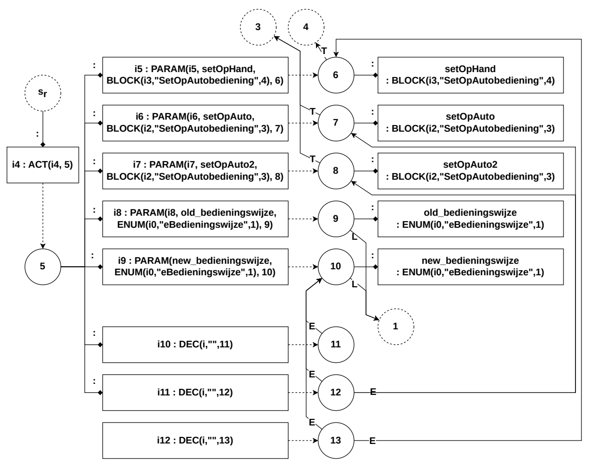
The well-typedness establishment of an object flow from s to t connects the flow scope ∇s_f with $s_f \xrightarrow{E} \mathit{scope}(s)$. Other derivations make sure to finish the path from the target to scope s_f by inserting $\mathit{scope}(t) \xrightarrow{E} s_f$. It is sufficient to only allow if-then-else decision nodes, as this trivial split in decisions makes the diagrams surprisingly easy to read for modellers whilst being sufficiently expressive. Every decision node has either one outgoing object flow with empty guard δ, or one flow with as guard special value `else` flowing to the next decision node and one object flow with a guarded value assignment $e_1 := e_2$ as name n to an output activity parameter. Value e_2 must be visible by a path following the object flows in reverse direction with E^*. To resolve enumeration literals as constants for enumerations in scope, T^+L is added to the allowed paths.

$$\frac{\mathit{scope}(t) \vdash \varepsilon \overset{\cdot}{\mapsto} e_1\colon T \quad \mathit{scope}(s) \vdash E^*(T^+L)? \overset{\cdot}{\mapsto} e_2\colon T}{(e_1 := e_2)_{\mathbf{OK}}}$$

An important benefit of these very restrictive semantics is that the diagrams describe a completely deterministic computation procedure. As a result, the diagrams can be completely expressed using data expressions in mCRL2 without the need to encode the semantical rules of the diagram in the resulting process specification. To ease the binding of input and output variables in mCRL2, a map declaration is defined for every diagram as demonstrated in Figure 6b.

3.6 Glue Decomposition Layer

The ADs in the glue layer use the same definitions for the flows in the diagrams. Decision nodes are not supported in this layer. The behaviour calls are calling the ADs defined in the previous section. In the


3
4
T
:
i5 : PARAM(i5, setOpHand, BLOCK(i3,"SetOpAutobediening",4), 6)
6
setOpHand : BLOCK(i3,"SetOpAutobediening",4)
s_r
:
i6 : PARAM(i6, setOpAuto, BLOCK(i2,"SetOpAutobediening",3), 7)
7
setOpAuto : BLOCK(i2,"SetOpAutobediening",3)
i4 : ACT(i4, 5)
i7 : PARAM(i7, setOpAuto2, BLOCK(i2,"SetOpAutobediening",3), 8)
8
setOpAuto2 : BLOCK(i2,"SetOpAutobediening",3)
i8 : PARAM(i8, old_bedieningswijze, ENUM(i0,"eBedieningswijze",1), 9)
9
old_bedieningswijze : ENUM(i0,"eBedieningswijze",1)
5
i9 : PARAM(new_bedieningswijze, ENUM(i0,"eBedieningswijze",1), 10)
10
new_bedieningswijze : ENUM(i0,"eBedieningswijze",1)
L
1
i10 : DEC(i,"",11)
11
i11 : DEC(i,"",12)
12
i12 : DEC(i,"",13)
13
E

(a) The extension of Figure 5b excluding $\xrightarrow{P}$ edges.

```
map
  compute_bedieningswijze2 :
      eBedieningswijze # SetOpAutobediening
      # SetOpAutobediening #
      SetOpHandbediening ->
      eBedieningswijze;
var
  old_bedieningswijze2 : eBedieningswijze,
      setOpAuto : SetOpAutobediening,
      setOpAuto2 : SetOpAutobediening,
      setOpHand : SetOpHandbediening;
eqn
  compute_bedieningswijze2(old_
      bedieningswijze2, setOpAuto,
      setOpAuto2, setOpHand) = if ((
      bepaaldheid5(setOpHand) == BEPAALD),
      HAND, if (((bepaaldheid4(setOpAuto)
      == BEPAALD) || (bepaaldheid4(
      setOpAuto2) == BEPAALD)), AUTO, old_
      bedieningswijze2));
```

(b) The generated mCRL2 specification.

Figure 6: Generated scope graph and mCRL2 specification of Figure 3a.

translation of the AD to an mCRL2 process specification, all behaviour calls are translated as $\sum_{o:\,T_o} o =$ ActivityMapping$(in_1,\ldots,in_n)$ where o is a fresh variable name of output type T_o, which is set equal to the result of calling the mapping with the bound input parameters $in_1,\ldots,in_n$. This process specification follows the pattern that can be optimised by sum elimination in the mCRL2 toolset. The attributes are translated to process parameters in the process equation representing the AD. All WriteVariable elements are translated to process variable updates in the recursive process specification. Again, the very restrictive subset of diagram elements permitted in this layer makes it trivial to express the semantics of every diagram by a process specification in mCRL2 by defining a specification with state variables for every attribute, using the previously defined maps and recursing in itself with updated state variables using the WriteVariable calls. Figure 3b depicts such a glue AD and the generated mCRL2 is moved to Appendix B due to its size.

3.7 Strengths and weaknesses

The translation sketched above has the following strengths. It is compositional and can be applied to translate the complete SysML tunnel specification by putting all components in parallel and combining the synchronous input and output into multi-actions in mCRL2 as explained in [11].

All model elements added to the approach are understandable with the SysML knowledge that was already required to understand the previous model. In this way, all engineers at RWS and its contractors are able to read and specify these kinds of diagrams. Another strong reason to choose this approach is that with our extension to replace ‘structured natural language’ the complete tunnel specification can be formulated as a large coherent and reasonably precise SysML specification.

Unfortunately, the SysML tooling does currently not enforce that the SysML specification is well-typed and internally consistent. But this is remedied as the static analysis using Statix of the exported XMI type checks the model and gives feedback on mistakes. The value of the defined semantics for the AD layers is shown in an analysis of the generated mCRL2 specification. Even though the presented work mainly focusses on the formal modelling and specification, all verification results of the previously analysed properties of the manual translation [10] could be replicated on the generated specification, as shown in Table 2. This is not an exhaustive list of all properties that must be verified on the system, but rather a selective subset which demonstrates that several types of properties are feasible to verify. The properties are checked for the complete sub-system instantiated with one instance of the BF, SF, and SP. The replication also includes verifying the modification of the model introduced in [10] to fix the model such that a bumpless transfer is guaranteed. The μ-calculus formulae with adapted naming to fit the generated model are included in the model repository of this paper.

Requirement	**Description**	**H**	**S**
Deadlock freeness	The control system can never reach a state in which no progress is possible.	✓	✓
Configured manual stand	The control system only sends the last configured manual-control values to the controlled installation when the control mechanism is manual-control.	✓	✓
No spontaneous change	The control system can never change the value sent to the controlled installation without explicitly receiving commands to do so.	✓	✓
Bumpless transfer	The control mechanism must be altered 'bumpless': there should not be an immediate change in behaviour of the control system without explicit requests to do so.	×	×

Table 2: Verification results of the hand-written (H) and SysML (S) mCRL2 translation.

The main weakness of this SysML based style is that every detail has to be specified graphically. To create the diagrams for this model, a lot of manual labour is needed to draw all the relations with components and to specify all flow names using the assignment language, as can already been seen by the glue layer of the BF in Figure 3b. Assigning and aligning all flows and flow name labels is sheer drudgery, and even with automated routing of flows the readability suffers greatly. With the huge number of diagrams, it is almost impossible for an engineer to keep an overview of the whole system. It is therefore questionable whether graphical formalisms such as SysML are fundamentally the most efficient way to model system with the complexity of tunnel control systems.

Another important weakness of this approach is the style in ADs to let all inputs and outputs occur simultaneously. Although, this matches neatly on multi-actions in mCRL2, it fundamentally leads to an exponential growth in the number of outgoing transitions in each state of the mCRL2 model. This puts fundamental constraints on the analysability of the whole tunnel control model.

4 Alternative models using Dezyne

As the SysML models are not ideal, it has been investigated whether the Dezyne specification language [3] offers a viable alternative. Dezyne offers a syntax that looks similar to widely used programming languages in industry and has visualisations closely related to SysML, whilst fitting in the current modelling workflow. Therefore, the adoption of the language within RWS is deemed feasible.

In Dezyne, components are defined which communicate over ports of formally specified interfaces with observable behaviour. Components can be composed together in systems by connecting ports with ports of other components or letting them communicate with ports of the environment. Dezyne requires that a number of properties are verified before generating code, such as freedom of deadlock, absence of illegal behaviour and interface compliancy meaning that a component exactly provides the interface that had to be separately specified. Under the hood, Dezyne uses mCRL2 as its verification engine and can export specifications and transition systems in formats usable in the mCRL2 tooling [2].

As a proof-of-concept, the SysML specification of the overpressure controller is manually translated to Dezyne. In the SysML model, all inputs arrive together in one big event. In the translation, such big events are split-up by letting each part of the input employ a single command or status event as Dezyne is not designed nor intended to handle such massive input events.

In the SysML model the massive input contains fields to indicate whether a particular input is to be considered to be present or absent in the input. In Dezyne this information is not necessary as input events with data that is absent simply do not need to take place. Omitting the indicator that data is present or absent has a hugely reducing effect on the complexity of the state space.

As illustration, consider the interface combining the two commands blocks in Figure 5a. This interface is translated to Dezyne as an interface containing two in-events SetOpAutobediening and SetOpHandbediening without parameters.

An issue with Dezyne version 2.18, is that values in the input cannot be used within the model and can only be passed on. This means that values in the input that have an influence on the behaviour are encoded in the event name. Unfortunately, this applies to all enumeration values in the investigated models. The reason for this design choice in Dezyne is that data that influences the behaviour substantially increases the size of state spaces, and hence hampers the possibility to verify properties.

Using these interface definitions, every AD of the original SysML specification is modelled as a component in Dezyne. We used two different styles of modelling. In the first style, components query the information they need from other components on-demand to determine their response to events. We call this the pull style. In the second style, components push all information that other components might need to these components whenever it becomes available.

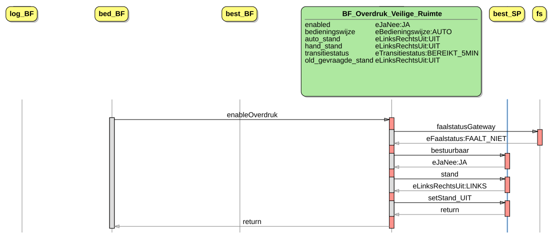
4.1 The pull style model

When using pull style models, the components query their required interfaces for all information they need to handle incoming events. The main benefit of this modelling approach is that it does not require introducing shadow variables to store the last received state information in multiple components. This prevents a large growth in the state space [7] caused by copies of values which do not atomically change due to the delay in propagation throughout the system. To achieve this, instead of having parameterised in-events to communicate the value of some property, an out-event is defined to indicate that the component interacting over the interface wants to know the value. In response to this event, the communicating component must reply with any of the enumeration literals permitted by the interface definition. With this approach, it is possible to generate the complete state space of each component and analyse those using the formal analysis in mCRL2. A typical trace is given in Figure 7a. Under substantial abstractions, we could even show that this model was bisimilar to the SysML model.

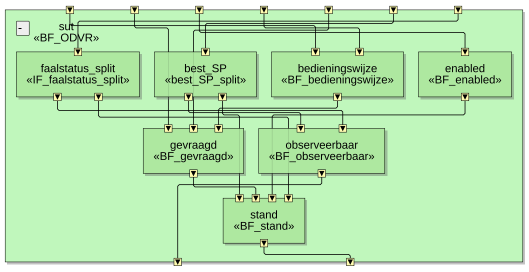
The main downside of this approach is that it does not benefit of most of the verification features offered by Dezyne itself. In the pull style, we need to model interfaces mainly in a stateless way. This has as a consequence that the verification methods that Dezyne offers such as the compliance verification of the implementations in components largely loose their value. Verification of the system as a whole is also troublesome as generating the overall behaviour from the components is currently not feasible.

4.2 The push style model

To make more use of the verification capabilities and align with the reactive components principles of Dezyne, the system has also been modelled in a push style [7]. In this style all components forward information to other components as soon as it becomes available. This approach makes it possible to define interfaces with meaningful states and rely on the verification of the Dezyne tooling. We modelled the overpressure system in the same style as in SysML, using ‘diamond patterns’ where subcomponents are controlled by multiple super-components, which is a style that Dezyne does not encourage. This forces us to use so called *optional trigger events* to push information to various super-components. Unfortunately, this causes the interfaces to grow, which cannot be handled by the verification engine of Dezyne. Even when the Dezyne tooling is executed on a large server with 3TB of memory, the verification of interfaces and component compliance with these interfaces takes days or does not finish within a week.


log_BF
bed_BF
best_BF
BF_Overdruk_Veilige_Ruimte
enabled eJaNee:JA
bedieningswijze eBedieningswijze:AUTO
auto_stand eLinksRechtsUit:UIT
hand_stand eLinksRechtsUit:UIT
transitiestatus eTransitiestatus:BEREIKT_5MIN
old_gevraagde_stand eLinksRechtsUit:UIT
best_SP
fs
enableOverdruk
faalstatusGateway
eFaalstatus:FAALT_NIET
bestuurbaar
eJaNee:JA
stand
eLinksRechtsUit:LINKS
setStand_UIT
return
return

(a) A trace of the overpressure system in the pull model.



(b) Decomposition into many computation components.

Figure 7: Model visualisations of Dezyne.

We remedy this situation by decomposing the system into many more components which compute standalone values providing and requiring much smaller interfaces. This is illustrated in Figure 7b. With this decomposition and abstraction, Dezyne is able to verify its standard properties on all components and interfaces.

4.3 Verification of general properties

Applying the built-in verification techniques to Dezyne models is known to have a very beneficial effect on the quality of the models both in development time and number of faults [8]. However, we fail to verify those properties over the whole state space that were so useful in increasing the primary quality of the tunnel model in [11]. In the pull style it was impossible to generate the overall state space of the system although the Dezyne toolset provides some means to do so. In push styles state spaces will become much bigger [7]. Although we feel that the Dezyne toolset could do a better job in state space generation, we do not expect this to be available for quite some years to come, especially because the way Dezyne generates a full state space does not employ symbolic techniques or parallelism [6, 14].

5 Conclusions

In this paper, the promising results in [11] are being elaborated to come to an industrially viable environment in which existing SysML models of road tunnel control systems can be analysed. In the first approach, a restrictive extension to the SysML specification is introduced which precisely specifies details with respect to the behaviour of the system which were either written down in natural language or were completely absent. While this approach allows the generation of the complete state space of sub-systems and formal verification of both liveness and safety properties over these systems, two major problems became obvious. The first one is that the modelling style of RWS leads to transition systems with massive fanout hampering verification when systems become large. More worrying is that the SysML modelling style is leading to an unwieldy cluttering of graphical entities which is very time consuming to make, and utterly hard for humans to keep track of.

As an alternative the commercial modelling language Dezyne has been investigated. This language offers benefits because it verifies a wide range of properties of the models that have been shown to substantially improve the quality of the specification. But these properties are primarily verified on individual components, and as it stands it does not appear to be easily possible to systematically verify the global properties of tunnel control systems. Although we believe that languages such as Dezyne can and will ultimately be developed further, they are as it stand insufficient as a work horse for tunnel control model.

This opens up the question on the next step within organisations such as Rijkswaterstaat as the possibility to verify has clearly shown itself indispensable in [11]. As it stands we believe that proceeding with commercially available tools is not the most profitable way to go. Academically developed specification languages such as LOTOS NT [5] and mCRL2 [4] might be more suitable as they have flexible and very expressive formalisms to express behaviour and correctness properties, and are actually supported by stable and capable toolsets that are available and reliable for decades. The interesting question is whether these academically developed languages will be picked up as primary tools within the context of for instance Rijkswaterstaat.

References

[1] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet & Eelco Visser (2018): *Scopes as Types*. *Proc. ACM Program. Lang.* 2(OOPSLA), doi:10.1145/3276484.

[2] Rutger van Beusekom, Jan Friso Groote, Paul Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa & Tim A. C. Willemse (2017): *Formalising the Dezyne Modelling Language in mCRL2*. In Laure Petrucci, Cristina Seceleanu & Ana Cavalcanti, editors: *Critical Systems: Formal Methods and Automated Verification*, Springer International Publishing, Cham, pp. 217–233, doi:10.1007/978-3-319-67113-0_14.

[3] Rutger van Beusekom, Bert de Jonge, Paul Hoogendijk & Jan Nieuwenhuizen (2021): *Dezyne: Paving the Way to Practical Formal Software Engineering*. *Electronic Proceedings in Theoretical Computer Science* 338, p. 19–30, doi:10.4204/eptcs.338.4.

[4] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems*. In Tomáš Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[5] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2011): *CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes*. In Parosh Aziz Abdulla & K. Rustan M. Leino, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 372–387, doi:10.1007/978-3-642-19835-9_33.

[6] Jan Friso Groote, Kevin H. J. Jilissen, Maurice Laveaux, P. H. M. van Spaendonck & Tim A. C. Willemse (2022): *Using the Parallel ATerm Library for Parallel Model Checking and State Space Generation*, pp. 306–320. Springer Nature Switzerland, Cham, doi:10.1007/978-3-031-15629-8_16.

[7] Jan Friso Groote, Tim W.D.M. Kouters & Ammar Osaiweran (2015): *Specification guidelines to avoid the state space explosion problem*. Software Testing, Verification and Reliability 25(1), pp. 4–33, doi:10.1002/stvr.1536.

[8] Jan Friso Groote, Ammar Osaiweran & Jacco H. Wesselius (2011): *Analyzing the effects of formal methods on the development of industrial control software*. In: IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011, IEEE Computer Society, pp. 467–472, doi:10.1109/ICSM.2011.6081983.

[9] Joint IEEE / EIA Working Group (1996): *Standard for Information Technology–Software Life Cycle Processes–Software Development–Acquirer-Supplier Agreement (Issued for Trial Use)*, doi:10.1109/IEEESTD.1996.6569022.

[10] Kevin H.J. Jilissen (2022): *A formal analysis of the tunnel control systems of the Rijkswaterstaat GITO*. Master's thesis, Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands. Available at `https://research.tue.nl/en/studentTheses/a-formal-analysis-of-the-tunnel-control-systems-of-the-rijkswater`.

[11] Kevin H.J. Jilissen, Peter Dieleman & Jan Friso Groote (2023): *A formal analysis of Dutch Generic Integral Tunnel Design models*. In: SAC '23: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, Association for Computing Machinery, Inc, United States, pp. 1681–1684, doi:10.1145/3555776.3577786. 38th Annual ACM Symposium on Applied Computing, SAC 2023, SAC 2023 ; Conference date: 27-03-2023 Through 31-03-2023.

[12] Karl Trygve Kalleberg & Eelco Visser (2007): *Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT*. In: SeventhWorkshop on Language Descriptions, Tools, and Applications (LDTA'07), Portugal, pp. 47–50. Available at `https://api.semanticscholar.org/CorpusID:15184499`.

[13] Lennart C.L. Kats & Eelco Visser (2010): *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs*. SIGPLAN Not. 45(10), p. 444–463, doi:10.1145/1932682.1869497.

[14] Maurice Laveaux, Wieger Wesselink & Tim A. C. Willemse (2022): *On-The-Fly Solving for Symbolic Parity Games*. In Dana Fisman & Grigore Rosu, editors: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II, Lecture Notes in Computer Science 13244, Springer, pp. 137–155, doi:10.1007/978-3-030-99527-0_8.

[15] Lucas Lima, André Didier & Márcio Cornélio (2013): *A Formal Semantics for SysML Activity Diagrams*. In Juliano Iyoda & Leonardo de Moura, editors: Formal Methods: Foundations and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 179–194, doi:10.1007/978-3-642-41071-0_13.

[16] Lars Moormann (2022): *Light at the end of the tunnel: Synthesis-based engineering for road tunnels*. Phd Thesis 1 (Research TU/e / Graduation TU/e), Mechanical Engineering. Available at `https://research.tue.nl/en/publications/light-at-the-end-of-the-tunnel-synthesis-based-engineering-for-ro`. Proefschrift.

[17] Pierre Neron, Andrew Tolmach, Eelco Visser & Guido Wachsmuth (2015): *A Theory of Name Resolution*. In Jan Vitek, editor: Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 205–231, doi:10.1007/978-3-662-46669-8_9.

[18] Object Management Group® Standards Development Organization (2015): *OMG Systems Modeling Language*. Technical Report, Object Management Group, Inc. Version 1.5.

[19] Object Management Group® Standards Development Organization (2015): *XML Metadata Interchange*. Technical Report, Object Management Group, Inc. Version 2.1.

[20] Rijkswaterstaat (2021): *Landelijke Tunnelstandaard*. Available at `https://standaarden.rws.nl/link/standaard/6080`. Release 1.2 SP2 B3.

[21] Luís Eduardo de Souza Amorim & Eelco Visser (2020): *Multi-purpose Syntax Definition with SDF3*. In Frank de Boer & Antonio Cerone, editors: *Software Engineering and Formal Methods*, Springer International Publishing, Cham, pp. 1–23, doi:10.1007/978-3-030-58768-0_1.

[22] Sparx Systems: *Enterprise Architect*. Available at `https://sparxsystems.com/products/ea/index.html`. Version 15.

A Artefacts

The artefacts belonging to this paper are available for download at `http://mars-workshop.org/repository.html`. Further instructions are given on the repository page for this paper.

B Generated mCRL2 of the Activity Diagram in Figure 3b

In Figure 3b, the Activity Diagram of the glue layer of the Base Functionality of the Overpressure Safe Space sub-system is shown. This diagram is automatically translated to the mCRL2 process specification below. In this specification, the mapping names and equations starting with `compute_` are generated from the ADs which are called by the Call Behaviour SysML diagram elements. The binding of variables is defined by the assignment in the name of object flows in the glue AD.

```
proc BF_Overdruk_Veilige_Ruimte(
  transitiestatus2 : eTransitiestatus,
  hand_stand : eLinksRechtsUit,
  enabled3 : eJaNee,
  bedieningswijze4 : eBedieningswijze,
  auto_stand : eLinksRechtsUit
) = sum
  best_BF2 : best_Coordinatie_Luchtkwaliteit_Veilige_Ruimte,
  faalstatus2 : MonitoringIntegriteit3BBesturingssysteemVeiligeRuimte_FaalstatusGateway,
  sp_tk2 : besttk_BF_Overdruk_Veilige_Ruimte_SP_Overdruk_Veilige_Ruimte,
  bed_BF : bed_BF_Overdruk_Veilige_Ruimte,
  new_stand3 : eLinksRechtsUit,
  new_stand2 : eLinksRechtsUit,
  setStand2 : SetStand,
  new_gevraagde_stand : eLinksRechtsUit,
  new_bedieningswijze2 : eBedieningswijze,
  observeerbaar4 : eJaNee,
  beschikbaarheid3 : eBeschikbaarheid,
  disabled5 : eJaNee,
  new_enabled : eJaNee,
  new_transitiestatus : eTransitiestatus,
  old_gevraagde_stand : eLinksRechtsUit,
  statusMtkLuiken2 : eOpenDicht
. (
  statusMtkLuiken2 == compute_statusmtkluiken(
    observeerbaar4,
    luikenGesloten2(sp_tk2))
  && old_gevraagde_stand == compute_old_gevraagde_stand(
    auto_stand,
    bedieningswijze4,
    hand_stand)
  && new_transitiestatus == compute_transitiestatus(
    new_gevraagde_stand,
    old_gevraagde_stand,
    transitiestatus2,
    stand5(sp_tk2))
  && new_enabled == compute_enabled2(
    disableOverdruk(bed_BF),
    enableOverdruk(bed_BF),
    enabled3)
  && disabled5 == compute_disabled3(new_enabled)
  && beschikbaarheid3 == compute_beschikbaarheid2(
    bestuurbaar3(sp_tk2),
    faalstatusGateway(faalstatus2),
    storingOverdrukregeling2(sp_tk2),
    storingOverdrukventilatoren2(sp_tk2),
    new_transitiestatus)
  && observeerbaar4 == compute_observeerbaar2(
    bestuurbaar3(sp_tk2),
    faalstatusGateway(faalstatus2),
    redenNietBestuurbaarOpstart(sp_tk2),
    redenNietBestuurbaarStoring3(sp_tk2))
  && new_bedieningswijze2 == compute_bedieningswijze2(
    bedieningswijze4,
    setOpAutobediening2(bed_BF),
    setOpAutobediening(best_BF2),
    setOpHandbediening(bed_BF))
  && new_gevraagde_stand == compute_gevraagde_stand(
    new_stand2,
    new_bedieningswijze2,
    new_stand3)
  && setStand2 == compute_setstand(
    bestuurbaar3(sp_tk2),
    new_enabled,
    faalstatusGateway(faalstatus2),
    new_gevraagde_stand,
    stand5(sp_tk2))
  && new_stand2 == compute_auto_stand(
    auto_stand,
    setAutobedieningsStand(best_BF2))
  && new_stand3 == compute_hand_stand(
    new_bedieningswijze2,
    hand_stand,
    setHandbedieningsStand(bed_BF))
  && true
) ->
  bedtk_BF(bedtk_BF_Overdruk_Veilige_Ruimte(
    new_stand2,
    new_bedieningswijze2,
    beschikbaarheid3,
    bestuurbaar3(sp_tk2),
    disabled5,
    new_enabled,
    new_gevraagde_stand,
    new_stand2,
    luikenGesloten2(sp_tk2),
    observeerbaar4,
    redenNietBestuurbaarPlaatselijkeBediening3(sp_tk2),
    redenNietBestuurbaarStoring3(sp_tk2),
    stand5(sp_tk2),
    statusMtkLuiken2,
    storingAlgemeen3(sp_tk2),
    storingCommunicatieUitgevallen3(sp_tk2),
    GEEN_STORING,
    storingOverdrukregeling2(sp_tk2),
    storingOverdrukventilatoren2(sp_tk2),
    new_transitiestatus
  ))
  | best_sp(best_BF_Overdruk_Veilige_Ruimte_SP_Overdruk_Veilige_Ruimte(setStand2))
  | bed_BF(bed_BF)
  | sp_tk2(sp_tk2)
  | faalstatus2(faalstatus2)
  | best_BF2(best_BF2)
. BF_Overdruk_Veilige_Ruimte(
  enabled3 = new_enabled,
  bedieningswijze4 = new_bedieningswijze2,
  transitiestatus2 = new_transitiestatus,
  hand_stand = new_stand3,
  auto_stand = new_stand2
);
```

C Implementations in Spoofax

As introduced in the paper, the implementation in Spoofax consists of syntax definitions, static analysis rules, and transformation rules. This appendix elaborates on noteworthy details of the implementation and further elaborates on some concepts introduced in the paper. The complete source code is available in Appendix A.

In the implementation, there are three (sub-)syntax definitions in Spoofax. Firstly, there is the definition of the XMI language which is parsed as an Abstract Syntax Tree (AST) from the Enterprise Architect (EA) XMI 2.1 export. Secondly, a sub-language is defined for the syntax of the Assignment-language, which is used in the syntax and semantics definition of some XMI properties. Lastly, there is the definition of the mCRL2 language, to which the input AST is transformed, as defined by van Antwerpen et al. [1]

C.1 Syntax in SDF3

For the syntax definitions, Spoofax allows defining symbols using lexical syntax definitions that match text with a regular expression. The symbols are summarised in Table 3.

Lexical Symbol	Usage
XML-TAG	Opening/closing tags in XML
XML-PROPERTY	Property name on XML tags
XML-STRING	General-purpose property value matching some string
XML-NUM	General-purpose property value matching numbers only

Table 3: Table with lexical symbols for SysML XMI elements.

The SDF3 [21] syntax allows the definition of context-free production rules. These production rules are string templates enclosed by [and] in which symbol names are enclosed in [and] brackets which must be further expanded. With these definitions, parsing an XML structure could be as trivial as defining a context-free symbols `XmlProperty` and `XmlElement` with the production rules in Figure 8.

```
XmlProperty.XmlProperty = [[XML-PROPERTY]="[XML-STRING]"]
XmlElement.XmlElement = [<[XML-TAG][{XmlProperty " "}*]/>]
XmlElement.XmlTree = [
  <[XML-TAG][{XmlProperty ""}*]>
      [{XmlElement ""}*]
  </[XML-TAG]>
]
```

Figure 8: Production rules to parse some XML tree.

While these rules would parse the complete XMI specification successfully, a lot of additional Statix rules would be needed to determine what the generic XML element and its properties represent and what elements are permitted in the subtree rooted at said element. Therefore, the Spoofax language definition contains context-free symbols and production rule definitions for every item encountered in the exported XMI. In these rules, the ordering of XML properties is fixed based on the order they appear in the XMI export for simplicity. When not fixing this ordering, the gained flexibility would cost quite a number of

[1]Available at `https://github.com/MetaBorgCube/metaborg-mcrl2/`

Statix rules for every XMI element to verify that the required properties are given, and that exactly one of them is given. As the exporter in EA has a stable output format, the implementation does not need this flexibility and resulting complexity.

Table 1 in the paper already introduced an abstraction over the actual constructors defined in the XMI language definition. The implementation deviates from this, for the simple reason that the XMI representation of some of the visual elements in SysML is spread out over multiple nodes in the XML structure. Another reason for this deviation, is the ability to re-use common subtree elements in multiple locations. As matching all these elements in a syntax definition is not hard but requires a lot of definitions, interested readers are referred to the artefacts accompanying this paper in Appendix A.

The sub-language for the assignments and expressions is defined. The naming of variables in this context is defined more restrictive, to simplify the translation to mCRL2. We opted for a lexical symbol LANG-ID which matches $[a-zA-Z\_][a-zA-Z\_0-9']^*$. Using this symbol, references to variables are defined as:

```
LangVariableRef.LangVarRoot = [[LANG-ID]]
LangVariableRef.LangVarProp = [[LangVariableRef].[LANG-ID]]
```

In this definition, we define that we can refer to variables with the LangVariableRef symbol by either directly specifying an LANG-ID or reference some property using named with a LANG-ID of some other LangVariableRef using the dot notation.

Assignments are syntactically defined as follows:

```
LangValue.LangVarRef = [[LangVariableRef]]
LangAssignment.LangAss = [[LangVariableRef] := [LangValue]]
```

The LangValue symbol represents a value in the language. As of now, only references to variables defined in the model are permitted as there are no built-in types such as booleans with pre-defined constant values. The assignment itself can then be defined as the string template assigning a value to a variable reference. Similar rules exist to match conditions for the guards.

C.2 Static Semantics in Statix

In Statix, the semantical correctness of the provided document is established by formulating a constraint problem. As discussed in the paper, the packages serve no purpose for the semantics of the SysML diagrams. Therefore, in the static analysis we recursively the tree of packages in the document and conclude that a package is correct if and only if we can conclude that all of its children are correct.

Consider the syntactical constructor UmlPackage(i,n,v,C) as defined in Statix for packages where i is the id, n the name, v the visibility (ignored), and C the list of children. The notation for the constraint to denote the correctness of such package becomes:

$$\frac{\forall_{c\in \mathrm{C}}\ c_{\mathbf{OK}}}{\mathrm{UmlPackage}(i,n,v,C)_{\mathbf{OK}}}$$

Different well-typedness constraints must be satisfied for the package elements based on the actual syntactical constructor type of the element in the AST. In the paper, an example for the enumeration constructor was introduced. A similar inference rule is defined below to establish the well-typedness of Blocks. Additional edges are added to the scope graph which allow the resolution of enumeration literals in assignments and comparisons. Two types of labels are used to facilitate this. An edge $s_b \xrightarrow{L} s_e$ is added when declaring a property of semantical type ENUM. For properties of semantical type BLOCK, an edge

$s_b \xrightarrow{T} s_{b'}$ is added. With the two additional inference rules below, the literals of referenced types using a dot notation can be resolved using path query T^*L.

$$\frac{s_b \vdash P^* \mapsto t_p \colon T_p \quad T_p \equiv ENUM(i_e, n_e, s_e) \quad s_b \xrightarrow{L} s_e \quad s_b \dashrightarrow (n_p, T_p)}{Property(i_p, n_p, t_p)_{\mathbf{OK}}}$$

$$\frac{s_b \vdash P^* \mapsto t_p \colon T_p \quad T_p \equiv BLOCK(i_{b'}, n_{b'}, s_{b'}) \quad s_b \xrightarrow{T} s_{b'} \quad s_b \dashrightarrow (n_p, T_p)}{Property(i_p, n_p, t_p)_{\mathbf{OK}}}$$

There are two inference rules to establish the well-typedness of a property. In the first rule, the premise is a declaration $t_p \colon T_p$ where T_p is of semantic type ENUM. In the second rule, the premise is a declaration $t_p \colon T_p$ where T_p is of semantic type BLOCK. Based on this distinction, the edge with the correct label is declared in the scope graph. Using these two inference rules, an inference rule for blocks similar to the one for enumerations is defined.

$$\frac{\nabla s_b \quad s_b \xrightarrow{P} s_r \quad T \equiv \text{BLOCK}(i, n, s_b) \quad s_r \dashrightarrow (i, T) \quad \forall_{Property(i_p, n_p, t_p) \in P}\, Property(i_p, n_p, t_p)_{\mathbf{OK}}}{\text{Block}(i, n, P)_{\mathbf{OK}}}$$

C.3 Transforming in Stratego

The transformation of the XMI AST to a mCRL2 AST is performed using term rewriting in Stratego. In Stratego, typed term rewriting strategies can be formulated. Everything is represented internally by terms, e.g. the syntactic constructors in AST nodes, AST annotations, scope graph nodes, edges, declarations, and semantic types. Usually, the source AST is traversed and transformed to the target AST. As the structure of the XMI AST is completely different than the target AST, some deviations are made. An example of such deviation follows.

Sort definitions are not created by traversing the AST for the definitions. Instead, they are created by querying the scope graph and translating the semantic types instead of the syntactical constructors. Consider the strategy `block-to-mcrl2` in Figure 9. This strategy transforms a term of the type of semantic types TYPE to a term of syntactic type MCRL2-SortDecl. The latter type is a syntactical constructor part of the mCRL2 syntax specification.

```
block-to-mcrl2(|stx, Hashtable, IndexedSet) :: TYPE -> MCRL2-SortDecl
block-to-mcrl2(|stx, table, set) :
    BLOCK(i, n, s) -> SortAlias(n', Struct([ConstrDecl(n', [l'], [])]))
  with
    n' := <id-to-mcrl2; unique-var-name(|stx, table, set)> n;
    l := <query-var(|stx)> s;
    l' := MCRL2-ConstrDecl-Projs(<map(block-var-to-mcrl2(|stx, table, set))> l)
```

Figure 9: Term rewriting strategy for semantic type BLOCK.

The strategy is specified in Stratego term rewriting rules by applying other term transformation strategies. A unique but consistent name n' is generated, `query-var` queries the scope graph for the data declarations in scope s which are also transformed to mCRL2 using the `block-var-to-mcrl2` strategy. An interesting rule is the declaration of term `l`. The `query-var` strategy, which is passed the static analysis results in term `stx`, queries the scope var for the datums of relation : in scope s. Applying this strategy to the blocks in Figure 5a, the mCRL2 sort specification in 10 is produced.

```
sort
  SetOpAutobediening = struct SetOpAutobediening(bepaaldheid4 :eBepaaldheid);
  SetOpHandbediening = struct SetOpHandbediening(bepaaldheid5 :eBepaaldheid);
```

Figure 10: The generated mCRL2 sort specification corresponding to blocks Figure 5.

D Dezyne pull-style models

In the Dezyne pull-style models, components only retain their own local state. They do not store any information which can be retrieved from other components. In order to perform their desired behaviour, the components must poll other components their state before being able to execute the right commands.

The interface definitions of the controlling component and the feedback loops must be merged, as Dezyne does not permit circular port bindings. As the feedback is now only supplied on demand, both the actions in the original controlling interface and the status updates in the feedback loop interface are defined as in-events on the interface provided by the controllable component. The controlling component requires this interface, and thus can query the status and executes the commands on the controllable component.

Consider again the interface definitions containing the `SetOpHandbediening` command and the `SetOpAutobediening` command depicted in Figure 5a. Assume that the component is also able to query the current control mechanism (`bedieningswijze` in Dutch) of the controlled component. The Dezyne representation of such interface is given in Figure 11.

```
import enums.dzn;

interface bed_BF_Overdruk_Veilige_Ruimte
{
  // control
  in void setOpHandbediening();
  in void setOpAutobediening();

  // feedback
  in eBedieningswijze bedieningswijze();

  behavior {
    // control
    on setOpHandbediening: ;
    on setOpAutobediening: ;

    // feedback
    on bedieningswijze: reply(eBedieningswijze.AUTO);
    on bedieningswijze: reply(eBedieningswijze.HAND);
  }
}
```

Figure 11: Example Dezyne interface snippet for the pull-style models.

According to this interface specification, the controlling component cannot directly observe any change in state as a result of the commands. The reason for the choice of this representation is that the component must not push its state to this component. Consider, for example, a second component controlling the control mechanism of this component. Even if we just sent one of the control mechanism commands, some other component might already have overwritten it. Therefore, on any further interaction, we must poll the control mechanism using the `bedieningwijze` event. An example component snippet in this modelling style which controls the interface defined in Figure 11 is given in Figure 12.

In multiple cases in the model, polled values are needed in several stages of the computation. In all such cases, the computation functions in Dezyne are parameterised, and the initial in-event which starts the thread of execution in the Dezyne component is made responsible for polling all the required state for dealing with said event. The polled values are passed as parameter to the functions to guarantee that, within the thread of execution of handling the event, all polled values are consistent and the amount of

```
import enums.dzn;
import bed_BF_Overdruk_Veilige_Ruimte.dzn;
import some_interface.dzn;

component controller
{
  provides some_interface some_if;
  requires bed_BF_Overdruk_Veilige_Ruimte bed_BF;

  behavior {
    on some_if.switchToManual(): {
      eBedieningswijze current = bed_BF.bedieningwijze();
      if (!current.HAND) {
        bed_BF.setOpHandbediening();
      }
    }
    on some_if.switchToAuto(): {
      eBedieningswijze current = bed_BF.bedieningwijze();
      if (!current.AUTO) {
        bed_BF.setOpAutobediening();
      }
    }
  }
}
```

Figure 12: Example Dezyne component snippet for the pull-style models.

message passing between components is minimised.

The verification performed by Dezyne of proper implementations in components is rather trivial for such interface definitions. The interface specifications are trivially deadlock free as it is always possible to send commands and status queries. Given that the interface behaviour specification is stateless, Dezyne merely has to check whether the events defined in the interface are legal to receive at all times, and all replies to said events are guaranteed to be sent with a value defined in the interface.

E Dezyne push-style models

In the Dezyne push-style models, components are aware of the state of the components they directly communicate with over a port. In order to perform their desired behaviour, the components no longer need to poll other components as they have a decently recent representation of the state of other components. As the state is not guaranteed to be the truth due to the propagation delay between components, components must be resilient against ‘unexpected’ commands. As most commands are variable updates, with notifications about said updates, an idempotent implementation suffices in most cases.

Again, consider the interface description given in Figure 11, but now for push-style models. As discussed in the paper, Dezyne does currently not permit passing enumeration data as parameter in the events. Therefore, the events are duplicated with the data encoded in the event names. Dezyne uses the `optional` event to indicate that something might or might not spontaneously happen over an interface. The representation in the interface is given in Figure 13.

```
import enums.dzn;

interface bed_BF_Overdruk_Veilige_Ruimte
{
  // control
  in void setOpHandbediening();
  in void setOpAutobediening();

  // feedback
  out void bedieningswijze_HAND();
  out void bedieningswijze_AUTO();

  behavior {
    eBedieningswijze bedieningswijze = eBedieningswijze.AUTO;

    // control
    [!bedieningswijze.HAND] on setOpHandbediening: { bedieningwijze = eBedieningswijze.
        HAND; }
    [!bedieningswijze.AUTO] on setOpAutobediening: { bedieningswijze = eBedieningswijze.
        AUTO; }

    // feedback
    on optional: { bedieningswijze_HAND; bedieningwijze = eBedieningswijze.HAND; }
    on optional: { bedieningswijze_AUTO; bedieningswijze = eBedieningswijze.AUTO; }
  }
}
```

Figure 13: Example Dezyne interface snippet for the push-style models.

To remove the need to introduce shadow variables to keep track of the state of the interface, the Dezyne 2.18 release is used for the verification. In the 2.18 version, Dezyne introducees a feature which allows modellers to reference interface variables in the specification of components. Using this feature, the component of Figure 12 can be specified in the push-style as shown in Figure 14.

In this implementation, the component accepts incoming events for remote changes of the control mechanism without explicitly handling such change. By the definition of the interface in Figure 13, the interface variable changes due to the occurrence of this event. In the push-style models, it is sometimes necessary to notify other interfaces of this change, such as emitting an out-event on the `some_if` port. More specifically, it is sometimes necessary to notify multiple ports of the change of some component state variable due to the change of a required port. Dezyne forbids emitting man out-event to more

```
import enums.dzn;
import bed_BF_Overdruk_Veilige_Ruimte.dzn;
import some_interface.dzn;

component controller
{
  provides some_interface some_if;
  requires bed_BF_Overdruk_Veilige_Ruimte bed_BF;

  behavior {
    on some_if.switchToManual(): {
      if (!bed_BF.HAND) {
        bed_BF.setOpHandbediening();
      }
    }
    on some_if.switchToAuto(): {
      if (!bed_BF.AUTO) {
        bed_BF.setOpAutobediening();
      }
    }
    on bed_BF.bedieningswijze_HAND(): {}
    on bed_BF.bedieningswijze_AUTO(): {}
  }
}
```

Figure 14: Example Dezyne component snippet for the push-style models.

than one provides port when dealing with a received out-event of a requires port. These kind of Y-forks potentially leads to behaviour which is beyond the scope of single component verification, and would thus invalidate the guarantees of Dezyne. Similarly, Dezyne forbids V-forks caused by emitting an out-event during the handling of an in-event on other port than the one over which the in-event was received.

To solve this problem, the `defer` statement of Dezyne is used. This enqueues the execution of some block of statements until at least after the handling of the current event is finished. Consider the `bedieningswijze-HAND` event handler in Figure 14. If we were to notify ports `p1` and `p2` of this change, this would be modelled in the used modelling style as shown in Figure 15

```
on bed_BF.bedieningswijze_HAND(): {
  defer() { p1.bedieningswijze_HAND(); }
  defer() { p2.bedieningswijze_HAND(); }
}
```

Figure 15: Example use of deferred pushing of state.

Now, every deferred execution is interacting with a single port and meets the requirements set by Dezyne. However, there arises a different problem with such implementation. As interfaces can now arbitrarily cause the emission of deferred events in a cycle of state change commands, the internal buffers in Dezyne can overflow and verification will fail. As a remedy to this phenomenon, additional state is introduced in every interface to indicate that the interface is ‘idle’. An interface is considered ‘idle’ in this context if the component implementing the interface is finished with executing the deferred statements directly caused by the handler of an in-event on that interface. The overall structure of behaviour specifications in interface definitions becomes as shown in Figure 16, together with the component implementation structure in Figure 17.

While this structure forms a good basis for simulating the models using the Dezyne tooling, verifica-

```
behavior {
  idle = true;
  // other variables

  [idle] {
    // control commands
    on some_command: { ...; idle = false; }
  }
  [!idle] on inevitable: { done; idle = true; }

  // feedback optional events
}
```

Figure 16: Example Dezyne interface snippet for the push-style models with idle state.

```
    on bed_BF.bedieningswijze_HAND(): {
      defer() { p1.bedieningswijze_HAND(); }
      defer() { p2.bedieningswijze_HAND(); }
      defer() { bed_BF.done(); }
    }
```

Figure 17: Example Dezyne component implementation snippet for the push-style models with done statement.

tion is still troublesome. The large combinatorial complexity of all variables in the interface definitions, together with the way the Dezyne semantics are encoded in mCRL2, still yields no results after a week of computations. This is deemed as infeasible in practice. The final remedy introduced in the paper, is further decomposing the components. Similar to the SysML extension, Dezyne components are introduced that are responsible for the computation of individual values. Then, the original Dezyne component is replaced with a Dezyne system which binds external interfaces to the internal component interfaces.

Using this structure, the verification by the Dezyne tooling has successfully been performed with an event queue and defer queue size of 8. Still, the tooling is not yet able to explicitly generate the state space of the overall system or verify other than the out-of-the-box provided properties for the overall system.

Testing Resource Isolation for System-on-Chip Architectures

Philippe Ledent Radu Mateescu Wendelin Serwe
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP[*], LIG, 38000 Grenoble, France
philippe.ledent@inria.fr radu.mateescu@inria.fr wendelin.serwe@inria.fr

Ensuring resource isolation at the hardware level is a crucial step towards more security inside the Internet of Things. Even though there is still no generally accepted technique to generate appropriate tests, it became clear that tests should be generated at the system level. In this paper, we illustrate the modeling aspects in test generation for resource isolation, namely modeling the behavior and expressing the intended test scenario. We present both aspects using the industrial standard PSS and an academic approach based on conformance testing.

1 Introduction

SoC (System-on-Chip) architectures are being designed and deployed as microcontrollers of embedded systems. An SoC is usually highly configurable in order to perform several specific tasks in numerous devices, including smartphones or objects in the IoT (Internet of Things). SoC security is gaining importance as SoCs become ubiquitous, notably because they are being specially manufactured for heavy usage of machine learning and artificial intelligence in the IoT. Considering the distributed nature of the IoT, software solutions to security are insufficient, because an attacker can easily gain access to some hardware and tamper with it. Hardware attacks consist in forcing an SoC to perform operations in order to access functionalities or information that should normally not be available. A critical security requirement is *resource isolation*, which forbids applications (or programs) running on a same SoC to access data not intended for them.

Ensuring this requirement at hardware level is hence becoming mandatory to strengthen security, but is complex and still leaves two challenging problems. First, there is yet no commonly accepted solution: [16] claims to have found a side channel attack that might be applicable to any microcontroller and enable an attacker to access data from secure memory. Second, there is yet no commonly accepted approach for validating a proposal for a hardware resource isolation solution: most research focuses on attacking hardware implementations instead of formally validating proposed protocols.

When it comes to IoT devices, microcontroller manufacturers use the ARM Platform Security Architecture[1] which comes with a security specification and the possibility of certification by ARM (PSA-Certified[2]). The ARM Security Models [2] is the open-source ARM architecture for IoT with security concerns. Here, we focus on the resource isolation aspects of the ARM Security Models that are implemented with the notions of *security* (TrustZone [3]) and *privilege* (TrustZone alone not being enough [12]). ARM provides the possibility to carry security and privilege over the hardware through signals of its AMBA communication protocols [1] between a source and a target component. Filtering properly this information can then be left to the target or a dedicated component on the way in charge of monitoring the communication.

[*]Institute of Engineering Univ. Grenoble Alpes
[1]https://newsroom.arm.com/news/psa-next-steps-toward-a-common-industry-framework-for-secure-iot
[2]https://www.psacertified.org/

Before certifying an SoC by ARM, industrial manufacturers are concerned about representing and testing resource isolation for themselves (the case study [5] showed that ARM-Certified Level 2 may *leak* confidential information such as AES encryption keys). Resource isolation should ensure that data contained in an IP (Intellectual Property, as are components usually called in the hardware community) protected with given security and privilege levels can only be accessed by an IP with corresponding or higher levels. This kind of requirement can be checked using classical tools and techniques for industrial verification, such as hardware simulators using directed tests and/or execution-time assertions. Although properly written assertions are perfect to monitor exactly the behavior of a design under test during a simulation, it is still necessary to generate appropriate test scenarios to be executed: on its own, assertion-based verification cannot generate such scenarios.

The terrifying complexity of modern SoCs pushes to represent and reason about SoC behavior at higher abstraction levels, to ease the fast generation of many tests. For this purpose, PSS (the Portable-test Stimulus Standard) [15] was published by the Accellera Consortium[3] that comprises manufacturers such as AMD, ARM, Intel, Nvidia, NXP, and STMicroelectronics, but also major CAD tool vendors, such as Cadence, Siemens EDA, and Synopsis. PSS aims at providing an easy way to generate (many) tests, without the prior need to explicitly model too much of the SoC's behavior. PSS defines a (programming) language to abstract the behavior of an SoC as a set of "*actions*", which communicate and interact through "*flow objects*". PSS also defines a methodology to generate tests from a VI ("*Verification Intent*", a test scenario given as a partial ordering of the actions) by filling any gaps of the VI with appropriate actions, meeting the ordering constraints expressed for the SoC. Industrial manufacturers are inclined to use PSS, because it uses a familiar syntax (close to C++) and is well integrated in their current design flow and tools.

Although PSS has the appearance of a model-based testing approach, the emphasis is clearly more on the test generation, trying to minimize the time spent on the modeling. Furthermore, because there is no formal semantics of PSS, nor a complete definition of the underlying behavior corresponding to the set of constraints describing an SoC in PSS, the tasks of verification engineers remain difficult. The major challenge faced by these PSS users is getting a grasp on the behavior used as basis for test generation. Frequently, an erroneous constraint is only detected when an unexpected test is generated, limiting the confidence in the quality and coverage of the generated tests.

In this paper, we compare the modeling-related aspects of two approaches for test-case generation, namely the PSS approach with an approach based on conformance test generation with test purposes [10] as supported by the CADP toolbox [7] and its modeling language LNT [8]. Both approaches involve two separate modeling tasks: coming up with an abstract model of the SoC's behavior and expressing the structure of the desired test scenarios. However, the focus of both approaches is different: conformance testing starts with a model, whereas PSS favors modeling the test scenarios. This reflects the needs of verification engineers in the hardware design industry: at the end of the day, they have to produce tests for the SoC, and modeling is is acceptable only if it serves this purpose. We also study the impact of the difference in focus on the generated test suites.

We illustrate both approaches on the problem of generating tests for resource isolation, using a model of an SoC where the details of the various bus communication protocols are abstracted (each transaction is represented by a single rendezvous), because their differences and details are irrelevant to the test case generation. For both approaches, we separately discuss the modeling challenges concerning the behavior of the SoC and the structure of the test scenarios.

Formal verification is slowly being integrated in SoC design and verification workflows as shown in

[3]`https://accellera.org`

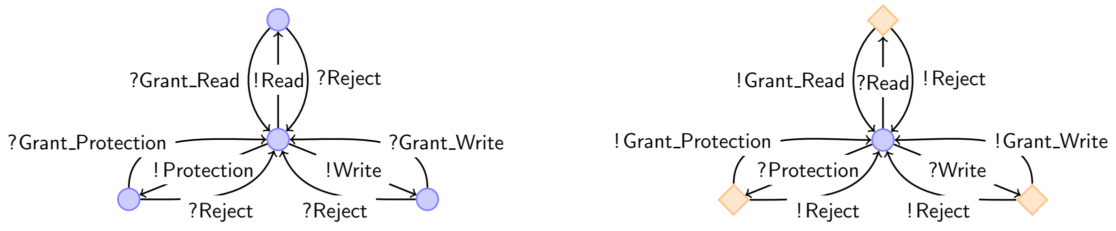


Figure 1: Symbolic automata representation of source (left) and target (right) behaviors

survey [9] but not for testing resource isolation. The closest work to our approach is [4] which proposes a high-level model of Intel 64 and ARMv8-A architectures to compare them but it neither formally specifies the behavior nor is the model used as basis for test generation.

The rest of this paper is organized as follows. Section 2 presents and compares several models of the resource-isolation related SoC behavior in LNT and PSS. Section 3 presents the modeling of test scenarios as test purposes in LNT and verification intents in PSS, together with the resulting test suites (sets of generated tests). Section 4 concludes. The complete LNT and PSS code is given in the appendices and provided in the MARS model repository.

2 Modeling the SoC Behavior for Resource Isolation

We illustrate resource isolation on an SoC with two kinds of IPs (components): sources (e.g., a CPU) and targets (e.g., a memory or dedicated hardware component storing sensible data). All IPs communicate through a bus-like shared interconnect, which can handle a single transaction at a time.[4] Both source and target have a *security level* (secure and non-secure) and a *privilege level* (privileged and non-privileged). Each target stores a data (data1 or data2). Each source can execute transactions to read or write the data of a target, or change the security and/or privilege levels of the data stored by the target. Each transaction consists of an access request emitted by the source, followed by a response (grant or reject) from the target. Each request by the source to the target includes the security and privilege levels of the source, as is the case for any AMBA [1] conform hardware protocol. The target should grant a read or write access if and only if both the security and privilege level of the source are at least those of the target. Concretely, a read or write request is rejected in the two (non-exclusive) situations where the target is secure (respectively, privileged) and the source is not. Changing the security and/or privilege level of the target is only granted to a secure and privileged source. We also allow the source to change its configuration (data to be written and security and privilege levels), thus including the case where a source (CPU) executes applications with different security and privilege levels.

Figure 1 shows the behaviors of a source (on the left) and a target (on the right) as two communicating symbolic automata, focusing on the executed actions and hiding all concrete data as well as security and privilege levels.[5] Both automata synchronize their transitions with identical labels; we omitted the transition corresponding to a change of the source configuration, because it is the only unsynchronized

[4]There are more complex communication protocols enabling a source to initiate further transactions with other targets, but this requires more than one interconnect.

[5]Taking them into account would yield unreadable figures: for instance, the source automaton would have as many different central (idle) states as there are different combinations of security and privilege levels (i.e., sixteen). See also the size of the LTS of the LNT model given in Sect. 2.1.

transition.

All requests are *initiated* by the source (marked with an exclamation-mark “!”) and *received* by the target (marked with a question-mark “?”). The situation is the opposite for granting or rejecting a request (initiated by the target and received by the source). Initially, both automata are in their central state, the source is secure and privileged, and the target is non-secure and non-privileged.

The source can attempt a Read, a Write, or change the Protection level of the target; the target responds by Granting or Rejecting the request depending of whether or not it was legal. For each transaction, the source states its security and privilege levels and moves to the state corresponding to its request, where it awaits a response (grant or reject) from the target, before it can issue the next request.

The target starts by awaiting a transaction request from the source. After reception it enters a state represented by a lozenge symbol indicating that it must analyze the request to determine whether the request will be granted or rejected. This decision constitutes the resource isolation. A request is rejected for security reasons if and only if a non-secure source attempts to access a secure target. Likewise, a request is rejected for privilege reasons if and only if a non-privileged source attempts to access a privileged target. Altogether, a request is accepted if and only if it is not rejected for either reason. Only a secure and privileged request can change the security and privilege levels of the target. Note that the target accepts a new request only in its central state, thus the target accepts a new request only *after* having generated a grant or reject of the pending request (if any).

2.1 SoC Behavior Modeling in LNT

Such an SoC can be expressed easily using LNT [8], a modern language combining a sound foundation in concurrency theory with user-friendly syntax akin to mainstream programming languages. Two LNT processes define the behavior of a source or target, each encoding the corresponding symbolic automaton as an infinite loop, each iteration of which selects among the various possible actions. The overall model of the SoC is obtained as a parallel composition of an instance of as many sources and targets as there are in the SoC. Each communication on the interconnect is modeled as a multiway rendezvous between all instances (reflecting the fact that all IPs can observe everything exchanged on the interconnect). The complete LNT model (about 200 lines) is given in Appendix A.

The LNT model defines four enumerated data types: the security (security) and privilege (privilege) levels[6], the available data values (data), and the identities of the various IPs (ip). For the latter, the model also defines a function source(id) returning **true** if and only if id identifies a source IP. The function valid_access (s, t, p, q) returns **true** if and only if a source with security level t and privilege level q should be granted the request to read or write the data stored in a target with security level s and privilege level p.

Figure 2 shows the LNT process TARGET modeling a target. TARGET has a variable parameter id identifying the IP; the **require**-clause of line 4 enforces that this IP is indeed a target. Among the ten local variables, three record the currently stored data (d), security (s), and privilege (p). The other local variables serve to collect values exchanged during the rendezvous, so as to impose constraints (e.g., that the IP emitting a request is a source or whether the request should be granted or rejected based on the security and privilege level of source and target) or handle data (e.g., change the stored data on line 18 or the security and privilege levels on line 25). Each request is represented by a rendezvous on the corresponding gate (Read, Write, or Protection), during which the source transmits its current security and privilege levels, which the target stores in its local variables t and q (this is indicated by the question

[6]Without loss of generality, we restrict the model to two privilege levels (rather than the four considered by ARM).

```
process TARGET [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                Reject_Write, Protection, Grant_Protection,
                Reject_Protection: Bus] (id: ip) is
   require not (source (id));
   var d,e: data, s,t,u: security, p,q,r: privilege, o, other: ip in
      d := data1; -- default value
      s := non_secure; p := non_privileged; -- lowest protection level
      loop
         select
            Read (?o, id, ?t, ?q) where source (o);
            if valid_access (s, t, p, q) then
               Grant_Read (o, id, d)
            else
               Reject_Read (o, id)
            end if
         [] Write (?o, id, ?t, ?q, ?e) where source (o);
            if valid_access (s, t, p, q) then
               d := e;
               Grant_Write (o, id)
            else
               Reject_Write (o, id)
            end if
         [] Protection (?o, id, ?t, ?q, ?u, ?r) where source (o);
            if (t == secure) and (q == privileged) then
               s := u; p := r;
               Grant_Protection (o, id, s, p)
            else
               Reject_Protection (o, id)
            end if
         -- communication between other IPs on the shared interconnect
         [] Read (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Read (?other, ?o, ?any data)
               where (o != id) and source (other)
         [] Reject_Read (?other, ?o)
               where (o != id) and source (other)
         [] Write (?other, ?o, ?any security, ?any privilege, ?any data)
               where (o != id) and source (other)
         [] Grant_Write (?other, ?o)
               where (o != id) and source (other)
         [] Reject_Write (?other, ?o)
               where (o != id) and source (other)
         [] Protection (?other, ?o, ?any security, ?any privilege,
                        ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Protection (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Reject_Protection (?other, ?o)
               where (o != id) and source (other)
         end select
      end loop
   end var
end process
```

Figure 2: LNT process of a target

marks ? in lines 10, 16, 23, etc.). Depending on the validity of the request, the latter is either granted or rejected (by a rendezvous on the corresponding gate). For a Write and Protection, the grant is preceded by an update of the local variables of the target with the values received from the source during the request (see lines 18 and 25).

The LTS corresponding to a parallel composition of eight sources (which can only initiate the three transactions Read, Write and Protection) and a single target can be generated in less than a minute, and has 182 states, 558 transitions, and 99 labels (after minimization modulo strong bisimulation).

In a second version of the LNT model, a source not engaged in a transaction can also change its configuration (the data written by the source and the security and privilege level of the source). This corresponds to considering sources as multitasking-enabled CPUs capable of executing several applications with different configurations, and to take care of the configuration changes induced by switching between applications. The LTS corresponding to this extended model is too large to be generated—the number of states is expected to be 8^8 times the size of the previous model. However, when removing the identification of the source IP from all transition labels and hiding all transitions corresponding to a configuration change, both LTSs are equivalent for branching bisimulation (the LTS minimized modulo branching bisimulation has 52 states, 268 transitions, and 39 labels).

The identity of the source IP seems thus not important. Indeed, when removing the identification of the source IP from all transition labels and hiding all transitions corresponding to a configuration change, a model with a single multitasking-enabled source also is equivalent for branching bisimulation to the model with eight sources that do not have multitasking enabled. Hence, with the possibility to change the source configuration, it is sufficient to model a single source.

The situation is more intricate concerning the number of targets. Actually, two targets are independent and thus equivalent to a single target with two memory cells with separate security and privilege levels. However, resource isolation is concerned with the access to a single target, so that it is not necessary to study SoCs with more than one target.

It is worth mentioning that the LTS can be analyzed with a full range of verification tools, e.g., those provided by the CADP toolbox. Besides the equivalence checking tools already used to compare the SoCs with different numbers of sources, it is possible to explore the LTS step by step and to verify temporal logic properties. This is helpful to gain confidence in the correctness of the modeled behavior.

2.2 SoC Behavior Modeling in PSS

A major modeling difference between LNT and PSS is that LNT is targeted at modeling the SoC, whereas PSS avoids modeling the overall behavior of the SoC, focusing on simply expressing constraints between the actions of the SoC. However, the latter is less convenient when it comes to precisely understand the modeled behavior, because it requires to assemble all these constraints together.

The understanding of the behavioral model induced by the constraints can be improved by adopting a modeling discipline, such as encoding the two symbolic automata of Fig. 1 (as seen in the previous section, it is sufficient to consider an SoC with a single source and a single target). For each automaton, each transition can be encoded as a PSS action, which inputs from and outputs to a (same) state flow object storing the data values of the automaton, using constraints to enable actions only for particular states of the automaton and controlling the state resulting from the execution of an action. Synchronization between the automata is then expressed using stream flow objects, mimicking the multiway rendezvous on the gates in the LNT model.

This intuitive approach yields the PSS model presented in Appendix B, featuring two state flow objects, nine stream flow objects, and a total of 21 actions (ten actions for the transitions of the source, nine actions for the transitions of the target, plus two actions to control the initial state of the two state flow objects—this is required by the PSS semantics). This significant increase in complexity is accompanied by the need to specify for all actions not only the fields of the state flow object that are modified, but also those that remain unchanged. All in all, the corresponding PSS model ends up with more than 500 lines.

It is possible to translate this PSS model to LNT (using a translator currently under development), leading to almost two thousand lines of LNT. This generic translation encodes each action and flow object as a separate LNT process, leading to a total of 32 processes. The LTS corresponding to each of these processes can be generated and minimized modulo divergence-preserving branching bisimulation, before composing all 32 LTSs into the overall LTS of the PSS model.[7] This generation of the corresponding LTS took about a day (on the yeti cluster in the Grenoble site of the Grid'5000 platform), exploiting the 64 cores using a distributed state space generation tool. However, the corresponding state space (before hiding all transitions related to the interactions between actions and flow objects) is prohibitively large: 1,700,860,640 states, 13,934,786,272 transitions, and 6,706 labels, stored in a file with a size of 88 GB. Note that more refined compositional generation strategies (e.g., smart generation [6]) did not succeed, as some intermediate state spaces for a subset of the processes are larger than the overall state space.

Taking into account that a rendezvous between several actions yields a unique visible transition, we investigated a simpler modeling approach encoding a monolithic automaton, incorporating the constraints of both source and target. This approach requires only ten actions (three requests, three grants, three rejects, and the configuration change), all inputting from and outputting to a single state flow object. The corresponding PSS code is given in Appendix C. The drawback of this approach is the increase in constraints for each action, because it is necessary to specify all fields of the state flow that remain unchanged by the action (each field related to the target is not affected by an action related to the source and vice-versa). Another inconvenient of this approach is that it would be very impractical to extend this model to an SoC with more IPs, due the complexity of getting a complete and correct set of constraints.

This monolithic PSS model can also be translated into (almost one thousand lines of) LNT, from which the corresponding LTS (2736 states, 4591 transitions, and 4592 labels) can be directly[8] generated in less than a minute. After hiding all transitions related to interactions with the state flow object, changing all transition labels to use the same gates and sets of offers as the LNT models of the previous section, and determinization (reduction for weak trace equivalence), the LTS is branching equivalent to those of the LNT models presented in the previous section.

Figure 3 gives the description of action `target_grant_read`. It inputs from and outputs to a state flow object, which keeps track of the configuration of the SoC. Execution of the action is subject to the **constraint**s specified in its body. The first constraint (line 5) enforces that the action can be executed only if another action has already output to the state flow object (each PSS state flow object has an implicit field `initial`, which is initialized to **true**, changed to **false** upon the first output to the flow object, and never changed again). The next two constraints express that the source automaton moves from `read` (line 7 constraining the value of field `sstate` of the input flow object `in_state`) back to `idle` (line 8 constraining field `sstate` of the output flow object `out_state`). The next two constraints (lines 10–13) express the validity of the transaction (inspecting only fields of the input flow object). The remaining eight constraints express that all other fields of the output flow object should keep the values of the fields of the input flow object.

This 24-line PSS description of the action (with its constraints) is more verbose than the correspond-

[7]The translation of stream flow objects makes use of the n-among-m synchronization currently only supported by the EXP.OPEN [11] tool.

[8]Due to the absence of stream flow objects, the generated LNT model does not require a n-among-m synchronization and can thus be handled directly by the LNT compiler.

```
action t_grant_read {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Read to Idle
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.source_sec == secure) ||
             (in_state.target_sec == non_secure);
  constraint (in_state.source_priv == privileged) ||
             (in_state.target_priv == non_privileged);
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}
```

Figure 3: Action for granting a read request in the monolithic PSS model

ing three lines of LNT (lines 13–15 in Fig. 2). This has several reasons. First, in PSS the states of the target have to be listed explicitly, whereas they are deduced from the control flow in LNT. Second, LNT has no implicit field `initial`. Last, but not least, in LNT it is not necessary to specify the variables that maintain their value.

3 Test Generation from Test Scenarios

The principal objective of the models of the SoC behavior presented in Section 2 is to enable the generation of tests to validate the SoC. Characterizing a set of desired tests is a modeling task of its own, based on the idea of expressing a partial ordering of some actions that have to appear in the generated tests, and of relying on tools exploiting the behavioral model to fill in any further actions necessary to obtain a complete test case. This approach emphasizes the expression of a *test scenario* defining the high-level structure of the tests, leaving the details to automatic tools. The notion of test scenario is called TP (*test purpose*) in conformance testing theory [10] and VI (*verification intent*) in PSS.

There are different techniques to construct tests from a test scenario. The TESTOR tool [13] proceeds by a *forward exploration* of a (particular) synchronous product between the TP and the behavioral model, extracting on-the-fly a test or a subgraph called CTG (complete test graph) containing all possible tests for the TP. The PSS methodology [15, Appendix F] uses a *backward traversal* of the VI, determining for each action its immediately necessary previous actions, based on the constraints in the verification intent and the behavioral model. In the following, we compare the effect of these different approaches on four

```
 1  process PURPOSE_1 [
 2              Reject_Read ,
 3              Reject_Write ,
 4              Reject_Protection ,
 5              TESTOR_ACCEPT : none] is
 6     select
 7        Reject_Read
 8     [] Reject_Write
 9     [] Reject_Protection
10     end select ;
11     loop TESTOR_ACCEPT end loop
12  end process
```

```
 1  action intent_1 {
 2    t_reject_read        Reject_Read ;
 3    t_reject_write       Reject_Write ;
 4    t_reject_protection  Reject_Protection ;
 5    activity {
 6      select{
 7          Reject_Read ;
 8          Reject_Write ;
 9          Reject_Protection ;
10      }
11    }
12  }
```

Figure 4: Test scenario 1 ("reject for any reason") as TP in LNT (left) and VI in PSS (right)

test scenarios for resource isolation.

3.1 Test Scenario 1: Reject for any Reason

A natural first test scenario for resource isolation is to search for tests featuring the detection of an illegal transaction, i.e., containing any of the three actions Reject_Read, Reject_Write, and Reject_Protection. Figure 4 shows how to express this scenario as a TP in LNT and a VI in PSS.

In LNT the TP is encapsulated in a process PURPOSE_1, the gate parameters (lines 2–5) of which are the three actions expected in the scenario plus the special gate TESTOR_ACCEPT indicating the goal of the TP. The behavior of this TP is the sequential composition of a non-deterministic choice (**select** instruction in lines 6–10, choices being separated by "**[]**") among the three actions, followed by a loop indicating the end of the TP.

In PSS the VI is a compound action, referencing the three actions via action handles (lines 2–4). The ordering of actions is specified by the **activity** block (lines 5–11), containing a non-deterministic **select**ion among the three actions (lines 6–10, choices being separated by "**;**").

For this TP, TESTOR generates a CTG (183 states, 567 transitions, and 101 labels) that contains all paths to reach any of the three actions, including paths with granted requests before the rejected one. A CTG can be considered a description of a tester, interacting with the SoC to drive it towards the goal of the TP, by selecting appropriate control actions (or inputs) depending on the outputs observed so far. In general, a CTG contains states, where the tester has to choose among different control actions to be executed. The CTG generated for this TP contains 384 choices, all of which can be covered by a suite of 357 test cases that can be generated automatically using the approach proposed in [14].

For this VI, the PSS backward traversal starts by (non-deterministically) choosing one of the three reject actions, and then determines which other actions must immediately precede, by checking which action could have written values to the state flow object so as to satisfy the input constraints of the selected action. The constraints on the sstate field imply the preceding action must be a request. For Reject_Read and Reject_Write, the constraints on the security and privilege levels imply that in the request, one of these values must be strictly lower than the one of the target. For the Reject_Protection, the constraints imply that the preceding Request_Protection stems from a source that is not both secure and privileged. For the monolithic behavioral model, this backward traversal continues until the action init_system_state

```
process PURPOSE_2 [
            Reject_Read,
            Reject_Write,
            Reject_Protection,
            Grant_Read,
            Grant_Write,
            Grant_Protection,
            TESTOR_ACCEPT: none] is
   par
      Grant_Read
   || Grant_Write
   || Grant_Protection
   || Reject_Read
   || Reject_Write
   || Reject_Protection
   end par;
   loop TESTOR_ACCEPT end loop
end process
```

```
action intent_2 {
  t_grant_read        Grant_Read;
  t_grant_write       Grant_Write;
  t_grant_protection  Grant_Protection;
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    schedule{
        Grant_Read;
        Grant_Write;
        Grant_Protection;
        Reject_Read;
        Reject_Write;
        Reject_Protection;
    }
  }
}
```

Figure 5: Test scenario 2 ("all possible responses" interleaved) as TP in LNT (left) and VI in PSS (right)

is found.[9] In practice, the PSS methodology aims at generating a single test at each invocation. When implemented using a breadth-first backward traversal (as is the case for some industrial PSS tools), this systematically yields any of the shortest possible tests.

3.2 Test Scenario 2: Test all Possible Responses (Interleaving Semantics)

This test scenario aims at observing all responses to the three transactions, in any order using the *interleaving* of the responses as shown in Figure 5. In LNT, the parallel composition operator **par** expresses the interleaving of the different branches separated by "||". In PSS, the **schedule** operator expresses the interleaving of the branches separated by ";".[10]

For this TP, TESTOR computes a CTG with 2649 states and 12,057 transitions; its 8832 choices can be covered with 8328 tests. The size of the CTG is due to the fact that once one of the responses has been observed, it is still possible to observe it before all responses have been observed. Hence, the CTG corresponds to an "unfolding" of the model six times, repeating the complete behavior of the SoC until all responses have been observed.

Searching for short(est) tests, the PSS methodology reduces the number of changes in the security and privilege levels of the source and the target. Therefore, in most tests the security and privilege levels for Grant_Read and Grant_Write (respectively Reject_Read and Reject_Write) are the same, and Grant_Protection and Reject_Protection are inserted where suitable. Notice that the syntactic order of the responses in the VI (and TP) actually corresponds to the shortest sequence. Indeed, because the model starts with a secure and privileged source and a non-secure and non-privileged target, all grants are possible. Increasing the security and/or privilege of the target and appropriately lowering the security and privilege of the source are then sufficient to observe the three rejections.

[9]For the generic PSS behavioral model, both init_source_state and init_target_state have to be found.
[10]The PSS operator **parallel** expresses a parallel execution of different behaviors using several threads.

```
process PURPOSE_3 [
               Reject_Read ,
               Reject_Write ,
               Reject_Protection ,
               Grant_Read ,
               Grant_Write ,
               Grant_Protection ,
               TESTOR_ACCEPT: none] is
    Grant_Read;
    Grant_Write;
    Grant_Protection;
    Reject_Read;
    Reject_Write;
    Reject_Protection;
    loop TESTOR_ACCEPT end loop
end process
```

```
action intent_3 {
  t_grant_read          Grant_Read;
  t_grant_write         Grant_Write;
  t_grant_protection    Grant_Protection;
  t_reject_read         Reject_Read;
  t_reject_write        Reject_Write;
  t_reject_protection   Reject_Protection;
  activity {
    Grant_Read;
    Grant_Write;
    Grant_Protection;
    Reject_Read;
    Reject_Write;
    Reject_Protection;
  }
}
```

Figure 6: Test scenario 3 (“all possible responses” in sequence) as TP in LNT (left) and VI in PSS (right)

3.3 Test Scenario 3: Test all Possible Responses (Sequential Semantics)

Most test generation strategies do not support the interleaving of actions, but require more *directed* specifications enforcing a particular sequence of actions. Test scenario 3 requests once again all possible responses but in a particular order, expressed in LNT and PSS using “;”, as illustrated on Figure 6.

Requesting such a *directed* scenario has consequences on the generated test suite for both LNT and PSS. The CTG generated by TESTOR will contain for two sequential actions of the TP every possible path of the model allowed in between. The CTG has 967 states and 3271 transitions; its 2208 choices can be covered with 2072 tests. This CTG is smaller than the one for test scenario 2, because only a single ordering of responses is requested.

The tests generated by PSS are once again the shortest ones and included in those generated for test scenario 2. This shows that more directed test scenarios limit the set of generated tests.

3.4 Test Scenario 4: Access Data with Different Protection

Using the notions of security and privilege, ARM-PSA diversifies the different levels of protection possible for an IP in an SoC. However, there is the strong assumption of a *trusted administrator* as all requests of a secure and privileged source are necessarily granted. Test scenario 4 expresses that whatever the security and privilege of the target, a source with the same security and privilege can write to the target, and any source with higher security and/or privilege (e.g., the administrator) will be able to read the written data. This scenario requires to express that there should be no change in the security or privilege between the write and read requests.

Test Scenario 4 focuses on how to express the refusal of some behavior. This is illustrated in Figure 7 by describing a corresponding TP in LNT, using the special gate TESTOR_REFUSE (line 8) to indicate that the preceding rendezvous on gate Grant_Protection should be excluded from the generated CTG. The null branch (line 10) of the **select** construct (lines 6–11) allows any other action. The where clause on line 12 guarantees (in combination with the condition on line 11 of Figure 2) that the final read is requested with higher security and/or privilege than the write on line 5.

```
process PURPOSE_4 [Read, Grant_Read, Write, Grant_Protection: Bus,
                   TESTOR_ACCEPT, TESTOR_REFUSE: none] is
   var s,t: security, p,q: privilege, d: data in
      Grant_Protection (?any ip, ip0, ?s, ?p)
      Write (?any ip, ip0, s, p, ?d); -- same s and p as in the previous line
      select
         -- refuse any further rendezvous on gate Grant_Protection
         Grant_Protection (?any ip, ip0, ?s, ?p); loop TESTOR_REFUSE end loop
      [] -- accept all other rendezvous
         null
      end select;
      Read (?any ip, ip0, ?t, ?q) where (s != t) or (p != q);
      Grant_Read (?any ip, ip0, d); -- access data with different security and privilege levels
      loop TESTOR_ACCEPT end loop
   end var
end process
```

Figure 7: Test scenario 4 (“access data with different security/privilege”) as TP in LNT

To the best of our knowledge, PSS has no such means to explicitly request absence of actions from the generated tests. Instead, the scenario has to be made more directed by explicitly including more actions in the VI so as to add constraints on these actions. In particular, the VI allows to **bind** an input flow object of an action a_2 to the output flow object of another action a_1, constraining action-inference and forcing a_1 to immediately precede a_2. The resulting, lengthy VI is given in Appendix C.

4 Conclusion

In this paper, we illustrated the modeling tasks for testing hardware resource isolation using both the approach promoted by the industrial standard PSS and an academic approach based on LNT and conformance testing. Both approaches require a model of behavior and an abstract test scenario, which is refined into concrete tests based on the behavioral model.

Despite these similarities, both approaches differ in the way of generating tests, using a forward (LNT) or backward (PSS) search. This difference not only yields different tests, but also impacts the modeling, due to the trade-off between putting constraints in the behavior model or the test scenario. On the one hand, LNT facilitates a complete, verifiable model of the behavior, from which extensive test suites can be generated with few, short test scenarios. On the other hand, PSS favors focusing on the test scenario (or verification intent), and requires longer test scenarios to obtain longer tests. While this avoids the risk of state space explosion, it comes at the price of losing the coverage guarantees available for conformance testing, in particular in the presence of cyclic behavior. Furthermore, the behavior is often under-constrained in PSS, especially when adding a new action to the behavior.

The models presented in this paper were used in an industrial context. An extended version of test scenario 3 requested in LNT a specific order of attempting each transaction for all combinations of source and target security and privilege levels. Concretely, for each attempted transaction, the source requests to write, to read, and then to change the target’s security and privilege. From the generated CTG, we derived a single long test (including all transaction attempts). This test was included in the nightly non-regression tests for a (confidential) SoC under development, sequentially executing the test for each of the over hundred target IPs of the SoC. This revealed a few cases of bad wiring, unaligned documentation, and misunderstandings between architect, design, and verification engineers.

Because the behavioral model of PSS is hard to grasp, modeling errors are frequently detected only by the generation of unexpected tests. We are currently working on the automated translation of PSS constructs into LNT to support the early analysis of the behavioral model, e.g., by model checking. This also includes guidelines for devising PSS models with an efficient translation to LNT.

References

[1] ARM: *AMBA Specification (Rev 2.0).* Available at `https://developer.arm.com/documentation/ihi0011/a`.

[2] ARM: *Platform Security Model 1.1.* Available at `https://developer.arm.com/documentation/den0128/latest`.

[3] ARM: *Security in an ARMv8 System.* Available at `https://developer.arm.com/documentation/100935/0100/Security-in-ARMv8-A-systems-`.

[4] Guillaume Averlant, Benoît Morgan, Éric Alata, Vincent Nicomette & Mohamed Kaâniche (2017): *An Abstraction Model and a Comparative Analysis of Intel and ARM Hardware Isolation Mechanisms.* In: *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 245–254, doi:`10.1109/PRDC.2017.48`.

[5] Fei Chen, Duming Luo, Jianqiang Li, Victor C. M. Leung, Shiqi Li & Junfeng Fan (2023): *Arm PSA-Certified IoT Chip Security: A Case Study.* *Tsinghua Science and Technology* 28(2), pp. 244–257, doi:`10.26599/TST.2021.9010094`.

[6] Pepijn Crouzen & Frédéric Lang (2011): *Smart Reduction.* In Dimitra Giannakopoulou & Fernando Orejas, editors: *Proceedings of Fundamental Approaches to Software Engineering (FASE'11), Saarbrücken, Germany*, *Lecture Notes in Computer Science* 6603, Springer, pp. 111–126, doi:`10.1007/978-3-642-19811-3_9`.

[7] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes.* *Springer International Journal on Software Tools for Technology Transfer (STTT)* 15(2), pp. 89–107, doi:`10.1007/s10009-012-0244-z`.

[8] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT.* In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *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`.

[9] Tomás Grimm, Djones Lettnin & Michael Hübner (2018): *A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip.* *Electronics* 7(6), doi:`10.3390/electronics7060081`.

[10] Claude Jard & Thierry Jéron (2005): *TGV: Theory, Principles and Algorithms – A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems.* *Springer International Journal on Software Tools for Technology Transfer (STTT)* 7(4), pp. 297–315, doi:`10.1007/s10009-004-0153-x`.

[11] Frédéric Lang (2005): *EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods.* In Judi Romijn, Graeme Smith & Jaco van de Pol, editors: *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands*, *Lecture Notes in Computer Science* 3771, Springer, pp. 70–88, doi:`10.1007/11589976_6`. Full version available as INRIA Research Report RR-5673.

[12] Wenhao Li, Yubin Xia & Haibo Chen (2019): *Research on ARM TrustZone.* *GetMobile: Mobile Comp. and Comm.* 22(3), pp. 17–22, doi:`10.1145/3308755.3308761`.

[13] Lina Marsso, Radu Mateescu & Wendelin Serwe (2018): *TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation*. In Dirk Beyer & Marieke Huisman, editors: *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece*, *Lecture Notes in Computer Science* 10806, Springer, pp. 211–228, doi:10.1007/978-3-319-89963-3_13.

[14] Lina Marsso, Radu Mateescu & Wendelin Serwe (2020): *Automated Transition Coverage in Behavioural Conformance Testing*. In: *32nd IFIP Int. Conference on Testing Software and Systems (ICTSS'20), Naples, Italy*, Springer, pp. 219–235, doi:10.1007/978-3-030-64881-7_14.

[15] Portable Stimulus Working Group (2001): *Portable Test and Stimulus Standard 2.0*. Accellera standards, Accellera Systems Initiative, Elk Grove, CA, USA. Available at https://accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf.

[16] C. Rodrigues, D. Oliveira & S. Pinto (2024): *BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect*. In: *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, Los Alamitos, CA, USA, doi:10.1109/SP54263.2024.00062.

A LNT Model

```
module model_8_1 with ==, != is

--------------------------------------------------------------------

type security is secure, non_secure end type
type privilege is privileged, non_privileged end type

type data is data1, data2 end type

type ip is ip0, ip1, ip2, ip3, ip4, ip5, ip6, ip7, ip8 end type

--------------------------------------------------------------------

channel Bus is
   (source, target: ip),
   (source, target: ip, d: data),
   (source, target: ip, s: security, p: privilege),
   (source, target: ip, s: security, p: privilege, d: data),
   (source, target: ip, s: security, p: privilege, t: security, q: privilege)
end channel

--------------------------------------------------------------------

function valid_access (s,t: security, p,q: privilege): Bool is
   -- returns true iff a source with protection level (t,q) is
   -- allowed to access a target with protection level (s,p)
   return not (((s == secure)     and (t == non_secure))      or
               ((p == privileged) and (q == non_privileged)))
end function

--------------------------------------------------------------------

function source (id: ip) : bool is
   -- returns true iff id is a source IP
   case id in
      ip0 -> return false
   |  any -> return true
   end case
end function

--------------------------------------------------------------------

process SOURCE [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                Reject_Write, Protection, Grant_Protection,
                Reject_Protection: Bus, Change_Source_Config: any]
               (id: ip, in var s: security, in var p: privilege,
                in var d: data, multitasking: Bool) is
   require source (id);
   var o, other: ip in
```

```
      loop
         select
            Read (id, ?o, s, p) where not (source (o));
            select
               Grant_Read (id, o, ?any data)
            [] Reject_Read (id, o)
            end select
         [] Write (id, ?o, s, p, d) where not (source (o));
            select
               Grant_Write (id, o)
            [] Reject_Write (id, o)
            end select
         [] Protection (id, ?o, s, p, ?any security, ?any privilege)
               where not (source (o));
            select
               Grant_Protection (id, o, ?any security, ?any privilege)
            [] Reject_Protection (id, o)
            end select
         [] only if multitasking then
               Change_Source_Config (id, id, ?s, ?p, ?d)
            end if
         -- communication between other IPs on the shared interconnectkbu
         [] Read (?o, ?other, ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Grant_Read (?o, ?other, ?any data)
               where (o != id) and not (source (other))
         [] Reject_Read (?o, ?other)
               where (o != id) and not (source (other))
         [] Write (?o, ?other, ?any security, ?any privilege, ?any data)
               where (o != id) and not (source (other))
         [] Grant_Write (?o, ?other)
               where (o != id) and not (source (other))
         [] Reject_Write (?o, ?other)
               where (o != id) and not (source (other))
         [] Protection (?o, ?other, ?any security, ?any privilege,
               ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Grant_Protection (?o, ?other, ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Reject_Protection (?o, ?other)
               where (o != id) and not (source (other))
         end select
      end loop
   end var
end process

-------------------------------------------------------------------

process TARGET [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                 Reject_Write, Protection, Grant_Protection,
                 Reject_Protection: Bus] (id: ip) is
   require not (source (id));
   var d,e: data, s,t,u: security, p,q,r: privilege, o, other: ip in
      d := data1; -- default value
      s := non_secure; p := non_privileged; -- lowest protection level
      loop
         select
            Read (?o, id, ?t, ?q) where source (o);
            if valid_access (s, t, p, q) then
               Grant_Read (o, id, d)
            else
               Reject_Read (o, id)
            end if
         [] Write (?o, id, ?t, ?q, ?e) where source (o);
            if valid_access (s, t, p, q) then
               d := e;
               Grant_Write (o, id)
            else
               Reject_Write (o, id)
            end if
         [] Protection (?o, id, ?t, ?q, ?u, ?r) where source (o);
            if (t == secure) and (q == privileged) then
               s := u; p := r;
               Grant_Protection (o, id, s, p)
            else
               Reject_Protection (o, id)
            end if
         -- communication between other IPs on the shared interconnect
         [] Read (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Read (?other, ?o, ?any data)
               where (o != id) and source (other)
         [] Reject_Read (?other, ?o)
               where (o != id) and source (other)
         [] Write (?other, ?o, ?any security, ?any privilege, ?any data)
               where (o != id) and source (other)
         [] Grant_Write (?other, ?o)
               where (o != id) and source (other)
         [] Reject_Write (?other, ?o)
               where (o != id) and source (other)
         [] Protection (?other, ?o, ?any security, ?any privilege,
                        ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Protection (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Reject_Protection (?other, ?o)
               where (o != id) and source (other)
         end select
      end loop
   end var
end process

-------------------------------------------------------------------------------
```

```
process SOC [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
             Protection, Grant_Protection, Reject_Protection: Bus,
             Change_Source_Config: any] (multitasking: Bool) is
   par Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
       Protection, Grant_Protection, Reject_Protection
   in
      SOURCE [...] (ip1, secure, privileged, data1, multitasking)
   || SOURCE [...] (ip2, secure, privileged, data2, multitasking)
   || SOURCE [...] (ip3, secure, non_privileged, data1, multitasking)
   || SOURCE [...] (ip4, secure, non_privileged, data2, multitasking)
   || SOURCE [...] (ip5, non_secure, privileged, data1, multitasking)
   || SOURCE [...] (ip6, non_secure, privileged, data2, multitasking)
   || SOURCE [...] (ip7, non_secure, non_privileged, data1, multitasking)
   || SOURCE [...] (ip8, non_secure, non_privileged, data2, multitasking)
   || TARGET [...] (ip0)
   end par
end process

----------------------------------------------------------------------

process SOC_2 [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
               Protection, Grant_Protection, Reject_Protection: Bus,
               Change_Source_Config: any] is
   par Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
       Protection, Grant_Protection, Reject_Protection
   in
      SOURCE [...] (ip1, secure, privileged, data1, true)
   || TARGET [...] (ip0)
   end par
end process

----------------------------------------------------------------------

process MAIN [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
              Protection, Grant_Protection, Reject_Protection: Bus,
              Change_Source_Config: any] is
   SOC [...] (false)
end process

----------------------------------------------------------------------

process PURPOSE_1 [Reject_Read, Reject_Write, Reject_Protection,
                   TESTOR_ACCEPT: none] is
   -- any reject
   select
      Reject_Read
   [] Reject_Write
   [] Reject_Protection
   end select;
   loop TESTOR_ACCEPT end loop
end process
```

```
------------------------------------------------------------------------------

process PURPOSE_2 [Grant_Read, Grant_Write, Grant_Protection, Reject_Read,
                   Reject_Write, Reject_Protection, TESTOR_ACCEPT: none] is
   -- any transaction (all possible outcomes) in any order
   par
      Grant_Read
   || Grant_Write
   || Grant_Protection
   || Reject_Read
   || Reject_Write
   || Reject_Protection
   end par;
   loop TESTOR_ACCEPT end loop
end process

------------------------------------------------------------------------------

process PURPOSE_3 [Grant_Read, Grant_Write, Grant_Protection, Reject_Read,
                   Reject_Write, Reject_Protection, TESTOR_ACCEPT: none] is
   -- any transaction (all possible outcomes) in a seqential order
   Grant_Read;
   Grant_Write;
   Grant_Protection;
   Reject_Read;
   Reject_Write;
   Reject_Protection;
   loop TESTOR_ACCEPT end loop
end process

------------------------------------------------------------------------------

process PURPOSE_4 [Read, Grant_Read, Write, Grant_Protection: Bus,
                   TESTOR_ACCEPT, TESTOR_REFUSE: none] is
   -- granted read with different security/privilege than the preceding write
   var s,t: security, p,q: privilege, d: data in
      Grant_Protection (?any ip, ip0, ?s, ?p);
      Write (?any ip, ip0, s, p, ?d);
      -- forbid any change of the security/privilege of the target
      select
         null
      [] Grant_Protection (?any ip, ip0, ?s, ?p);
         loop TESTOR_REFUSE end loop
      end select;
      Read (?any ip, ip0, ?t, ?q) where (s != t) or (p != q);
      Grant_Read (?any ip, ip0, d);
      loop TESTOR_ACCEPT end loop
   end var
end process

------------------------------------------------------------------------------
```

```
end module
```

B PSS Model

```
component pss_top {

  // ------------------------------------------------
  // Types
  // ------------------------------------------------

  enum data_e {
    data1, data2
  }

  enum security_e {
    secure, non_secure
  }

  enum privilege_e {
    privileged, non_privileged
  }

  // ------------------------------------------------
  // Stream Flow Objects for Communication
  // (three streams per operation, for request, grant, and reject)
  // ------------------------------------------------

  // streams for read

  stream request_read_stream {
    rand security_e  sec;  // security of the source requesting to read
    rand privilege_e priv; // privilege of the source requesting to read
  }
  pool request_read_stream request_read_stream_pool;
  bind request_read_stream_pool *;

  stream grant_read_stream {
    rand data_e data; // read data
  }
  pool grant_read_stream grant_read_stream_pool;
  bind grant_read_stream_pool *;

  stream reject_read_stream {}
  pool reject_read_stream reject_read_stream_pool;
  bind reject_read_stream_pool *;

  // streams for write

  stream request_write_stream {
    rand security_e  sec;  // security of the source requesting to write
    rand privilege_e priv; // privilege of the source requesting to write
    rand data_e      data; // data to be written
  }
  pool request_write_stream request_write_stream_pool;
  bind request_write_stream_pool *;

  stream grant_write_stream {}
  pool grant_write_stream grant_write_stream_pool;
  bind grant_write_stream_pool *;

  stream reject_write_stream {}
  pool reject_write_stream reject_write_stream_pool;
  bind reject_write_stream_pool *;

  // streams for setting the protection

  stream request_protection_stream {
    rand security_e  sec;        // security of the requesting source
    rand privilege_e priv;       // privilege of the requesting source
    rand security_e  next_sec;   // new security
    rand privilege_e next_priv;  // new privilege
  }
  pool request_protection_stream request_protection_stream_pool;
  bind request_protection_stream_pool *;

  stream grant_protection_stream {}
  pool grant_protection_stream grant_protection_stream_pool;
  bind grant_protection_stream_pool *;

  stream reject_protection_stream {}
  pool reject_protection_stream reject_protection_stream_pool;
  bind reject_protection_stream_pool *;

  // ---------------------------------------------------------------
  // Finite State Machine for the Source
  // ---------------------------------------------------------------

  enum source_state_e {
    idle, read, write, change
  }

  state source_state {
    rand source_state_e sstate;
    rand data_e      data;
    rand security_e  sec;
    rand privilege_e priv;
  }
  pool source_state source_state_pool;
  bind source_state_pool *;

  // initialize source
  action init_source {
    input  source_state in_state;
```

```
    output source_state out_state;

    constraint in_state.initial == true;
    // fix initial values (to cut nondeterminism)
    constraint out_state.sstate == idle;
    constraint out_state.sec    == secure;
    constraint out_state.priv   == privileged;
    constraint out_state.data   == data1;
  }

  // source read request
  action s_request_read {
    input  source_state        in_state;
    output source_state        out_state;
    output request_read_stream out_stream;

    constraint in_state.initial == false;
    // idle -¿ read
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == read;
    // maintain fields
    constraint out_state.sec   == in_state.sec;
    constraint out_state.priv  == in_state.priv;
    constraint out_state.data  == in_state.data;
    // write to stream
    constraint out_stream.sec  == in_state.sec;
    constraint out_stream.priv == in_state.priv;
  }

  action s_grant_read {
    input  source_state      in_state;
    input  grant_read_stream in_stream;
    output source_state      out_state;

    constraint in_state.initial == false;
    // read -¿ idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec   == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
    // no constraint on the (thus, random) data read from the input stream
  }

  action s_reject_read {
    input  source_state       in_state;
    input  reject_read_stream in_stream;
    output source_state       out_state;

    constraint in_state.initial == false;
    // read -¿ idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  // source write request
  action s_request_write {
    input  source_state         in_state;
    output source_state         out_state;
    output request_write_stream out_stream;

    constraint in_state.initial == false;
    // idle -¿ write
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == write;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
    // write to stream
    constraint out_stream.data == in_state.data;
    constraint out_stream.sec  == in_state.sec;
    constraint out_stream.priv == in_state.priv;
  }

  action s_grant_write {
    input  source_state in_state;
    input  grant_write_stream in_stream;
    output source_state out_state;

    constraint in_state.initial == false;
    // write -¿ idle
    constraint in_state.sstate  == write;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  action s_reject_write {
    input  source_state        in_state;
    input  reject_write_stream in_stream;
    output source_state        out_state;

    constraint in_state.initial == false;
    // write -¿ idle
    constraint in_state.sstate  == write;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  // source protection change request
  action s_request_protection {
    input  source_state               in_state;
    output source_state               out_state;
    output request_protection_stream  out_stream;

    constraint in_state.initial == false;
    // idle -¿ change
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == change;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
    // write to stream
    // no constraint on the new security and new privilege of the target
    // but source still states its security and privilege
    constraint out_stream.sec  == in_state.sec;
    constraint out_stream.priv == in_state.priv;
  }

  action s_grant_protection {
    input  source_state             in_state;
    input  grant_protection_stream  in_stream;
    output source_state             out_state;

    constraint in_state.initial == false;
    // change -¿ idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  action s_reject_protection {
    input  source_state              in_state;
    input  reject_protection_stream  in_stream;
    output source_state              out_state;

    constraint in_state.initial == false;
    // change -¿ idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  // change application running on the source: modify security, privilege, and data
  action change_source_config {
    input  source_state in_state;
    output source_state out_state;

    constraint in_state.initial == false;
    // stay in idle
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == idle;
    // no constraint: randomly change source security, privilege, and data
    // (change application running on the source)
  }

  // ----------------------------------------------------------------------
  // Finite State Machine for the Target
  // ----------------------------------------------------------------------

  enum target_state_e {
    idle, read, write, change
  }

  state target_state { // Target FSM
    rand target_state_e sstate; // FSM STATE
    // Target internal data
    rand data_e data;               // current data
    rand security_e sec;            // current sec protection
    rand privilege_e priv;          // current priv protection
    // Remember last transaction
    rand security_e tx_sec;         // transaction sec
    rand privilege_e tx_priv;       // transaction priv
    rand data_e tx_data;            // transaction data (write request)
    rand security_e next_sec;       // transaction change sec request
    rand privilege_e next_priv;     // transaction change priv request
  }
  pool target_state target_state_pool;
  bind target_state_pool *;

  action init_target {
    input  target_state in_state;
    output target_state out_state;

    constraint in_state.initial == true;
    // Cut nondeterminism by assigning inital values
    constraint out_state.sstate    == idle;
    constraint out_state.data      == data1;
    constraint out_state.sec       == non_secure;
    constraint out_state.priv      == non_privileged;
    constraint out_state.tx_sec    == non_secure;
    constraint out_state.tx_priv   == non_privileged;
    constraint out_state.tx_data   == data1;
    constraint out_state.next_sec  == non_secure;
    constraint out_state.next_priv == non_privileged;
  }

  // target READ request
  action t_request_read {
    input  target_state in_state;
    input  request_read_stream in_stream;
    output target_state out_state;

    constraint in_state.initial == false;
    // Idle -¿ Read
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == read;
    // save stream data
    constraint out_state.tx_sec      == in_stream.sec;
    constraint out_state.tx_priv     == in_stream.priv;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;

  }

  action t_grant_read {
    input  target_state in_state;
    output target_state out_state;
    output grant_read_stream out_stream;

    constraint in_state.initial == false;
    // Read -¿ Idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.tx_sec == secure) ||
               (in_state.sec == non_secure);
    constraint (in_state.tx_priv == privileged) ||
              (in_state.priv == non_privileged);
    // Write on stream (give the data)
    constraint out_stream.data == in_state.data;
  }

  action t_reject_read {
    input  target_state in_state;
    output target_state out_state;
    output reject_read_stream out_stream;

    constraint in_state.initial == false;
    // Read -¿ Idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.sec == secure) || (in_state.priv == privileged);
    constraint (
                // sec check
                ((in_state.tx_sec  == non_secure) &&
                 (in_state.sec == secure))
                ||
                // priv check
                ((in_state.tx_priv == non_privileged) &&
                 (in_state.priv == privileged))
                );
    // Write on stream (fail verdict)
  }

  // target WRITE request
  action t_request_write {
    input  target_state in_state;
    input  request_write_stream in_stream;
    output target_state out_state;

    constraint in_state.initial == false;
    // Idle -¿ Write
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == write;
    // save stream data
    constraint out_state.tx_sec      == in_stream.sec;
    constraint out_state.tx_priv     == in_stream.priv;
    constraint out_state.tx_data     == in_stream.data;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
  }

  action t_grant_write {
    input  target_state in_state;
    output target_state out_state;
    output grant_write_stream out_stream;

    constraint in_state.initial == false;
    // write -¿ idle
    constraint in_state.sstate  == write;
    constraint out_state.sstate == idle;
    // Check protection
    constraint (in_state.tx_sec == secure) ||
               (in_state.sec == non_secure);
    constraint (in_state.tx_priv == privileged) ||
               (in_state.priv == non_privileged);
    // update data
    constraint out_state.data == in_state.tx_data;
    // maintain fields
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
  }

  action t_reject_write {
    input  target_state in_state;
    output target_state out_state;
    output reject_write_stream out_stream;

    constraint in_state.initial == false;
    // Write -¿ Idle
    constraint in_state.sstate  == write;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.sec == secure) || (in_state.priv == privileged);
    constraint (
                // sec check
                ((in_state.tx_sec  == non_secure) &&
                 (in_state.sec == secure))
                ||
                // priv check
                ((in_state.tx_priv == non_privileged) &&
                 (in_state.priv == privileged))
                );
    // Write on stream (fail verdict)
  }

  // target Protection change request
  action t_request_protection {
    input  target_state in_state;
    input  request_protection_stream in_stream;
    output target_state out_state;

    constraint in_state.initial == false;
    // idle -¿ change
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == change;
    // save stream data
    constraint out_state.tx_sec    == in_stream.sec;
    constraint out_state.tx_priv   == in_stream.priv;
    constraint out_state.next_sec  == in_stream.next_sec;
    constraint out_state.next_priv == in_stream.next_priv;
    // maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_data   == in_state.tx_data;
  }

  action t_grant_protection {
    input  target_state in_state;
    output target_state out_state;
    output grant_protection_stream out_stream;

    constraint in_state.initial == false;
    // Change protection -¿ Idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // Update protection
    constraint out_state.sec  == in_state.tx_sec;
    constraint out_state.priv == in_state.tx_priv;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.tx_sec == secure);
    constraint (in_state.tx_priv == privileged);
  }

  action t_reject_protection {
    input  target_state in_state;
    output target_state out_state;
    output reject_protection_stream out_stream;

    constraint in_state.initial == false;
    // Change protection -¿ Idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (

}
```

C Monolithic PSS Model

This PSS model also includes the four verification intents mentioned in Section 3.

```
component pss_top {
  // ------------------------------------------------------------------
  // Types
  // ------------------------------------------------------------------

  enum data_e {
    data1, data2
  }

  enum security_e {
    secure, non_secure
  }

  enum privilege_e {
    privileged, non_privileged
  }

  // ------------------------------------------------------------------
  // Finite State Machine for the System
  // --------------------------------------------------------

  enum system_state_e {
    idle, read, write, change
  }

  state system_state {
    // State of the FSM encoding the SoC
    rand system_state_e sstate;     // FSM STATE

    // Information about the source IP
    rand security_e   source_sec;   // current source security
    rand privilege_e  source_priv;  // current source privilege
    rand data_e       source_data;  // current source used by source for WRITE

    // Information about the target IP
    rand security_e   target_sec;   // current target security
    rand privilege_e  target_priv;  // current target privilege
    rand data_e       target_data;  // current data stored in target

    // New security and privilege (only meaningful for transaction PROTECTION)
    rand security_e   new_sec;      // new target security
    rand privilege_e  new_priv;     // new target privilege
  }
  pool system_state system_state_pool;
  bind system_state_pool *;

  // --------------------------------------------------------
  // Finite State Machine Actions
  // --------------------------------------------------------

  // Force an initial state
  action init_system {
    input  system_state in_state;
    output system_state out_state;

    // Execute only in the initial state
    constraint in_state.initial       == true;
    // Cut nondeterminism by assigning inital values
    constraint out_state.sstate       == idle;
    // Source (highst security and privilege levels)
    constraint out_state.source_sec   == secure;
    constraint out_state.source_priv  == privileged;
    constraint out_state.source_data  == data1;
    // Target (lowest security and privilege levels)
    constraint out_state.target_sec   == non_secure;
    constraint out_state.target_priv  == non_privileged;
    constraint out_state.target_data  == data1;
    // New target protection
    constraint out_state.new_sec      == non_secure;
    constraint out_state.new_priv     == non_privileged;
```

```
}

// ------------------------------------------------------------------
// When in IDLE state, let the source change it's data and protection.
// This represents the change of the application currently running on the source.
action change_source_config {
  input  system_state in_state;
  output system_state out_state;

  // Do not execute in the initial state
  constraint in_state.initial == false;
  // Stay in idle
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == idle;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
  // Randomly change source security, privilege, and data
  // (change application running on the source)
}

// ------------------------------------------------------------------
// READ

action s_request_read {
  input  system_state in_state;
  output system_state out_state;

  // Do not execute in the initial state
  constraint in_state.initial == false;
  // Move from Idle to Read
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == read;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_grant_read {
  input  system_state in_state;
  output system_state out_state;
```

```
    constraint in_state.initial == false;
    // Move from Read to Idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // Check protection
    constraint (in_state.source_sec == secure) ||
               (in_state.target_sec == non_secure);
    constraint (in_state.source_priv == privileged) ||
               (in_state.target_priv == non_privileged);
    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_sec  == in_state.target_sec;
    constraint out_state.target_priv == in_state.target_priv;
    constraint out_state.target_data == in_state.target_data;
    constraint out_state.new_sec     == in_state.new_sec;
    constraint out_state.new_priv    == in_state.new_priv;
  }

  action t_reject_read {
    input  system_state in_state;
    output system_state out_state;

    constraint in_state.initial == false;
    // Move from Read to Idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;

    // Check protection
    constraint (in_state.target_sec == secure) ||
               (in_state.target_priv == privileged);
    constraint (    // security check
                    ((in_state.source_sec == non_secure) &&
                    (in_state.target_sec == secure))
                 || // privilege check
                    ((in_state.source_priv == non_privileged) &&
                    (in_state.target_priv == privileged)));

    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_sec  == in_state.target_sec;
    constraint out_state.target_priv == in_state.target_priv;
    constraint out_state.target_data == in_state.target_data;
    constraint out_state.new_sec     == in_state.new_sec;
    constraint out_state.new_priv    == in_state.new_priv;
  }
```

```
// ------------------------------------------------------------------------
// WRITE

action s_request_write {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Idle -¿ Write
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == write;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_grant_write {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Write to Idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.source_sec == secure) ||
             (in_state.target_sec == non_secure);
  constraint (in_state.source_priv == privileged) ||
             (in_state.target_priv == non_privileged);
  // update data
  constraint out_state.target_data == in_state.source_data;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_reject_write {
  input  system_state in_state;
  output system_state out_state;
```

```
    constraint in_state.initial == false;
    // Write -¿ Idle
    constraint in_state.sstate  == write;
    constraint out_state.sstate == idle;
    // Check protection
    constraint (in_state.target_sec == secure) ||
               (in_state.target_priv == privileged);
    constraint (    // security check
                    ((in_state.source_sec == non_secure) &&
                     (in_state.target_sec == secure))
                 || // privilege check
                    ((in_state.source_priv == non_privileged) &&
                     (in_state.target_priv == privileged)));
    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_sec  == in_state.target_sec;
    constraint out_state.target_priv == in_state.target_priv;
    constraint out_state.target_data == in_state.target_data;
    constraint out_state.new_sec     == in_state.new_sec;
    constraint out_state.new_priv    == in_state.new_priv;
  }

  // ----------------------------------------------------------------
  // Change PROTECTION of target

  action s_request_protection {
    input  system_state in_state;
    output system_state out_state;

    constraint in_state.initial == false;
    // Move from Idle to Change
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == change;
    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_sec  == in_state.target_sec;
    constraint out_state.target_priv == in_state.target_priv;
    constraint out_state.target_data == in_state.target_data;
    // Randomly select new target security and privilege
  }

  action t_grant_protection {
    input  system_state in_state;
    output system_state out_state;
    constraint in_state.initial == false;
    // Move from Change to Idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // Check protection
    constraint (in_state.source_sec == secure);
    constraint (in_state.source_priv == privileged);
    // Update target protection
    constraint out_state.target_sec  == in_state.new_sec;
    constraint out_state.target_priv == in_state.new_priv;
    // Reset new target protection
    constraint out_state.new_sec     == non_secure;
    constraint out_state.new_priv    == non_privileged;
    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_data == in_state.target_data;
  }

  action t_reject_protection {
    input  system_state in_state;
    output system_state out_state;

    constraint in_state.initial == false;
    // Move from Change to Idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // Check protection
    constraint (// security check
                (in_state.source_sec  == non_secure) ||
                // privilege check
                (in_state.source_priv == non_privileged));
    // Reset new target protection
    constraint out_state.new_sec     == non_secure;
    constraint out_state.new_priv    == non_privileged;
    // Maintain source fields
    constraint out_state.source_sec  == in_state.source_sec;
    constraint out_state.source_priv == in_state.source_priv;
    constraint out_state.source_data == in_state.source_data;
    // Maintain target fields
    constraint out_state.target_sec  == in_state.target_sec;
    constraint out_state.target_priv == in_state.target_priv;
    constraint out_state.target_data == in_state.target_data;
  }

  // ------------------------------------------------------------
  // Verification intents
  // ------------------------------------------------------------

  // Any reject
```

```
action intent_1 {
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    select{
      Reject_Read;
      Reject_Write;
      Reject_Protection;
    }
  }
}

// All responses (interleaving semantics)
action intent_2 {
  t_grant_read        Grant_Read;
  t_grant_write       Grant_Write;
  t_grant_protection  Grant_Protection;
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    schedule{
      Grant_Read;
      Grant_Write;
      Grant_Protection;
      Reject_Read;
      Reject_Write;
      Reject_Protection;
    }
  }
}

// All responses (sequential semantics)
action intent_3 {
  t_grant_read        Grant_Read;
  t_grant_write       Grant_Write;
  t_grant_protection  Grant_Protection;
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    Grant_Read;
    Grant_Write;
    Grant_Protection;
    Reject_Read;
    Reject_Write;
    Reject_Protection;
  }
}

// Access data with different security and/or privilege
```

```
  // This test scenario executes the following steps:
  // 1) Elevate target security/privilege
  // 2) Write data to target with same security/privilege as the target
  // 3) Change source security/privilege, keeping target security/privilege unchanged
  // 4) read the target
  // We did not find how to express unwanted behavior (e.g., how to forbid to change target security/privilege).
  // Thus we request that there is a change of the source configuration immediately after the write transaction
  // was granted and rely on the shortest path to avoid any further change of the target security/privilege before
  // the read transaction.
  action intent_4 {
    change_source_config Change_Source;
    t_grant_read         Grant_Read;
    t_grant_protection   Grant_Protection;
    t_grant_write        Grant_Write;
    activity {
      Grant_Protection;  // get s and p
      Grant_Write;       // do an accepted write with the same s and p
      Change_Source;
      // Rely on shortest path to not do any other Grant_Protection
      Grant_Read;    // do an accepted read with another s or another p
    }

    constraint {
      // Grant_Write with the same security and privilege as Grant_Protection
      Grant_Write.in_state.source_sec ==
        Grant_Protection.out_state.source_sec;
      Grant_Write.in_state.source_priv ==
        Grant_Protection.out_state.source_priv;

      // Read granted to a source different security or privilege as the Write
      ((Grant_Read.in_state.sec  != Grant_Write.out_state.sec) ||
       (Grant_Read.in_state.priv != Grant_Write.out_state.priv));

      // Read with same target security and privilege as Grant_Protection
      // (no guarantee of absence of change in between)
      Grant_Read.in_state.target_sec ==
        Grant_Protection.out_state.target_sec;
      Grant_Read.in_state.target_priv ==
        Grant_Protection.out_state.target_priv;

      // Read the data that was written
      // (no guarantee of absence of change in between)
      Grant_Read.in_state.target_data == Grant_Write.out_state.target_data;
    }

    // Allow nothing between Grant_Write and Change_Source
    // This prevents all other actions because there is only one flow object
    bind Grant_Write.out_state Change_Source.in_state;
  }
}
```

D SVL Script for all Verification Steps

The following SVL script[11] generates and compares the LTSs mentioned in Sections 2.1 and 2.2. It requires the translation to LNT of the monolithic PSS model (available in the MARS model repository).

```
-- generation of the LTS for a SoC with 8 source IPs

"model_8_1.bcg" =
   reduction of "model_8_1.lnt";

-- generation of the LTS for a SoC with a single source IP

"model_2_1.bcg" =
   reduction of
   -- remove all actions from absent sources (only IP1 is present)
   total cut all but "[^!]*␣!IP1␣.*" in
   "model_8_1.lnt":"SOC_2";

-- generation of the LTS for the PSS model

"RI_monolithic.bcg" =
   strong reduction of
   weak trace reduction of
   branching reduction of
   -- 4. suppression of supperfluous offers (to be completed)
   total rename
      "\(CHANGE_SOURCE_CONFIG\)␣.*␣\(![^!]*␣![^!]*␣![^!]*\)␣![^!]*␣![^!]*␣!
         [^!]*␣![^!]*␣![^!]*" -> "\1␣\2",
      "\(GRANT_READ\)␣.*␣\(![^!]*\)␣![^!]*␣![^!]*" -> "\1␣\2",
      "\(GRANT_WRITE\).*" -> "\1",
      "\(GRANT_PROTECTION\)␣.*␣\(![^!]*␣![^!]*\)␣![^!]*␣![^!]*␣![^!]*" -> "
         \1␣\2",
      "\(REJECT_[A-Z]*\).*" -> "\1",
      "REQUEST_\(READ\)␣![^!]*␣\(![^!]*␣![^!]*\)␣.*" -> "\1␣\2",
      "REQUEST_\(WRITE\)␣![^!]*␣\(![^!]*␣![^!]*␣![^!]*\)␣.*" -> "\1␣\2",
      "REQUEST_\(PROTECTION\)␣![^!]*␣\(![^!]*␣![^!]*\)␣.*␣\(![^!]*␣![^!]*\)
         " -> "\1␣\2␣\3"
   in
   -- 3. suppression of the prefix SOURCE/TARGET
   rename
      "SOURCE_\([A-Z_]*\)" -> "\1",
      "TARGET_\([A-Z_]*\)" -> "\1"
   in
   -- 2. removal of the first offer (indicating the action)
   total rename
      "\([^!]*\)!PSS_TOP_X_[^!]*\(.*\)" -> "\1\2"
   in
   -- 1. removal of the gate prefix "PSS_TOP_X_"
   rename
      "PSS_TOP_X_\(.*\)" -> "\1"
```

[11] SVL (Script Verification Language) is the language for describing verification scenarios for the CADP toolbox.

```
    in
    -- hiding initialisation and interaction with the state flow object
    divbranching reduction of
    hide
        ".*OUTPUT",
        ".*INPUT",
        ".*INIT_SYSTEM"
    in
        "../PSS/RI_monolithic.lnt"
    end hide;

-- comparion of the three LTSs

property MODEL_EQUIVALENCE
    "after hiding ip identities, alls models are equivalent"
is
    branching comparison
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^ ]*\) ![^!]* ![^!]*\(.*\)" -> "\1 \2" in
        "model_8_1.bcg"
        ==
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^ ]*\) ![^!]* ![^!]*\(.*\)" -> "\1 \2" in
        "model_2_1.bcg";
    expected TRUE;

    branching comparison
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^ ]*\) ![^!]* ![^!]*\(.*\)" -> "\1 \2" in
        "model_8_1.bcg"
        ==
        hide CHANGE_SOURCE_CONFIG in
        "RI_monolithic.bcg";
    expected TRUE;
end property
```

Formal Verification of Consistency for Systems with Redundant Controllers

Bjarne Johansson
ABB AB, Västerås, Sweden
Mälardalen University, Västerås, Sweden
`bjarne.johansson@se.abb.com`

Bahman Pourvatan Zahra Moezkarimi Alessandro Papadopoulos Marjan Sirjani
Mälardalen University, Västerås, Sweden
`firstname.lastname@mdu.se`

A potential problem that may arise in the domain of distributed control systems is the existence of more than one primary controller in redundancy plans that may lead to inconsistency. An algorithm called NRP FD is proposed to solve this issue by prioritizing consistency over availability. In this paper, we demonstrate how by using modeling and formal verification, we discovered an issue in NRP FD where we may have two primary controllers at the same time. We then provide a solution to mitigate the identified issue, thereby enhancing the robustness and reliability of such systems.

1 Introduction

Control systems are essential in the automation solution of domains such as offshore oil extraction, refineries, and hydropower plants - sectors where downtime can lead to significant financial losses or even life-threatening incidents. These automation solutions incorporate redundancy to mitigate the risk of unplanned downtime due to hardware failures by duplicating critical components like controllers. The common approach is standby redundancy, where an active primary controller manages the process, and a passive backup is ready to take over in case of primary failure [21]. These controllers, or Distributed Controller Nodes (DCN), interact with the physical world through Field Communication Interfaces (FCI), connecting to input/output (I/O) devices. The FCI supplies process values to the DCN, which then executes control actions based on these inputs and sends outputs back to the FCI.

For a backup DCN to seamlessly assume the primary role, it must detect the primary's failure and resume the primary role with the former primary's last known state. The primary cyclically replicates its latest state to the backup and sends a heartbeat, i.e., a message with predetermined intervals for failure detection. Heartbeat absence signifies a possible primary failure. Controller redundancy communication is conventionally carried out over a dedicated, point-to-point connection [27, 18, 20], as illustrated in Figure 1. Failure of the redundancy link can partition the DCN pair, disrupting synchronization and causing their internal states to diverge. This divergence might result in inconsistent outputs to the FCI.

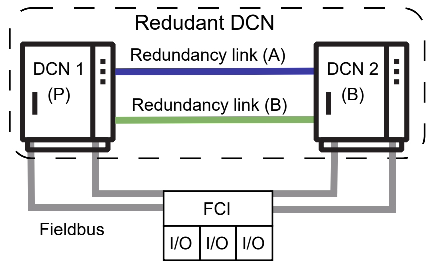


Figure 1: A redundant DCN (controller) pair synchronized with dedicated, redundant redundancy link.

Two strategies are common when managing failures in redundancy communication links: (i) disabling redundancy following the failure of one of the links or (ii) continuing in redundant mode. These strategies reflect the alternatives a distributed system has in case of partitioning: remain consistent and sacrifice availability or vice versa—consequence of the Consistency, Availability, and Partitioning tolerance (CAP) theorem [6].

Disabling redundancy after a redundancy link failure compromises availability, as the backup won't activate if the primary controller fails before the link is repaired. While this method prioritizes consistency, a concurrent loss of both redundancy links can still lead to a dual primary situation [20].

The alternative, operating redundantly with only one functioning redundancy link, risks causing a dual primary situation if the remaining redundancy link fails. This is because the backup can not distinguish missing heartbeats due to a failure of the link from a failure of the primary. Some vendors call a dual primary scenario non-synchronized active units, signifying the consistency compromise following from CAP [18]. Controllers unable to communicate can not synchronize, leading to an inconsistent state in the redundant pair.

The advent of Industry 4.0 is steering industrial controllers towards a network-centric design [2, 4, 16]. As defined by the Open Process Automation Forum (OPAF), the DCNs and FCI are integrated into a cohesive communication network. Additionally, this network backbone can support redundancy communication and replace the redundancy link shown in Figure 1 with a network, see Figure 2.

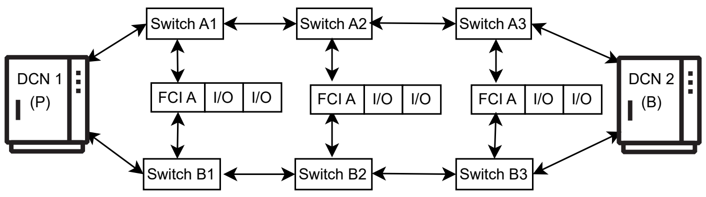


Figure 2: Redundant controllers connected over a redundant, disjoint network backbone.

When communication between a redundant DCN pair fails, as shown in Figure 3a, traditional approaches either disable redundancy at the first failure (F1) or allow the system to operate in a non-synchronized dual-primary mode, as shown in Figure 3b. Johansson et al. [10] introduce the Network Reference Point Failure Detection (NRP FD) for such redundant DCN systems. NRP FD prioritizes consistency while reducing the impact on availability. It uses an external Network Reference Point (NRP) as a tiebreaker for primary role determination, aiding the backup DCN in differentiating between primary and network failures. For a DCN to attain and retain the primary role, it must maintain communication

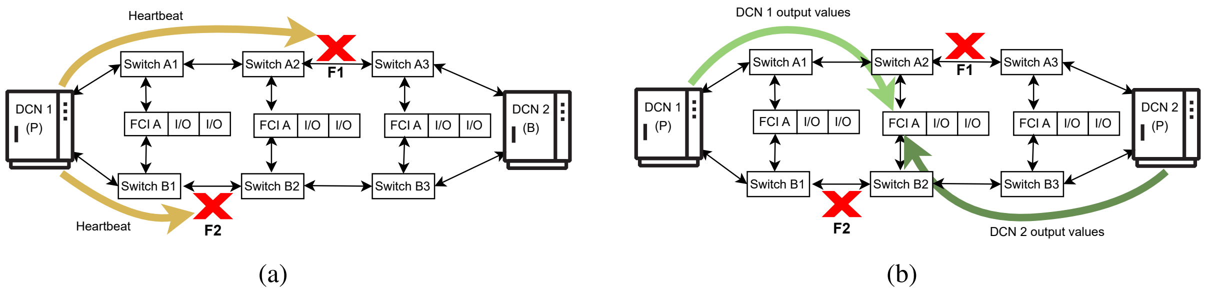


Figure 3: (a) F1 and F2 exemplify network failures partitioning the redundant controller pair, preventing the heartbeat (and other communication) between DCN 1 and DCN 2. (b) Due to F1 and F2 caused partitioning, both DCN 1 and DCN 2 become primary and drive potentially inconsistent outputs.

with the NRP. The importance of addressing dual primary risks is emphasized in manuals recommending spatially separated redundancy links in current systems to avoid simultaneous damage and undefined system states [20].

To design an algorithm that guarantees the uniqueness of the primary the following questions need to be answered:

- How should the backup know about a failure?
- When should the backup become a primary?

As described by Johansson et al. in [10], the NRP FD uses heartbeats for primary failure detection (heartbeat) and a separate message for NRP reachability testing and detecting network failure. This introduces a potential vulnerability: the absence of a heartbeat is a sign of the primary failure, while NRP reachability is verified separately. Consequently, temporary disturbances could lead to inconsistencies, underscoring the importance of testing with temporal disturbances. Hence, one other question also have to be answered:

- How should we take care of the transient errors in switches or DCNs?

Since nondeterministic behavior is generally undesirable in control systems, particularly in high-integrity systems crucial for safety-critical solutions, like the ABB AC 800M High Integrity system [1], we need assurance of the correctness of the algorithm. Therefore, this paper describes in detail the modeling and formal verification of the NRP FD algorithm, considering the main safety property of "NoDualPrimary". We use Timed Rebeca which is an actor-based modeling language for reactive and distributed systems and its model checker tool Afra to model and verify NRF FD. We model different failures including transient errors and illustrate the results. We also propose an enhanced lease-based version of NRP FD that ensures a singular primary in the case of transient errors.

2 Network Reference Point Failure Detection (NRP FD) Algorithm

NRP FD targets failure detection in redundant controller pairs. In a standard system, two controllers, DCN 1 and DCN 2, function as primary and backup, respectively, as illustrated in Figure 4. The primary is unique in the system and interacts with I/O devices, while the backup, in standby mode, activates only upon primary failure. This concept is known as standby redundancy [21]. These controllers, DCN 1 and DCN 2, require communication, typically through a network facilitated by switches [4, 16]. Redundant controllers are often paired with dual independent networks for enhanced reliability, as depicted in Figure 4.

NRP FD is a heartbeat-based failure detection algorithm where the primary controller sends regular heartbeat messages to the backup via the networks connecting the redundant DCN pair [10]. These heartbeats, a push-based failure detection method, involve the primary sending messages to the backup at a known interval [19]. NRP FD differs from traditional heartbeat-based failure detection due to its NRP usage. An NRP must meet two requirements: (i) it should not share common cause failures with the redundant DCN pair, and (ii) be accessible from only one DCN in case of network partitioning. Each controller typically has one NRP candidate per independent network, as illustrated in Figure 4, where network switches serve as potential NRPs. The uper network in Figure 4 includes three switches $Switch\ A1$, $Switch\ A2$, and $Switch\ A3$, and the lower network includes $Switch\ B1$, $Switch\ B2$, and $Switch\ B3$. The NRP candidate set for the primary is $\{Switch\ A1, Switch\ B1\}$ and for the backup is $\{Switch\ A3, Switch\ B3\}$, and $Switch\ A1$ is the NRP.

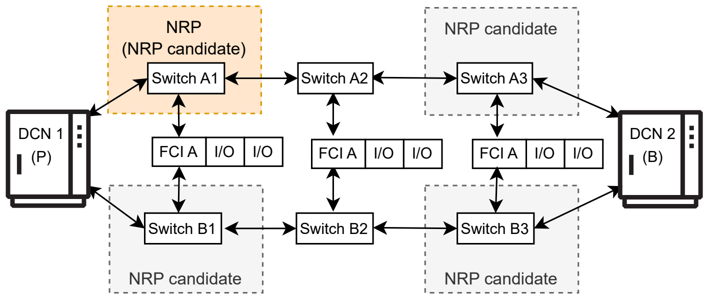


Figure 4: The redundant network backbone with the NRP and NRP candidates highlighted.

The operational procedure of NRP FD is as follows: before enabling redundancy, the primary DCN selects an NRP from the available NRP candidates. The heartbeat message communicates the NRP selection to the backup. The primary continuously monitors the NRP, ensuring its accessibility and proposing a change to the backup if the NRP is unreachable. If the backup doesn't acknowledge this change within a set time, the primary leaves the primary role. Concurrently, the backup continuously monitors heartbeats from the primary. If these are missing for a predetermined duration, the backup assesses its NRP connection. Should this connection be active, the backup takes the primary role. The following section will provide more details of the algorithm and its Timed Rebecca model.

3 Modeling and Verification of NRP FD using Timed Rebeca

We use Timed Rebeca language and its integrated model checker tool, Afra, to model and verify NRP FD. For modeling NRP FD, we have used the description of the protocol and the diagrams provided in [10] as well as several meetings with the industrial partners to clarify the details and choose the appropriate level of abstraction, which we will discuss in the remainder of this section.

3.1 The actor-based language, Timed Rebeca

Rebeca (Reactive Object Language) [26, 22] is an actor-based language designed for modeling and formal verification of reactive concurrent and distributed systems. Actors [8, 3] are units of concurrency. In Rebeca models, reactive objects known as rebecs resemble actors with no shared variables, asynchronous message passing, and unbounded message buffers. Each rebec has a single thread of execution. Communication with other rebecs is achieved by sending messages, and periodic behavior is executed by sending messages to itself. Rebeca has no explicit receive statement, and its send statements are non-blocking. Each rebec has variables, methods (message servers), and a dedicated message queue for received messages. How a rebec reacts to a message is specified in message servers. The rebec processes messages by de-queuing from the top and executing the corresponding message server non-preemptively. The state of a rebec can change during the execution of its message servers through assignment statements.

Rebeca is an imperative language with a syntax similar to Java. A Rebeca model consists of several reactive classes and a main section. Each reactive class describes the type of a certain number of rebecs. Rebecs (actors) are instantiated in the main block. While message queues in the semantics of Rebeca are inherently unbounded, a user-specified upper bound for the queue size is necessary to ensure a finite state space during model checking. Reactive classes include constructors, sharing the same name as the class, responsible for initializing the actor's state variables and placing initially required messages in the actor's message buffer.

In this work, we use Timed Rebeca (the timed extension of Rebeca) [24, 12] with a global logical time. Timed Rebeca considers synchronized local clocks for all actors throughout the model. Instead of a message queue, Timed Rebeca uses a message bag in which messages carry their respective time tags. The sender tags its local time to a message at the time of sending. Timed Rebeca introduces three timing primitives: "delay," "after," and "deadline." A delay statement represents the passage of time for an actor while executing a message server, i.e., it is used to model computation times. All other statements are assumed to execute instantaneously. The keywords "after" and "deadline" are augmented to a message send statement. The term "after(n)" means it takes n units of time for a message to reach its receiver. Using the after construct, we can model network delay and periodic events. We can use a nondeterministic assignment to n, and model nondeterministic arrival times for a message (event). The term "deadline(n)" conveys that if the message is not retrieved within n units of time, there will be a timeout. An abstract syntax of Timed Rebeca is provided in Appendix A. Timed Rebeca is extended with priorities [25]. Priorities are assigned to rebecs and message handlers to control the order of their execution and hence enhance the determinism of the system's behavior [14]. If more than one actor or event are enabled at the same time, then the model checker builds all the possible execution traces, using priorities you can cut some of the branches.

3.2 Modeling NRP-FD in Timed Rebeca

We model Figure 4 using Timed Rebeca. The model is extensible meaning that the number of switches and nodes can be increased. In the Timed Rebeca model each node and each switch is modeled as an actor, their communication is modeled as message passing, and reactions to each message, signal, and timed event are modeled using message servers. A Rebeca model includes reactive class definitions, defining the behavior of the rebecs (actors) within the model. L1[1] illustrates some parts of the Timed Rebeca model for NRP FD.

In the NRP-FD model, we have two different element types, Node and Switch. Each element type is defined as a reactive class, *Node* (L1, line 10) and *Switch* (L1, line 40). Each reactive class has a constructor. A constructor is a unique method which is called when the actor is instantiated. Initialization of the variables is done in the constructor. We instantiate two nodes with ids 100 and 101 and six switches (A1-A3 and B1-B3) in the main section (L1, lines 59-68). A node can be a primary or a backup, and a switch can be a non-terminal switch (not connected to a DCN), an NRP candidate, or an NRP. Each node has an NRP candidate (switch) for each network, i.e., switches A1 and B1 with ids 1 and 4, respectively for DCN1 and switches A3 and B3 for DCN2 with ids 3 and 6, respectively (L1, lines 66-67). The parameters in the instantiation statements are used to set different types and also pass other necessary information to the constructor.

We select DCN1 with id 100 as the primary at the beginning of the algorithm (second parameter in lines 66-67 of L1). There are two known rebecs in the reactive class *Node*, meaning it can send messages to these rebecs. We have a method call in the constructor of the *Node*, i.e., *runMe* (L1, line 22). In *runMe* (L1, line 28) the DCN checks its state using the state variable *mode* and then serves the corresponding behavior (L1, lines 30-34). Note that, the last line of *runMe* (L1, line 35) is a self-call followed by an after with *heartbeat_period* as its parameter, modeling a periodic event, i.e., "$runMe()after(heartbeat\_period)$;". It means that in every *heartbeat_period* (determined in the code L1, line 1), *runMe* is executed. The *heartbeat_period* should be significantly larger than other timing parameters. This is because all events must be handled during a heartbeat interval. Regarding timing

[1]We use L1, L2 and L3 to refer to Listing 1, Listing 2 and Listing 3, respectively.

```
env int heartbeat_period = 1000;
env int max_missed_heartbeats = 2;
env int ping_timeout =500;
env int nrp_timeout = 500;
env byte NumberOfNetworks = 2;
env int switchA1failtime = 2500;
...
env int networkDelay = 1;
env int networkDelayForNRPPing = 1;
reactiveclass Node (4){
    knownrebecs {Switch out1, out2;}
    statevars {...}
    Node (int Myid, int Myprimary, int NRPCan1_id, int NRPCan2_id, int myFailTime) {
        id = Myid;
        NRPCandidates[0] =NRPCan1_id;
        NRPCandidates[1] =NRPCan2_id;
        NRP_network = -1;
        primary = Myprimary;
        mode = WAITING;
        ...
        if(myFailTime!=0) nodeFail() after(myFailTime);
        runMe();
    }
    msgsrv new_NRP_request_timed_out(){...}
    msgsrv ping_timed_out() {...}
    msgsrv pingNRP_response(int mid){...}
    msgsrv new_NRP(int mid,int prim, int mNRP_network, int mNRP_switch_id) {...}
    msgsrv runMe(){
        if(?(true,false)) nodeFail();
        switch(mode){
            case 0: //WAITING : ...
            case 1: //PRIMARY : ...
            case 2: //BACKUP : ...
            case 3: //FAILED : ...
        self.runMe() after(heartbeat_period);
     }
     msgsrv heartBeat(byte networkId, int senderid) {...}
     msgsrv nodeFail(){...}
}
reactiveclass Switch(10){
    knownrebecs {...}
    statevars {...}
    Switch (int myid, byte networkId, boolean endSwitch , Switch sw1, Switch sw2, int myFailTime) {
       mynetworkId = networkId;
       id = myid;
       terminal=endSwitch;
       amINRP = false;
       failed = false;
       switchTarget1 = sw1;
       switchTarget2 = sw2;
       ...
    }
    msgsrv switchFail(){ failed = true; amINRP=false;}
    msgsrv pingNRP_response(int senderNode){...}
    msgsrv pingNRP(int switchNode, int senderNode, int NRP) {...}
    msgsrv new_NRP(int senderNode, int mNRP_network, int mNRP_switch_id) {...}
    msgsrv heartBeat(byte networkId, int senderNode) {...}
}
main {
    @Priority(1) Switch switchA1(DCN1):(1, 0, true , switchA2 , switchA2 , switchA1failtime);
    @Priority(1) Switch switchA2(DCN1):(2, 0, false , switchA1 , switchA3 , switchA1failtime);
    @Priority(1) Switch switchA3(DCN2):(3, 0, true , switchA2 , switchA2 , switchA3failtime);
    @Priority(1) Switch switchB1(DCN1):(4, 1, true , switchB2 , switchB2 , switchB1failtime);
    @Priority(1) Switch switchB2(DCN1):(5, 1, false , switchB1 , switchB3 , switchB1failtime);
    @Priority(1) Switch switchB3(DCN2):(6, 1, true , switchB2 , switchB2 , switchB3failtime);
    @Priority(2) Node DCN1(switchA1, switchB1):(100, 100, 1, 4, node1failtime);
    @Priority(2) Node DCN2(switchA3, switchB3):(101, 100, 3, 6, node2failtime);
}
```

Listing 1: (L1) An abstracted version of the Timed Rebeca model of NRP FD (Full version in Appendix C).

parameters in modeling, we carefully consider values so that the model matches the reality. We will discuss more on timing in the following.

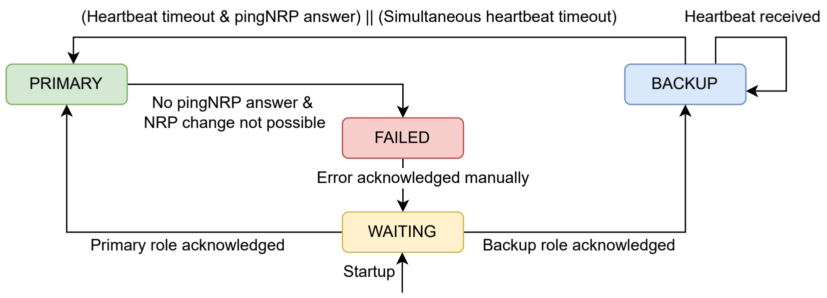

(Heartbeat timeout & pingNRP answer) || (Simultaneous heartbeat timeout)
Heartbeat received
PRIMARY
BACKUP
No pingNRP answer & NRP change not possible
FAILED
Error acknowledged manually
WAITING
Primary role acknowledged
Backup role acknowledged
Startup

Figure 5: Different modes of a DCN in NRP FD in the Rebeca model. *WAITING* is the initial mode. The node transitions from *WAITING* to *PRIMARY* or *BACKUP* based on the value passed to its constructor. From *PRIMARY*, it moves to *FAILED* if after sending a *pingNRP* it receives no response from NRP within the deadline, and it cannot change the NRP either. In the *BACKUP* state, the node transitions to *PRIMARY* if the heartbeat timeouts and *pingNRP* detects a responsive NRP, or when the heartbeat timeout occurs simultaneously for both networks. In the latter case the backup node assumes that the primary node failed, because it is unlikely that there is a failure in both networks. The node stays in *BACKUP* mode as long as it is receiving heartbeats. It remains in *FAILED* until the situation is resolved manually.

In NRP FD, DCNs have four modes, *WAITING*, *BACKUP*, *PRIMARY*, and *FAILED*, as detailed in the diagram in Figure 5. In the Rebeca model, we set the initial mode of DCN to *WAITING* in the constructor of *Node*, L1, line 19. We pass the primary id to both nodes and in the *WAITING* mode the variable denoting the role is set accordingly, and an NRP is announced.

In the *PRIMARY* mode, the primary DCN tests the NRP reachability with *pingNRP*, i.e., sends message *pingNRP* to the NRP which then is served using the message server *pingNRP* (L1, line 55). In a real system, the *pingNRP* could be realized with an Internet Control Message Protocol (ICMP) echo (commonly known as ping) or another suitable protocol depending on the NRP's capabilities. If the NRP fails to respond, the primary announce a new NRP, assuming alternatives are available (using *new_NRP* message server, L1, line 56). After assuring that an NRP exists, the primary DCN sends heartbeats. If there is no available NRP, the primary transition to the *FAILED* mode (*ping_timed_out* in L1, line 25).

In the *BACKUP* mode, the DCN expects heartbeats from the primary. The heartbeat period and tolerance limits (i.e., the number of missed heartbeats before a timeout is declared) must be carefully set to minimize false positives due to transient disturbances. Given that typical DCN redundancy involves two disjoint network paths, a heartbeat is expected on each network path per period. Simultaneous timeouts on all paths likely indicate a primary failure rather than failure of both networks. Thus, NRP FD offers an optimization: transitioning directly to the *PRIMARY* mode upon simultaneous heartbeat timeouts, bypassing the *pingNRP* exchange. However, this optimization slightly increases the risk of dual primaries. This is a bug that model checking catches. The number of maximum missed heartbeats is set to 2 (*max_missed_heartbeat* in L1, line 2). L2, shows the *BACKUP* part of the message server *runMe*. The variables *heartbeats_missed_1* and *heartbeats_missed_2* are counters for heartbeats on the two networks which will increase at each period, and is reset to zero when a heartbeat is received.

The backup DCN counts consecutive *heartbeats_missed* for each network. If both counters exceed the defined limit of *max_missed_heartbeat* (L2, line 4), the backup detects a failure and sends a *pingNRP* to the NRP to verify its reachability. If the NRP is reachable, the DCN transitions from *BACKUP* to the *PRIMARY* state (in *ping_timed_out*, L1, line 25).

In the *FAILED* mode, NRP FD awaits the acknowledgment that manually confirms the resolution of the issues that triggered the transition to *FAILED*.

```
1  case 2: //BACKUP :
2   heartbeats_missed_1++;
3   heartbeats_missed_2++;
4   if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 > max_missed_heartbeats){
5    if(heartbeats_missed_1==heartbeats_missed_2 && heartbeats_missed_2==max_missed_heartbeats+1){
6     mode = PRIMARY;
7     primary=id;
8     ...
9    }else{
10    heartbeats_missed_1 =
       ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
11    heartbeats_missed_2 =
       ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
12    if(NRP_network==0){
13     ping_pending = true;
14     NRP_network=-1;
15     out1.pingNRP(id, NRP_switch_id) after(5);
16     ping_timed_out() after(ping_timeout);
17    }else{ ...  // the other network }
18    NRP_pending = true;
19   }
20  }
21  else if(heartbeats_missed_1 > max_missed_heartbeats || heartbeats_missed_2 >
    ↪  max_missed_heartbeats){...}
```

Listing 2: (L2) The behavior of a DCN in *BACKUP* mode, in the message server *runMe* (full version is provided in Appendix C).

Accuracy of the model. Based on the real situation, we consider the topology and the way the DCNs interact with each other. The rationale for tolerating up to two lost heartbeats ($max\_missed\_heartbeats =$ 2) is based on the low bit error rate of gigabit Ethernet and the ability of a heartbeat message to fit within a standard 1500-byte Ethernet frame. This suggests a low likelihood of losing heartbeat messages, especially across two disjoint networks, thus minimizing the risk of false positives due to regular disturbances. The *heartbeat_period*, combined with *max_missed_heartbeats*, determines the reaction time - the duration from the occurrence of a primary failure to the point at which the backup takes over the primary role. The takeover time requirement varies by domain; for process control, a maximum of 500 milliseconds is tolerable, as suggested by Hegazy et al. [7]. System manuals indicate feasible heartbeat periods are in the tens of milliseconds range[20, 18]. Regarding propagation and *pingNRP* response times, the propagation of a full-sized Ethernet frame on Gigabit Ethernet is about 12 microseconds, negligible compared to the heartbeat period. The NRP's response time is implementation-dependent, potentially under a millisecond. If ICMP ping is employed, a few milliseconds response times are achievable [10]. We've defined the *heartbeat_period* as 1000 time units and set the *ping_timeout* and *nrp_timeout* to 500 time units. We also consider *networkDelay* and *networkDelayForNRPPing* as 1 unit of time. We use the keyword *after* when DCNs ping the NRP node and set it to 5 units of time. These values are chosen to be approximately close to the actual values and preserve the sequence of the messages. Therefore, they may vary, for instance, to a greater or lesser extent. But all timing events should be handled within one period, 1000 time units in our model. We used the *after* construct where we needed to respect the order of execution.

4 Model checking of NRP FD using Afra

We can define our desired properties using assertions in a separate file in Afra and perform model checking. A snapshot of Afra is provided in Appendix B. The main safety property, "*NoDualPrimary*," is shown in L3. This property is set to recognize the dual primary state, i.e., in no state the modes of the two DCNs are both primary. We first define a set of atomic propositions, and then the assertions based on these propositions. Timed Rebeca has a TCTL model checking but it is not integrated in Afra. In many cases, looking at the visualization of the state space helps us see the problems with the algorithm.

```
property {
        define {
                DCN1Primary = (DCN1.mode ==1);
                DCN2Primary = (DCN2.mode ==1);
        }
Assertion{ NoDualPrimary:!(DCN1Primary && DCN2Primary); }}
```

Listing 3: The safety property "*NoDualPrimary*" for NRP FD.

For model checking, we consider the regular system behaviour, and scenarios where we have failures of DCNs and switches. We examine all the possible failure combinations of DCNs and switches at the start of handling an event, and perform model checking to provide a comprehensive analysis. We have modeled failures in three scenarios each of which can have different cases:
1. Failures on each event. In this scenario, we add the following commands at the beginning of each message server for DCNs and switches, simulating the possibility of their failure. Since this scenario models the failure where an event should be handled, we refer to it as event-based. The expression "$?(true, false)$" represents a nondeterministic choice between true and false. When the value true is chosen then a variable is set, this variable is checked in the beginning of the messages servers and if it is set the message server is not executed.

```
//Possible failure for a DCN:
if(?(true,false)) nodeFail();
//Possible failure for a Switch:
if(?(true,false)) switchFail();
```

2. Failures that occur at specific times. We define a set of variables to model the failure of different DCNs and switches at specific times. By manipulating these variables, we can model various combinations of DCN and switch failures at different times across multiple model checking runs.

```
env int switch1failtime = 0; env int switch2failtime = 2500; env int switch3failtime = 0;
env int switch4failtime = 2500;
env int node1failtime = 0; env int node2failtime = 0;
..
//Failure of a DCN at a specific point of time. Value zero means no failure.
if(myFailTime!=0) nodeFail() after(myFailTime);
...
//Failure of a Switch at a specific point of time. Value zero means no failure.
if (myFailTime!=0) switchFail() after(myFailTime);
```

3. Transient failures. These failures could occur, for example, if an attacker deliberately drops the heartbeats for more than the maximum allowed misses (*max_missed_heartbeats*) on both networks.

Subsequently, the backup DCN, upon detecting missed heartbeats, checks the NRP. If the NRP is reachable, it becomes the primary, assuming that the primary has failed, resulting in a dual-primary situation. So we model a transient failure where both heartbeats are missed. Part (not all) of the code for this scenario is the following, which states that only if we do not have an attacker, then the heartbeats will be sent.

```
if(attacker<1){
    out1.heartBeat(0, id) after(networkDelay);
    out2.heartBeat(1, id) after(networkDelay);
}
```

Table 1 illustrates the scenarios we have considered and checked. Number of states and transitions are also reported. Note that in the cases where the the assertion is violated, model checking is stopped after reaching a counter example. Case 1 is the case with no failure. Case 2 is the event-based failure scenario where we investigate all combinations of failures for any DCN or switch, where they stop reacting to the events. Cases 3 to 5 consider failures at time 2500 for DCN1, switchA1 and switchA3, respectively. This number is intended to go through a full round of algorithm execution, with two heartbeats. We have considered case 3 for the *PRIMARY* failure as DCN1 is initially set as the primary DCN. We also consider cases 4, 6, and 7 as failures of switches A1 and B1 can cause the primary DCN to be disconnected from the networks. Case 5 is also considered to model a situation where the backup cannot ping the NRP. Case 8 is modeling the transient error. There are three cases where the model violates the property.

Table 1: Different test scenarios, without any failures, and with different types of failures

Case	Configuration for failures	Result	no. of states and transitions
1	Without failure	✓	38, 49
2	Failures on each event	✗	3539, 4677
3	DCN1 fails at time 2500	✓	113, 138
4	switchA1 fails at time 2500	✓	114, 134
5	switchA3 fails at time 2500	✓	146, 179
6	switchA1 fails at time 2500 and switchB1 at time 3500	✓	187, 223
7	switchA1 and switchB1 fails simultaneously at time 2500	✗	70, 88
8	Heartbeats are missing because of transient errors	✗	35, 42

Afra generates a counter-example in cases of any violation (here for cases 2, 7 and 8 of Table 1). We can explore the states in the counter-example and see the value of the state variables in each of them. A snapshot of the state space showing the dual primary situation for the case 7 is depicted in Figure 8 of Appendix D. These cases may be rare situations in reality, but in formal verification we detect and eliminate the corner cases. To overcome these issues, we provide an extension for NRP FD, which will be described next.

4.1 Leasing NRP FD

To address failure issues, we provide an enhanced NRP FD version called Leasing NRP FD. First, we remove the optimization, i.e., transitioning directly from *BACKUP* to the *PRIMARY* mode upon simultaneous heartbeat timeouts, bypassing the *NRPPing* exchange (L2, lines 5-9).

While NRP FD prioritizes consistency, even without optimization, there remains a non-zero probability of failure. The heartbeat and *pingNRP* messages are separate: the heartbeat indicates whether the primary is alive, and the *pingNRP* informs the backup about its separation from the NRP or the NRP's failure. Since these messages are distinct and can be independently disrupted, it's theoretically possible, as indicated by verification, that a temporary disturbance might disrupt the heartbeats. This disruption could lead the backup to believe the primary has failed, and upon a successful *pingNRP* following the transient disturbance, it might erroneously become the *PRIMARY*, even while the other DCN remains primary. To address this vulnerability, we introduce the Leasing NRP FD, where the primary role is 'leased' from the NRP. This leasing can be implemented in various ways. In our model, the NRP timestamps the latest *pingNRP* from the primary, and then the backup checks this timestamp. Full version of Leasing NRP FD is provided in Appendix C and also on the Rebeca GitHub page[2]. Even with a low probability of dual primary occurrences in the original NRP FD, this inherent algorithmic trait could lead to nondeterministic behavior, which is unacceptable in safety-critical solutions. Thus, there's a need for algorithms like Leasing NRP FD, which eliminate such violations and are more suitable for safety-critical systems. For this new algorithm, Afra created 15891 states, and 34053 transitions, and the assertion is satisfied.

5 Why Timed Rebeca?

In [23], Sirjani argues that when selecting a modeling language, expressiveness is a key factor, but faithfulness to the system being modeled and usability for the modeler are equally crucial. Faithfulness is about how similar the model and the system are. It determines if and how the structures and features supported by the modeling language match with the requirements of the system's domain. Faithfulness makes reusability possible, also in cases gives us better analyzability and traceability. Usability concerns the modeler, and how swiftly the modeler can use the language. These two aspects together are called as friendliness in [23].

Timed Rebeca is a language for modeling asynchronous communication in distributed systems, incorporating a focus on time-related aspects. Regarding faithfulness, actors are units of concurrency like the controllers and switches in our case study. Timed Rebeca is event-driven, taking messages/events from the message/event bags and executing their corresponding message servers. Timed Rebeca is used for modeling and verification in many domains including different network protocols, schedulability in sensor networks and Network on Chip (NoC) [23]. Considering our problem in the domain of distributed control systems, Timed Rebeca provides a natural mapping of structures, features, and flow of control for our purpose such as modeling the topology of the network, behavior of the DCNs and switches based on their roles, the way they communicate using message passing, progress of time required for handing a message, network delay, and periodic events using primitive timing keywords. Message queues/buffer are not explicit and the modeler does not need to manage them. Timing concept is intuitive, and you model the behavior from the perspective of each actor.

Regarding usability, it has a structure like a programming language, hence, it is easy for programmers to use. Debugging can be done based on the counterexamples and going through the model checking process iteratively. Timed Rebeca is supported by an Eclipse IDE called Afra [11]. Afra provides a model checker tool for the family of Rebeca languages. The modeler enters the model and the properties in separate files, then model check and debug the model in Afra. Timed models result in an infinite number of states in the state space due to the progress of time, leading to unbounded transition systems. A shift-equivalence relation is introduced for Timed Rebeca in [12, 13] to ensure a bounded state space. Afra utilizes this relation to generate the state space including local actor states and logical time. Desired properties can be written as assertions in a separate file in Afra. In case of violation, a counter-example is shown visually alongside the model which gives us the ability to traverse and check the values of the actors' variables. As the state space is provided in an XML file, it is also possible to have a visual

[2]https://github.com/rebeca-lang

representation of the entire state space (see an example in Figure 8, App. D). All the above gives us a natural and easy way to model our system, and also provide us analzability and traceability.

6 Related work

Control systems evolve from hierarchical, controller-centric structures toward a flatter, network-centric architecture, enhancing interconnectivity and facilitating communication with cloud services and edge devices [4, 16]. These advancements have been leveraged for fault tolerance—employing backup DCNs in the cloud or orchestrators to recover from DCN failures [7, 9]. To our knowledge, the NRP FD algorithm is the first effort to reduce the CAP theorem's [6] availability tradeoff while preserving consistency in DCN redundancy scenarios [10]. The tradeoff mandated by the CAP theorem is evident in today's redundant DCN systems. Control system user manuals concretize the tradeoff with the different approaches described, which either strive to maintain consistency or prioritize availability upon redundancy link failure [20, 18]. Fault tolerance is ensured using duplicate links, as depicted in Figure 1. With duplicated links, consistency can be prioritized by disabling DCN redundancy if one link fails [20]. However, a dual primary situation arises if both links fail simultaneously. Vice versa, availability is prioritized by not disabling redundancy upon one link failure [18]. The Leasing NRP FD version assures consistency by maintaining a single primary in all failure scenarios.

Appointing a primary is a leader election problem, and various leader election algorithms exist, such as the well-known Bully algorithm [5]. However, the Bully algorithm, and variants thereof, elects multiple leaders in networking partitioning situations, one leader per partition. Alternatively, consensus protocols like Raft and Paxos require a majority [17, 15], ensuring consistency even when partitions occur, as only the majority-containing partition progresses. However, the most common DCN redundancy configurations, typically comprising a primary and a backup, do not allow a majority to form in the event of a partition separating the DCNs [21]. The NRP FD method introduces the NRP that, in combination with a DCN, establishes a majority [10]. The NRP could be as simple as a layer two network switch responding to an ICMP Ping, providing a means to favor consistency over availability. This paper describes the modeling and verification of the NRP FD strategy, along with a novel, lightweight enhancement ensuring a single primary, i.e., guaranteeing that consistency is preserved due to more than one DCN taking the primary role. The algorithm is being extended in different directions, considering different configurations and features. Our aim is to enrich our model align with the extensions of NRP FD, when the extensions are available.

7 Conclusion and Future Work

In this paper we describe the process of modeling and formal verification of NRP FD protocol which is used for preserving consistency in DCN redundancy scenarios using Timed Rebeca and Afra. We investigate different failure scenarios and identify situations where network partitioning can lead to a dual primary. We propose an extension, Leasing NRP FD, which preserves consistency and ensures robustness against different failures. For future research, we focus on the extensibility and flexibility of the proposed protocol including the exploration of a dynamic network topology, multiple backups and multiple primaries. The latter could be a redundancy plan with a single backup for multiple primaries, each with different and unique characteristics such as specific heartbeat time and network delay. Additionally, we aim to incorporate probability considerations rather than just focusing on the possibility (of failures). As another future direction, we plan to investigate the availability trade-off. While NRP-FD prioritizes consistency, this may result in compromising availability. Quantifying this trade-off is a potential direction for further research.

Acknowledgment

We acknowledge the support of the Swedish Knowledge Foundation via the synergy project SACSys (Safe and Secure Adaptive Collaborative Systems) and the Profile DPAC (Dependable Platforms for Autonomous Systems and Control). We also acknowledge the support of the Swedish Foundation for Strategic Research (SSF) via the Serendipity project.

References

[1] *AC 800M High Integrity*. `https://new.abb.com/control-systems/safety-systems/system-800xa-high-integrity/ac-800m-hi-controller`. Accessed: 2024-03-07.

[2] *The DCS of Tomorrow - ABB's Process Automation System Vision Whitepaper*. `https://new.abb.com/control-systems/control-systems/envisioning-the-future-of-process-automation-systems/automation-system-whitepaper`. Accessed: 2024-03-07.

[3] Gul Agha (1986): *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, doi:10.7551/mitpress/1086.001.0001.

[4] Johan Åkerberg, Johan Furunäs Åkesson, Jorgen Gade, Maryam Vahabi, Mats Björkman, Mehrzad Lavassani, Rahul Nandkumar Gore, Thomas Lindh & Xiaolin Jiang (2021): *Future industrial networks in process automation: Goals, challenges, and future directions*. Applied Sciences 11(8), p. 3345, doi:10.3390/app11083345.

[5] H. Garcia-Molina (1982): *Elections in a Distributed Computing System*. IEEE Trans. Comput. 31(1), pp. 48–59, doi:10.1109/TC.1982.1675885.

[6] Seth Gilbert & Nancy Lynch (2002): *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. Acm Sigact News 33(2), pp. 51–59, doi:10.1145/564585.564601.

[7] T. Hegazy & M. Hefeeda (2015): *Industrial Automation as a Cloud Service*. IEEE Trans. Par. and Distr. Syst. 26(10), pp. 2750–2763, doi:10.1109/TPDS.2014.2359894.

[8] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A universal modular actor formalism for artificial intelligence*. In: Proceedings of the 3rd international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., pp. 235–245. Available at `http://ijcai.org/Proceedings/73/Papers/027B.pdf`.

[9] Bjarne Johansson, Mats Rågberger, Thomas Nolte & Alessandro V Papadopoulos (2022): *Kubernetes orchestration of high availability distributed control systems*. In: IEEE Int. Conf. on Ind. Tech. (ICIT), doi:10.1109/ICIT48603.2022.10002757.

[10] Bjarne Johansson, Mats Rågberger, Alessandro Papadopoulos & Thomas Nolte (2023): *Consistency Before Availability: Network Reference Point based Failure Detection for Controller Redundancy*. In: 28th International Conference on Emerging Technologies and Factory Automation, pp. 1–8, doi:10.1109/ETFA54631.2023.10275664.

[11] Ehsan Khamespanah, Marjan Sirjani & Ramtin Khosravi (2023): *Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models*. In Hossein Hojjat & Erika Ábrahám, editors: Fundamentals of Software Engineering, Springer Nature Switzerland, Cham, pp. 72–87, doi:10.1007/978-3-031-42441-0_6.

[12] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi & Mohammad-Javad Izadi (2015): *Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system*. *Science of Computer Programming* 98, pp. 184–204, doi:10.1016/j.scico.2014.07.005.

[13] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan & Ramtin Khosravi (2015): *Floating time transition system: more efficient analysis of timed actors*. In: *Formal Aspects of Component Software*, Springer, pp. 237–255, doi:10.1007/978-3-319-28934-2_13.

[14] Ramtin Khosravi, Ehsan Khamespanah, Fatemeh Ghassemi & Marjan Sirjani (2024): *Actors Upgraded for Variability, Adaptability, and Determinism*. In: *Workshop on State-of-the-Art of Active Objects*, pp. 226–260, doi:10.1007/978-3-031-51060-1_9.

[15] Leslie Lamport (2001): *Paxos Made Simple*. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58. Available at `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[16] Björn Leander, Bjarne Johansson, Tomas Lindström, Olof Holmgren, Thomas Nolte & Alessandro V Papadopoulos (2023): *Dependability and Security Aspects of Network-Centric Control*. In: *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, pp. 1–8, doi:10.1109/ETFA54631.2023.10275344.

[17] Diego Ongaro & John Ousterhout (2014): *In search of an understandable consensus algorithm*. In: *2014 USENIX annual technical conference (USENIX ATC 14)*, pp. 305–319. Available at `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[18] PACSys (2023): *PACSystems™ RX3i Hot Standby CPU Redundancy*. `https://emerson-mas.my.site.com/communities/en_US/Documentation/PACSystems-Hot-Standby-CPU-Redundancy-Users-Manual`. Accessed: 2024-03-07.

[19] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler & Theo Ungerer (2007): *A new Adaptive Accrual Failure Detector for Dependable Distributed Systems*. In: *In ACM Symposium on Applied Computing (SAC 2007*, pp. 551–555, doi:10.1145/1244002.1244129.

[20] Siemens (2024): *Siemens System Manual S7-1500R/H redundant system*. `https://cache.industry.siemens.com/dl/files/833/109754833/att_965668/v3/s71500rh_manual_en-US_en-US.pdf`. Accessed: 2024-03-07.

[21] Andrei Simion & Calin Bira (2023): *A review of redundancy in PLC-based systems*. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI* 12493, pp. 269–276, doi:10.1117/12.2644462.

[22] Marjan Sirjani (2006): *Rebeca: Theory, Applications, and Tools*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem P. de Roever, editors: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, *Lecture Notes in Computer Science* 4709, Springer, pp. 102–126, doi:10.1007/978-3-540-74792-5_5.

[23] Marjan Sirjani (2018): *Power is Overrated, Go for Friendliness! Expressiveness, Faithfulness, and Usability in Modeling: The Actor Experience*. In Marten Lohstroh, Patricia Derler & Marjan Sirjani, editors: *Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10760, Springer, pp. 423–448, doi:10.1007/978-3-319-95246-8_25.

[24] Marjan Sirjani & Ehsan Khamespanah (2016): *On Time Actors*. In Erika Ábrahám, Marcello M. Bonsangue & Einar Broch Johnsen, editors: *Theory and Practice of Formal Methods*, *Lecture Notes in Computer Science* 9660, Springer, pp. 373–392, doi:10.1007/978-3-319-30734-3_25.

[25] Marjan Sirjani, Edward A. Lee & Ehsan Khamespanah (2020): *Verification of Cyberphysical Systems*. *Mathematics* 8(7), doi:10.3390/math8071068.

[26] Marjan Sirjani, Ali Movaghar & MohammadReza Mousavi (2001): *Compositional Verification of an Object-Based Model for Reactive Systems*. In: *AVoCS 2001*. Available at `https://rebeca-lang.org/assets/papers/2001/CompositionalVerificationOfAnObject-BasedModelForReactiveSystems.pdf`.

[27] Jacek Stój (2020): *Cost-effective hot-standby redundancy with synchronization using EtherCAT and real-time ethernet protocols*. IEEE Transactions on Automation Science and Engineering 18(4), pp. 2035–2047, doi:10.1109/TASE.2020.3031128.

A Rebeca Syntax

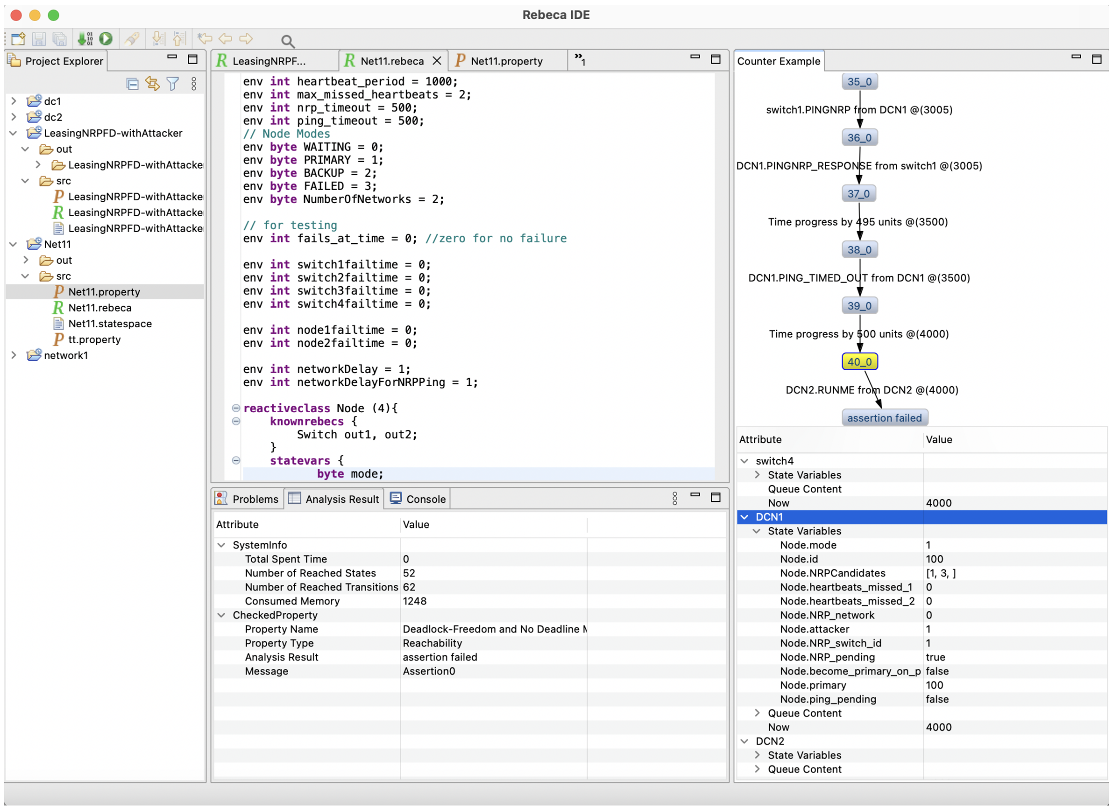
An abstract syntax of Timed Rebeca is provided in Figure 6.

$$\begin{aligned}
Model &::= Class^{*}\ Main \\
Main &::= \mathbf{main}\ \{\ InstanceDcl^{*}\ \} \\
InstanceDcl &::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*}); \\
Class &::= \mathbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars\ MsgSrv^{*}\ \} \\
KnownRebecs &::= \mathbf{knownrebecs}\ \{\ VarDcl^{*}\ \} \\
Vars &::= \mathbf{statevars}\ \{\ VarDcl^{*}\ \} \\
VarDcl &::= type\ \langle v\rangle^{+}; \\
MsgSrv &::= \mathbf{msgsrv}\ methodName(\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
Stmt &::= v = e;\ \mid v = ?(e, \langle e\rangle^{+});\ \mid Call;\ \mid \mathbf{delay}(t);\ \mid if\ (e)\ \{\ Stmt^{*}\ \}[else\ \{\ Stmt^{*}\ \}] \\
Call &::= rebecName.methodName(\langle e\rangle^{*})\ [\mathbf{after}(t)]\ [\mathbf{deadline}(t)]
\end{aligned}$$

Figure 6: An abstract syntax for Timed Rebeca. The identifiers className, rebecName, methodName, literal and type are self-explanatory. The identifier v denotes a variable. The symbol e denotes an expression, which can be either arithmetic, boolean or a non-deterministic choice. Angular brackets $\langle ... \rangle$ serve as meta-parenthesis, with superscript $+$ denoting at least one repetition and superscript $*$ denoting zero or more repetitions. Meanwhile, the use of $\langle ... \rangle$ with repetition indicates a comma-separated list. Square brackets [...] indicate that the enclosed text is optional [23].

B Afra

A snapshot of Afra is provided in Figure 7. The state space statistics are shown in the bottom middle. The generated counterexample is shown in the top right of the panel. At the bottom right, we can see the value of the state variables in each selected state from the counterexample.



```
env int heartbeat_period = 1000;
env int max_missed_heartbeats = 2;
env int nrp_timeout = 500;
env int ping_timeout = 500;
// Node Modes
env byte WAITING = 0;
env byte PRIMARY = 1;
env byte BACKUP = 2;
env byte FAILED = 3;
env byte NumberOfNetworks = 2;

// for testing
env int fails_at_time = 0; //zero for no failure

env int switch1failtime = 0;
env int switch2failtime = 0;
env int switch3failtime = 0;
env int switch4failtime = 0;

env int node1failtime = 0;
env int node2failtime = 0;

env int networkDelay = 1;
env int networkDelayForNRPPing = 1;

reactiveclass Node (4){
    knownrebecs {
        Switch out1, out2;
    }
    statevars {
            byte mode;
```

Attribute	Value
SystemInfo	
Total Spent Time	0
Number of Reached States	52
Number of Reached Transitions	62
Consumed Memory	1248
CheckedProperty	
Property Name	Deadlock-Freedom and No Deadline M
Property Type	Reachability
Analysis Result	assertion failed
Message	Assertion0

```
35_0
  switch1.PINGNRP from DCN1 @(3005)
36_0
  DCN1.PINGNRP_RESPONSE from switch1 @(3005)
37_0
  Time progress by 495 units @(3500)
38_0
  DCN1.PING_TIMED_OUT from DCN1 @(3500)
39_0
  Time progress by 500 units @(4000)
40_0
  DCN2.RUNME from DCN2 @(4000)
assertion failed
```

Attribute	Value
switch4	
State Variables	
Queue Content	
Now	4000
DCN1	
State Variables	
Node.mode	1
Node.id	100
Node.NRPCandidates	[1, 3,]
Node.heartbeats_missed_1	0
Node.heartbeats_missed_2	0
Node.NRP_network	0
Node.attacker	1
Node.NRP_switch_id	1
Node.NRP_pending	true
Node.become_primary_on_p	false
Node.primary	100
Node.ping_pending	false
Queue Content	
Now	4000
DCN2	
State Variables	
Queue Content	

Figure 7: A snapshot of Afra.

C Timed Rebeca model of the Leasing NRP FD

In the following the Timed Rebeca model of the Leasing NRP FD is provided.

```
env int heartbeat_period = 1000;
env int max_missed_heartbeats = 2;
env int ping_timeout =100;
env int nrp_timeout = 100;
// Node Modes
env byte WAITING = 0;
env byte PRIMARY = 1;
env byte BACKUP = 2;
env byte FAILED = 3;
env byte NumberOfNetworks = 2;

env byte MAX_SWITCHES = 99;
// for testing
env int fails_at_time = 0; //zero for no failure

env int switchA1failtime = 0;
env int switchA2failtime = 0;
env int switchA3failtime = 0;
env int switchB1failtime = 0;
env int switchB2failtime = 0;
env int switchB3failtime = 0;

env int node1failtime = 0;
env int node2failtime = 0;

env int networkDelay = 1;
env int networkDelayForNRPPing = 1;

reactiveclass Node (4){
    knownrebecs {
        Switch out1, out2;
    }
    statevars {
        byte mode;
        int id;
        int [2] NRPCandidates;
        int heartbeats_missed_1;
        int heartbeats_missed_2;
        int NRP_network;
        int attacker;
        int which;
        boolean prevWhich;
        int NRP_switch_id;
        boolean NRP_pending;
        boolean become_primary_on_ping_response;
        int primary;
        boolean ping_pending;
        boolean init;
    }
    Node (int Myid, int Myprimary, int NRPCan1_id, int NRPCan2_id, int myFailTime) {
        id = Myid;
        attacker = 0;
        which=0;
        prevWhich=true;
        NRPCandidates[0] =NRPCan1_id;
        NRPCandidates[1] =NRPCan2_id;
        heartbeats_missed_1 = 0;
        heartbeats_missed_2 = 0;
        NRP_network = -1;
        NRP_switch_id = -1;
        NRP_pending = true;
        become_primary_on_ping_response = false;
        primary = Myprimary;
        ping_pending = false;
        init=true;

        mode = WAITING;
        if(myFailTime!=0) nodeFail() after(myFailTime);
        runMe();
    }
    msgsrv new_NRP_request_timed_out() {
        // if(?(true,false)) nodeFail();
        if (mode == BACKUP) {
            if (NRP_pending) {
                NRP_pending = false;
                if (become_primary_on_ping_response)
                    become_primary_on_ping_response = false;
            }
        }
    }
    // logical action ping_timed_out(ping_timeout)
    msgsrv ping_timed_out() {
        // if(?(true,false)) nodeFail();
        if (mode == BACKUP) {
            if (ping_pending) ping_pending = false;
            else{
                if(which>1){
                    mode = PRIMARY;
                    heartbeats_missed_1 = 0;
                    heartbeats_missed_2 = 0;
                    primary=id;
                    if(NRP_network==0) out1.new_NRPBack(id, id,NRP_network, NRP_switch_id);
                    else out2.new_NRPBack(id,id, NRP_network, NRP_switch_id);
                    mode = PRIMARY;
                    heartbeats_missed_1 = 0;
                    heartbeats_missed_2 = 0;
                    primary=id;
                    NRP_pending = true;
                }else NRP_pending = true;
            }
        }else if (mode == PRIMARY){
            if (ping_pending){
                NRP_network++;
                if(NRP_network<NumberOfNetworks){
                    NRP_switch_id = NRPCandidates[NRP_network];
                    if(NRP_network==0) out1.new_NRP(id, id,NRP_network, NRP_switch_id);
                    else out2.new_NRP(id,id, NRP_network, NRP_switch_id);
                } else {
                    NRP_network=NumberOfNetworks;
                    mode=  WAITING;
                }
                NRP_pending = true;
            } else{
                if(attacker<1){
                    out1.heartBeat(0, id) after(networkDelay);
                    out2.heartBeat(1, id) after(networkDelay);
                }
            }
        }
    }
    msgsrv pingNRP_response(int mid, boolean w, boolean pw){
        // if(?(true,false)) nodeFail();
        if (mode==WAITING);
        else if (mode == BACKUP){
            if(!w && !pw) which++;
            else which=0;
            if(which>1)
            ping_pending = false;
        }
        else if (mode == PRIMARY)
            ping_pending = false;
        else if (mode==FAILED);
    }
    msgsrv new_NRP(int mid,int prim, int mNRP_network, int mNRP_switch_id) {
        // if(?(true,false)) nodeFail();
        if(mode!= FAILED){
            NRP_network = mNRP_network;
            NRP_switch_id = mNRP_switch_id;
        }
    }
    msgsrv new_NRPBack(int mid,int prim, int mNRP_network, int mNRP_switch_id) {
        // if(?(true,false)) nodeFail();
        if(mode!= FAILED){
            NRP_network = mNRP_network;
            NRP_switch_id = mNRP_switch_id;
        }
    }
    msgsrv runMe(){
      switch(mode){
        case 0: //WAITING :
            if(init){
                if (id == primary){
                    mode = PRIMARY;
                    NRP_network++;
                    if(NRP_network<NumberOfNetworks){
                        NRP_switch_id = NRPCandidates[NRP_network];
                        if(NRP_network==0)out1.new_NRP(id,id, NRP_network, NRP_switch_id);
                        else out2.new_NRP(id,id, NRP_network, NRP_switch_id);
                    } else NRP_network=NumberOfNetworks;
                } else mode =BACKUP;
                init=false;
            }
            break;
        case 1: //PRIMARY :
            attacker++;
            if(attacker>1) attacker=1;
            if(NRP_network==0){
                ping_pending = true;
                out1.pingNRP(id,id, NRP_switch_id) after(5);
                ping_timed_out() after(ping_timeout);
            }else{
                ping_pending = true;
                out2.pingNRP(id,id, NRP_switch_id) after(5);
                ping_timed_out() after(ping_timeout);
            }
            NRP_pending = true;
            break;
        case 2: //BACKUP :
            heartbeats_missed_1++;
            heartbeats_missed_2++;
            if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 >
            ↪  max_missed_heartbeats){
               heartbeats_missed_1 =
               ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
               heartbeats_missed_2 =
               ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
                // if(heartbeats_missed_1==heartbeats_missed_2 &&
                ↪  heartbeats_missed_2==max_missed_heartbeats+1){
                    // mode = PRIMARY;
                    // heartbeats_missed_1 = 0; // Prevent detecting again immediately.
                    // heartbeats_missed_2 = 0;
                    // primary=id;
                    // NRP_pending = true;
                // }else{
                if(NRP_network==0){
                    ping_pending = true;
                    //NRP_network=-1;
                    out1.pingNRP(id,id, NRP_switch_id) after(15);
                    ping_timed_out() after(ping_timeout);
                }else{
                    ping_pending = true;
                    //NRP_network=-1;
                    out2.pingNRP(id,id, NRP_switch_id) after(15);
                    ping_timed_out() after(ping_timeout);
                }
                NRP_pending = true;
                // }
            }else if(heartbeats_missed_1 > max_missed_heartbeats|| heartbeats_missed_2 >
            ↪  max_missed_heartbeats){
                if(NRP_network==0 && heartbeats_missed_1 > max_missed_heartbeats) {
                    ping_pending = true;
                    out1.pingNRP(id,id, NRP_switch_id) after(5);
                    ping_timed_out() after(ping_timeout);
                }else if(NRP_network==1 && heartbeats_missed_2 > max_missed_heartbeats){
                    ping_pending = true;
                    out2.pingNRP(id,id, NRP_switch_id) after(5);
                    ping_timed_out() after(ping_timeout);
                }
               heartbeats_missed_1 =
               ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
               heartbeats_missed_2 =
               ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
            }
            break;
        case 3: //FAILED :
            break;
      }
      self.runMe() after(heartbeat_period);
    }
    msgsrv heartBeat(byte networkId, int senderid) {
        // if(?(true,false)) nodeFail();
        if (mode==BACKUP){
            if (networkId == 0) heartbeats_missed_1 = 0;
            else heartbeats_missed_2 = 0;
        }
    }
    msgsrv nodeFail(){
        primary=-1;
        mode = FAILED;
        NRP_network=-1;
        NRP_switch_id=-1;
        heartbeats_missed_1 = 0;
        heartbeats_missed_2 = 0;
        NRP_pending = true;
        become_primary_on_ping_response = false;
        ping_pending = false;
     }
}
reactiveclass Switch(10){
    knownrebecs {
       Node nodeTarget1;
    }
    statevars {
        byte mynetworkId;
        int id;
        boolean which;
        boolean prevWhich;
        boolean failed;
        boolean amINRP;
        boolean primaryPinged;
        boolean terminal;
        Switch switchTarget1;
        Switch switchTarget2;
        int primary;
    }
    Switch (int myid, byte networkId, boolean endSwitch , Switch sw1, Switch sw2, int myFailTime) {
        mynetworkId = networkId;
        primary=0;
        id = myid;
        primaryPinged=false;
        terminal=endSwitch;
        amINRP = false;
        failed = false;
        switchTarget1 = sw1;
        switchTarget2 = sw2;
        which=true;
        if (myFailTime!=0) switchFail() after(myFailTime);
    }
    msgsrv switchFail(){
        failed = true;
        amINRP=false;
    }
    msgsrv pingNRP_response(int senderNode,boolean w,boolean pw){
        // if(?(true,false)) switchFail();
        if(!failed)
            if(terminal && senderNode <= MAX_SWITCHES) nodeTarget1.pingNRP_response(id, w,pw); //Pass back
            else if(senderNode >id) switchTarget1.pingNRP_response(id, w,pw);
        else switchTarget2.pingNRP_response(id, w,pw);
    }
    msgsrv pingNRP(int switchNode, int senderNode, int NRP) {
        // if(?(true,false)) switchFail();
        if(!failed)
            if(terminal && NRP==id){
                prevWhich = which;
                which= (senderNode==primary);
                if(switchNode <= MAX_SWITCHES) switchTarget1.pingNRP_response(id,which, prevWhich);
                ↪  //Response
                else nodeTarget1.pingNRP_response(id,which, prevWhich);
            }else if(switchNode >id) switchTarget1.pingNRP(id,senderNode,NRP);
        else switchTarget2.pingNRP(id,senderNode, NRP);
    }
    msgsrv new_NRP(int senderNode,int prim, int mNRP_network, int mNRP_switch_id) {
        // if(?(true,false)) switchFail();
        if(!failed){
            if(id==mNRP_switch_id) {
                amINRP=true;
                primary=prim;
            } else amINRP=false;
            if(terminal && senderNode <= MAX_SWITCHES)nodeTarget1.new_NRP(id,prim, mNRP_network,
            ↪  mNRP_switch_id);
            else if(senderNode >id) switchTarget1.new_NRP(id,prim, mNRP_network, mNRP_switch_id); //Pass
            ↪  back
            else switchTarget2.new_NRP(id,prim, mNRP_network, mNRP_switch_id);
        }
    }
    msgsrv new_NRPBack(int senderNode,int prim, int mNRP_network, int mNRP_switch_id) {
        // if(?(true,false)) switchFail();
        if(!failed){
            if(id==mNRP_switch_id) {
                amINRP=true;
                primary=prim;
            } else amINRP=false;
            if(terminal && senderNode <= MAX_SWITCHES)nodeTarget1.new_NRPBack(id,prim, mNRP_network,
            ↪  mNRP_switch_id);
            else if(senderNode >id) switchTarget1.new_NRPBack(id,prim, mNRP_network, mNRP_switch_id);
            ↪  //Pass back
                else switchTarget2.new_NRPBack(id,prim, mNRP_network, mNRP_switch_id);
        }
    }
    msgsrv heartBeat(byte networkId, int senderNode) {
        // if(?(true,false)) switchFail();
        if(!failed)
            if(terminal && senderNode <= MAX_SWITCHES) nodeTarget1.heartBeat(networkId,id)
            ↪  after(networkDelay);
            else if(senderNode > id) switchTarget1.heartBeat(networkId,id) after(networkDelay);
                else switchTarget2.heartBeat(networkId,id) after(networkDelay);
    }
}

main {
    @Priority(1) Switch switchA1(DCN1):(1, 0, true , switchA2 , switchA2 , switchA1failtime);
    @Priority(1) Switch switchA2(DCN1):(2 ,0, false , switchA1 , switchA3 , switchA1failtime);
    @Priority(1) Switch switchA3(DCN2):(3, 0, true , switchA2 , switchA2 , switchA3failtime);
    @Priority(1) Switch switchB1(DCN1):(4, 1, true , switchB2 , switchB2 , switchB1failtime);
    @Priority(1) Switch switchB2(DCN1):(5, 1, false , switchB1 , switchB3 , switchB1failtime);
    @Priority(1) Switch switchB3(DCN2):(6, 1, true , switchB2 , switchB2 , switchB3failtime);

    @Priority(2) Node DCN1(switchA1, switchB1):(100, 100, 1, 4, node1failtime);
    @Priority(2) Node DCN2(switchA3, switchB3):(101, 100, 3, 6, node2failtime);
}
```

D State Space

The state space of Timed Rebeca model for the NRP FD (including the problematic optimization) implementing case 7 of Table 1 has 70 states and 88 transitions. Case 7 is where switchA1 and switchB1 fail simultaneously at time 2500. A portion of the visualized state space is provided in Figure 8. We define the followings in the the property file (see L3):

```
DCN1Primary = (DCN1.mode ==1);
DCN2Primary = (DCN2.mode ==1);
DCN2Backup = (DCN2.mode ==2);
switchA1Failed = (switchA1.failed);
switchB1Failed = (switchB1.failed);
switchA1NRP = (DCN1.NRP_switch_id==1 && DCN2.NRP_switch_id==1);
...
```

The term $DCN1Primary$ means that the mode of DCN1 is PRIMARY (similar for DCN2) and the term $switchA1Failed$ means that the state variable $failed$ of switcheA1 is $true$ (similar for switcheB1). $switchA1NRP$ means that the state variable *NRP_switch_id* equals 1 (the id of switchA1) for both DCNs. In case 7 of Table 1, both switches fail at time 2500. As we are at the time 3000 in S59, switchA1Failed and switchB1Failed are true at the states depicted. Both DCN1 and DCN2 execute a runMe in each heartbeat period:

```
heartbeat_period = 1000 // line 1 of Listing 1
...
self.runMe() after(heartbeat_period) // line 35 of Listing 1
..
```

In each period, PRIMARY (DCN1) checks its NRP availability. In the state S63, DCN1 sends a PINGNRP message to switchA1 in the new heartbeat period, @3000. By receiving PINGNRP, switchA1 which is failed, does nothing (line 296 of Appendix C). In the state S65, by running Ping_timed_out, DCN1 will notice that switchA1 has failed. DCN1 tries to select a new NRP from its NRP candidate set (here switchB1 which is not operational at the moment). Note that there is no active NRP in S66. At the next runMe, @4000, DCN2 changes its mode to PRIMARY due to missing more than maximum heartbeats allowed on both networks simultaneously. We can see in S70 a dual primary situation occurred. We commented out the assertion such that the model checker continues creating the state space.

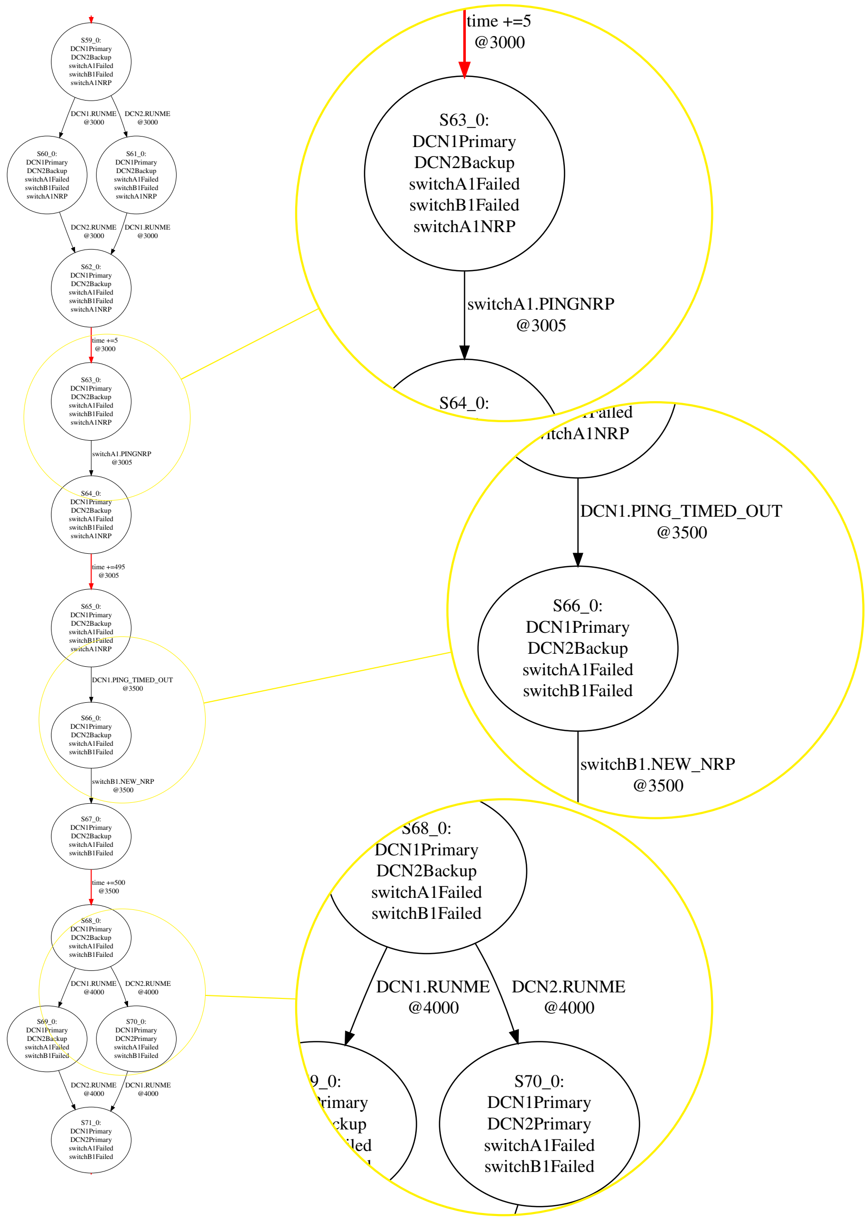


Figure 8: A part of the visualized state space for the Timed Rebeca model of the NRP FD.

Sliced Online Model Checking for Optimizing the Beam Scheduling Problem in Robotic Radiation Therapy

Lars Beckers[1] Stefan Gerlach[2] Ole Lübke[1]

Alexander Schlaefer[2] Sibylle Schupp[1]

{lars.beckers, stefan.gerlach, ole.luebke, schlaefer, schupp}@tuhh.de

[1]Institute for Software Systems [2]Institute of Medical Technology and Intelligent Systems
Hamburg University of Technology, Hamburg, Germany*

In robotic radiation therapy, high-energy photon beams from different directions are directed at a target within the patient. Target motion can be tracked by robotic ultrasound and then compensated by synchronous beam motion. However, moving the beams may result in beams passing through the ultrasound transducer or the robot carrying it. While this can be avoided by pausing the beam delivery, the treatment time would increase. Typically, the beams are delivered in an order which minimizes the robot motion and thereby the overall treatment time. However, this order can be changed, i.e., instead of pausing beams, other feasible beam could be delivered.

We address this problem of dynamically ordering the beams by applying a model checking paradigm to select feasible beams. Since breathing patterns are complex and change rapidly, any offline model would be too imprecise. Thus, model checking must be conducted online, predicting the patient's current breathing pattern for a short amount of time and checking which beams can be delivered safely. Monitoring the treatment delivery online provides the option to reschedule beams dynamically in order to avoid pausing and hence to reduce treatment time.

While human breathing patterns are complex and may change rapidly, we need a model which can be verified quickly and use approximation by a superposition of sine curves. Further, we simplify the 3D breathing motion into separate 1D models. We compensate the simplification by adding noise inside the model itself. In turn, we synchronize between the multiple models representing the different spatial directions, the treatment simulation, and corresponding verification queries.

Our preliminary results show a 16.02 % to 37.21 % mean improvement on the idle time compared to a static beam schedule, depending on an additional safety margin. Note that an additional safety margin around the ultrasound robot can decrease idle times but also compromises plan quality by limiting the range of available beam directions. In contrast, the approach using online model checking maintains the plan quality. Further, we compare to a naive machine learning approach that does not achieve its goals while being harder to reason about.

1 Introduction

Radiation therapy presents a widely used option for cancer treatment. Typically, beams from a range of different directions are used to deliver a therapeutically effective dose to the tumor while maintaining a tolerable dose for all other tissues. The latter is particularly important for critical structures where radiation damage would result in severe side effects. While different treatment systems have been proposed, one interesting approach is robotic radiosurgery where a robotic arm carries the beam source, allowing for a very large solid angle of possible beam directions. The optimal choice of beams results from careful treatment planning[14], which accounts for the beams' attenuation when passing through the patient.

*This study was partially funded by DFG SCHU 2479, and DFG SCHL 1844/6-1.

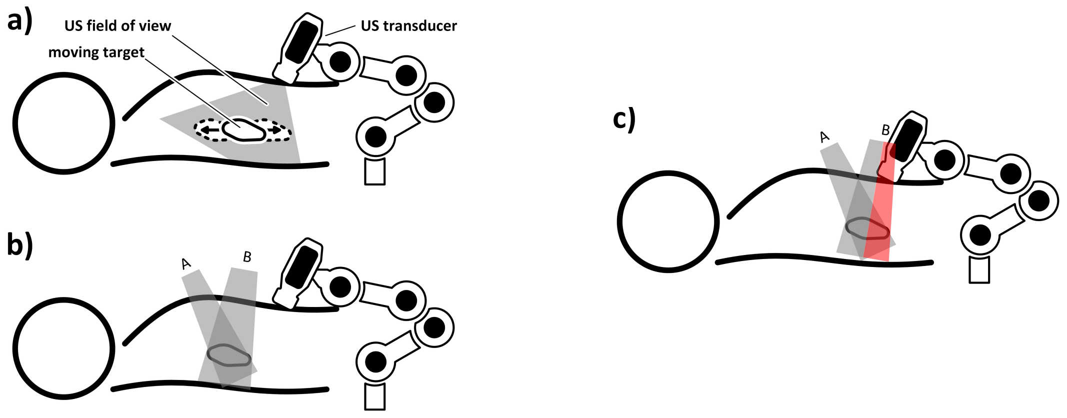


Figure 1: Illustration of the situation we consider. Figure a) shows the the general setup with the patient and the robot mounted ultrasound (US) transducer, as well as the target which moves along the patient's superior-inferior axis. Figure b) depicts a position of the target where two beams, A and B, are both feasible and can be delivered. Figure c) shows another target position for which beam B would be partially blocked and therefore infeasible. Note, that the range of motion of the target may vary and for some intervals during the overall treatment, beam B may also be feasible throughout the full breathing cycle.

One of the particular advantages of robotic beam delivery is the ability to adjust the position and orientation of the beam source quickly. This is desirable, as some tumors are subject to substantial motion, especially due to respiration and in regions close to the diaphragm. Different respiratory motion prediction and control strategies have been studied to realize real-time motion compensated treatments, which are now routinely used with the robotic *CyberKnife* (Accuray, USA)[5]. However, a key aspect of motion compensation is the detection of the tumor's actual motion. Typically, this is achieved by a combination of X-ray imaging and external marker tracking. However, recently the use of robotic ultrasound has been considered, which would allow fast, continuous and non-ionizing imaging of the tumor's motion[8, 6, 7]. This advantage comes at the cost of integrating the ultrasound transducer and the robot carrying it into the overall system setup. Particularly, safe beam delivery must be guaranteed at all times, i.e., as Figure 1 shows, the treatment beams must not pass through ultrasound transducer and robot, as this would compromise the dose estimate.

At planning time, the overall collision free delivery of all beams can be optimized, i.e., a generalized traveling salesman problem (GTSP) is solved to determine the best sequence of beams and to obtain the coordinated motion of both robots[15]. However, during treatment the motion trajectories of the tumor vary, e.g., as patients may change from chest to abdominal breathing. A straightforward approach to maintain the planned dose is to pause the treatment beam whenever the motion causes the current beam to collide with the ultrasound transducer or robot. Given that this prolongs the overall treatment time and that other beams may be feasible, it would be desirable to predict the feasibility of collision free beam delivery given the current tumor motion pattern and to select beams which can be safely delivered without interruption.

In this setting, we propose studying model checking as a means to realize safe and effective treatments. Instead of using a fixed order on the beams, we consider modeling the current respiratory motion and to verify for each beam whether it is feasible, i.e., can be delivered. While model checking verifies properties—regarding safety, in particular—of modeled systems, in case of a breathing patient it is impossible to provide a full model of all possible future behavior. Thus, if we want to verify if a beam is feasible during a treatment session, we need to reduce the scope of this verification, both in terms of detail represented by the model and in terms of the covered time span, down to the next few seconds. Still, we need to regularly verify whether beams are feasible based on current motion data to make up for inaccurancies and to cover the whole treatment session. This approach of repeated verification with current data is called *Online Model Checking*.

While the basic breathing motion is similar among patients, the specifics of their situation are different e.g., whether their inhalation is short or deep, fast or slow, how much it varies over time, when exactly, how grave the differences between those patterns are. Overall, a model will necessarily be inaccurate and the online checking approach is used to limit this effect. The predicted patient's motion—as the foundation to strategically select beams—needs to be formally verified to ensure safety, whether it stays within the safe region. Moreover, at any time it would be important to consider a number of beams in order to determine some feasible beam to be delivered next, because verification of a single beam may not be sufficient to get an optimal or even viable beam. At the same time it is possible that multiple beams satisfy the safety requirements, requiring a decision as part of an optimization problem.

We want to satisfy above requirements within a software system that improves the overall treatment delivery. For this software, its timeliness is important, since the online verification is necessarily limited in its future scope and we need to offer new beams in a timely manner. Instead of terminating when the time allotted for the session is insufficient, it will use the available time more efficiently and thus allow for more completed treatments. Based on these requirements and existing tools, our work resulted in the following contributions:

- An approach for a verified schedule of beams in radiation therapy. This approach is based on an existing online model checking environment for respiratory motion. We contribute the statistical verification of dynamically selected beam candidates in real time.
- A representation of 3-dimensional respiratory motion as product of three 1-dimensional model slices to limit the model's complexity. This requires synchronization of verfication queries and results within the real time environment.
- An implementation in Rust, using the UPPAAL model checker for verification. Further, the implementation of experiments including data generation for evaluation of supporting our approach with machine learning methods.

The result is a dynamic beam scheduling which selects feasible beams and thereby reduces the treatment time. Upcoming beams are selected according to our online model of patient motion data, while the timing requirements of the approach are honored. We can keep the timing requirements by the use of a simple, 1D model that allows us to keep verification times short. In turn, we necessarily increase the complexity of our implementation that needs to synchronize the 3 parallel online models and its beam verification tasks. We propose an architecture that combines three parallel, model-generating processes, the verification processes for beams as required, and the actual client that simulates the therapy. We reduce idle time where no beam can be delivered due to collisions by 16.02 % to 37.21 % depending on the amount of additional safety margin around the ultrasound robot.

Furthermore, we discuss an approach to improve the beam selection by using AI models to make predictions on the breathing patterns and beam verification times. This would allow us to make better decisions in our selection and verification. We tried to train a multi-layer perceptron regressor on beam verification duration, as well as a classifier on 1D breathing movement data to discern breathing pattern irregularities into different categories. However, this was not successful to derive meaningful strategies from: The regressor predicted verification durations on the lower end of their range and the classifier did not learn the intended categories, where more difficult patterns are characterized by noisier or rapidly changing movements. Here, an algorithm detecting extreme values may be better suited.

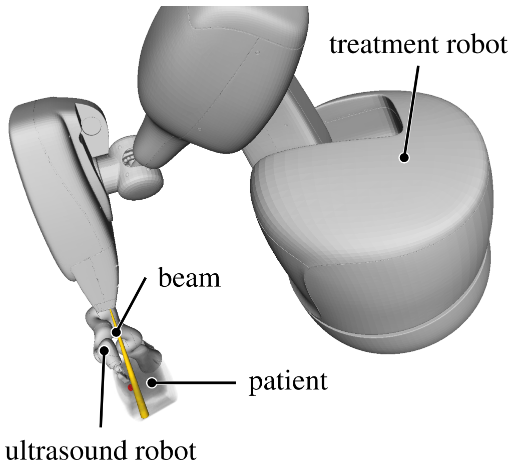


Figure 2: Two interacting robots: One robot is used for treatment delivery and the other robot is used for ultrasound image guidance.

```
ID,Time[ms],Threshold[mm]
80731,24281,5.0
76503,11222,5.0
75681,13682,7.5
74528,23749,10.0
67108,2243,10.0
79427,7133,12.5
70571,16927,15.0
77460,4354,15.0
70211,16240,15.0
68851,1488,17.5
59592,7335,17.5
74430,14448,17.5
69894,29738,20.0
77674,1609,20.0
78301,16381,22.5
81561,3116,25.0
61025,17047,27.5
71430,1758,31.0
81038,21519,31.0
```

Figure 3: Excerpt from a 1D beam list.

In the following section, we provide details of our beam radiation therapy case study. In section 3, we present the necessary background to our approach. This is followed by sections 4 and 5, in which we discuss our models and data, respectively. In section 6, we elaborate on the architecture and implementation of our contribution. The experimental results are covered in section 7. In section 8, we discuss possible machine learning enhancements to our approach. Finally, we close with a summary in section 9.

2 Ultrasound Guided Motion Compensated Radiation Therapy

In radiation therapy, multiple beams are directed at a target region within the human body, which is typically referred to as a *planning target volume* (PTV). The cumulative dose delivered by the beams is optimized to reach a therapeutically effective level inside the PTV, while being as low as possible outside and particularly in so-called *organs at risk* (OAR). During plan optimization the beams are weighted to realize a clinically acceptable trade-of between these objectives. For our sample scenario we consider robotic beam delivery with the CyberKnife and robotic ultrasound based on the lightweight and redundant *LBR iiwa med robot* (KUKA, Germany) as illustrated in Figure 2.

During a treatment session, a patient inhales and exhales and may exhibit varying breathing patterns, e.g., slower and faster breathing, deep and shallow breathing, abdominal and chest breathing, and a number of potentially sudden changes, e.g., due to coughing. When considering motion compensation, the movement of the PTV is off-set by an equivalent motion of the beams, i.e., the beam carrying robot moves the beams synchronously. Hence, the beams move relative to all static objects, including the ultrasound transducer and the robot carrying it. However, beams need to be switched off whenever they would pass through these objects, as the effective dose would be compromised by the additional attenuation and scattering.

While an effective treatment requires all beams to be delivered, there is typically no requirement to observe a particular order. In fact, the order is often optimized such that the time to visit all beam starting positions is minimized, as the motion time adds to overall treatment time. In case of ultrasound guidance the coordinated motion of both robots can be approached as a GTSP and a time-optimal beam schedule can be computed[15]. However, as the breathing patterns are not known a priori, it may occur that some beams cannot be delivered without interruption. At the same time, other beams may be feasible. Any beam that satisfies the safety requirements and does not collide with ultrasound transducer or robot is a viable alternative beam to be delivered. Thus, we consider beam scheduling as the optimization problem to minimize the time beams are inactive due to collisions.

Consider, for example, a patient exhibiting an increasingly deeper breathing motion during the treatment session. We can compute the minimum distance between beam and ultrasound transducer and robot at any time in the breathing cycle, which we refer to as threshold. Now, starting the session with beams with large thresholds and unsuccessfully trying to apply beams with small thresholds later will potentially be a worse solution than a randomized ordering. Furthermore, the robot requires a substantial amount of time to change its configuration between beams, particularly when the next beam is far away from the previous beam.

Therefore, our proposal is a dynamic list of beams that is checked for possible beams within the next time slot. While it is preferable to continue along the current time-optimal list, especially finishing a currently running beam, we can substitute waiting times where no beam is applied with other beams that can be delivered. Afterwards, the new time-optimal list of beams can be computed and treatment continued. Thus, safety requirements are upheld while the treatment duration can be reduced.

3 Online Model Checking

In online model checking (OMC), properties of a modeled system are verified regularly in an interval[17]. The system processes exhibit uncertainties. Thus, statistical model checking is typically used, which assesses the probability of its queried property over multiple, simulated runs of its system until it has established the necessary confidence. This behaviour can be customized through the configuration of the model checker, but we do not employ this in this paper. Each verification is limited in its temporal scope to reduce the required verification time to fit within the interval. Thus, the model is only valid for a short amount of time before the drift from reality gets too large. Still, the temporal scope needs to be large enough to ensure that in each verification step there is enough time for an emergency stop if verification fails and the sequence of verification intervals overlaps to cover the progressing system entirely. OMC has already been used in several different applications[16][11][13], including cyber-physical systems and medical settings[3][9]. However, it had not been used for beam verification.

For proper verification, a model is regularly modified with new data. In reality, this data may be supplied by sensors recording live events. In experimental settings, it can be either recorded data or entirely synthetic. The most common modification is to update and reassign model variables that represent the current state, e.g., the patient's position. Thus, for a continuing online setting, we receive a line of models synthesized from the template, each representing one time step in the verification process.

For our setting, we make use of an existing online model checking software that predicts the respiratory motion of a patient in its primary movement dimension[12]. It acts as a frontend to the underlying statistical model checker, UPPAAL-SMC[2][4]: It receives data updates to provide these to the model, implements reading in model templates and writing out the modified instances every three seconds for a validity window of six seconds. As it has been used to demonstrate respiratory motion[1], the template model and required data sampling and transformation have also already been implemented. Thus, adaptions to our problem have remained minimal. The OMC software also verifies that its modified model is an adequate representation according to its input data. It does so by advancing time by one second within the interval, and assessing the probability that the observed patient position is within a bounding box of the expected position. For completeness, we provide the queries and corresponding evaluation model in Appendix B, but regard this in the following as a black box step, as only its verdict is relevant to us.

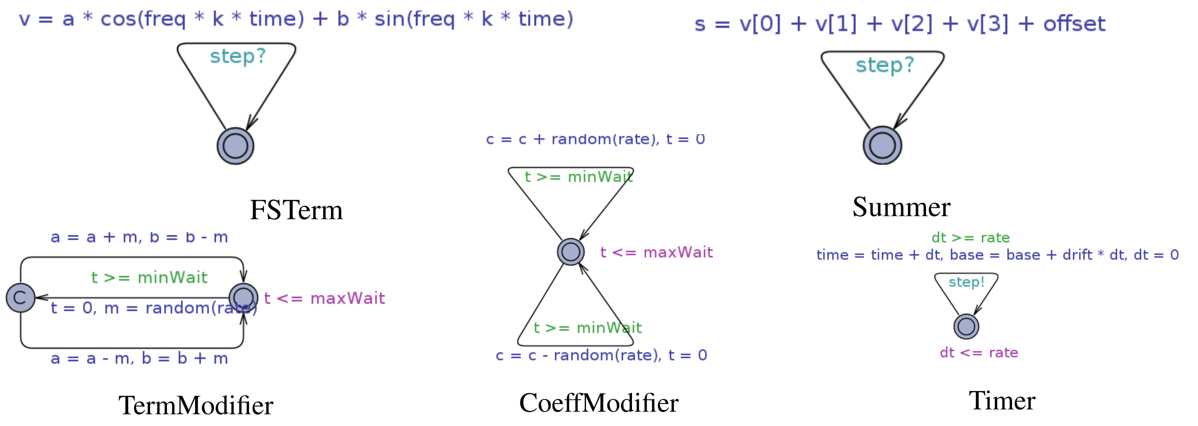


Figure 4: The complete 1D model: UPPAAL automata network template representing the respiratory motion.[12] Its declarations are given in Appendix A. The continually updated variables are also shown in Figure 5. Note the time stepping through UPPAAL's synchronization messages.

4 Modeling

In the following section, we will provide details on the respiratory motion model and how we use the generated motion models for our beam scheduling problem, so that our new software can assess the deliverability of potential beams w.r.t. the motion. We approximate respiratory motion as a superposition of sine curves in space, in each of the three dimensions. We present two modeling designs that simplify the representation of respiratory motion and explain how oversimplification is prevented. Furthermore, we describe the verification of the beams.

4.1 1D motion modeling

The first simplification, introduced in [1], concerns the reduction to a 1D motion representation through modification of the motion model shown in Figure 4. The represented axis is orthogonal to the patient's body and is the primary movement direction, while the other two dimensions behave similarly but with less extreme motion. The idea is that the orthogonal axis is the most important one. In each step, the OMC software samples new positions, which would be measured by the treatment controller in reality, as it progresses in time, and updates this base model accordingly.

The model is a simplified breathing motion representation based on the following formula:

$$x(t) = d \cdot t + \sum_{k=1}^{4} c_k \cdot cos(k \cdot f \cdot t) + s_k \cdot sin(k \cdot f \cdot t), \text{ with } t \text{ time, } d \text{ linear drift, } f = \frac{2\pi}{period}.$$

The position samples are Fourier-transformed to obtain the decomposed s and c terms of a sine and a cosine wave over time at the samples' frequency. Four terms are summed and must later be offset by a base value, which the wave moves around. In Figure 4, we already see all necessary components to describe this as an UPPAAL model. The *Summer* automaton just computes a sum s when a new step is due. This sum is assigned to the model's global *result* variable. The sum's first four components are computed by four instances of *FSTerm*, which computes the four combined sine-cosine waves of the formula, where a and b are the transformed samples, k is the ordinal number of the term, *freq* the sample's frequency, and *time* is a stepped view into the clock. Those summed together with the base position, parameterised as *offset* here, is enough to compute the patient's position. The *Timer* automaton is used to drive this computation in steps at a regular interval of 38 milliseconds, parameterised as *dt*, and moves the base according to its expected *drift*.

```
const double period = 5088.0;
const double drift = -0.0;

double base = -3.6508;
double a[4] = { -0.608, 0.205,
    0.0744, -0.0764 };
double b[4] = { 2.5745, -0.414,
    -0.0149, 0.0096 };
```

X axis

```
const double period = 5088.0;
const double drift = 0.0;

double base = 1.698;
double a[4] = { 0.2631, -0.0887,
     -0.0322, 0.0331 };
double b[4] = { -1.1144, 0.1792,
     0.0065, -0.0041 };
```

Y axis

```
const double period = 5088.0;
const double drift = 0.0;

double base = 1.8164;
double a[4] = { 0.0757, -0.0255,
     -0.0093, 0.0095 };
double b[4] = { -0.3202, 0.0516,
     0.0019, -0.0012 };
```

Z axis

Figure 5: Declarations of the models of a single timestep of patient DB126-Fx1. As the position and movement is different, the beam verification queries require different thresholds per dimension.

However in Figure 4, we also see two additional automata. In order to compensate that the real motion does not adhere to a (simplified) mathematical model based on an exact formula, the model includes an *accuracy* parameter that determines how much randomness is applied to the coefficients and terms. Thus, when setting the accuracy value below 100%, a range of possible motion curves is considered starting from the latest current data point. This is important, because not only the model itself is simplifying, but also the validity period of at most six seconds is already quite long considering that a patient may suddenly start coughing. The accuracy value is set at the start of a session, thus a line of models created for a single patient will have a fixed value. The *TermModifier* automaton is instantiated for all four indices into the *a* and *b* arrays. The values are modified at random, based on a *rate* derived from the accuracy parameter. Instead of regular time steps, there is simply a minimum and maximum wait time between modifications. Similarly, the *CoeffModifier* is instantiated twice to modify both the *base* and *frequency*, represented as *c* here.

Besides the automaton network, the model declares the variables describing the actual movement. The continually updated, declared variables of such a model are shown in Figure 5. The *period* value describes the period of the regular breathing sine curve. The *base* value is the origin of this period. The four cosine and sine terms are composed as arrays *a* and *b*. As mentioned, there is a (small) *drift* value added to the base for every time step. Among other intermediate variables, the computed value describing the position of the patient at each step is saved as *result*. Thus in Section 4.3, we query this value within the relevant time period with respect to the requested threshold for our verification purposes. We provide the full declarations of the model in Appendix A.

4.2 3D motion modeling

While the actual respiratory motion is three-dimensional, it often has a clear principle component, i.e., along the superior-inferior direction. Still, the motion may be non-linear and a refined model would consider all three spatial motion components. The second simplifying model design is to represent 3D motion by a network of three 1D models. Because the other two dimensions can be approximated themselves using the same mathematical formula (see 4.1), we can reuse the 1D model for each instead of extending it to 3D variables and thusly more complex formulas. This allows us to stick with a relatively simple model that can be verified fast enough for our purposes.

However, using one model per dimension means that we need three parallel OMC processes that each generate new, separate models. Since for any online approach we need timestamped, 3-dimensional position data, this kind of input dataset needs to be retrieved by sensors to be distributed among the three processes. As a consequence, the process' results need to be aligned to make sure all models are instantiated, represent the same time slot and are successfully validated.

The synchronization problem extends further to the beam checks that have to be done on all models separately with separate threshold data: In Figure 5, we show for a single timestep the variable sets of the three models it is comprised of. Because the motion along the three axes has different positions each, each beam has necessarily six threshold values instead of one, i.e., a set of two for each model.

It is beneficial to coordinate the accuracy parameters for the three movement directions, covering the respective inaccuracies as with the 1D case, but also to control that the different dimensions are not moving completely distinctly as the separate models suggest. In the future, we want to merge different possibilities of 3D beam positions to reduce the number of verification calls.

4.3 Beam verification

The aim of the treatment is to deliver all beams, which requires that at some point during the treatment all beams have become deliverable, i.e., are not blocked due to the motion. Thus, the core verification question is whether the motion remains within the threshold that is given per beam. Our software uses the results of the verification in 4.1 to determine a list of deliverable beams per time slot among a requested list of beams. Along with the online approach, the requested beams are updated by the treatment software each time slot and the verification and selection of beams is repeated until the treatment is complete.

The beams themselves are not formally modeled, but represented by a query on the motion model that we check per beam. In case of 3D, that leads to three queries per beam, i.e., one per dimension. The query is created directly from the source data: `Pr[<= {scope}] ([] result <= {upper} && result >= {lower})`, where *scope* is set to three seconds (see 4.1), *result* is the patient position in one of the three dimensions (see 4.2), and *lower* and *upper* are derived from the beam threshold (see Figure 3 for a symmetric 1D case); `[]` is the usual "global" operator in CTL. The statistical model checker then determines the probability with which, within *scope*, the patient's position will invariantly be safe.

We use the verification results of the beams to gather a list of deliverable beams that is provided to the treatment software. Representing a safety requirement, beams may not be considered as deliverable if the threshold of any movement direction is violated according to the verification result. If the resulting probability is higher than our cutoff, a beam may be delivered and is added to the list. We present results in Section 7 with a cutoff of 0.5, whereas further experiments suggest a cutoff of 0.91. In the case of 3D, the deliverability of a beam then becomes the minimum of each of the results on the three distinct models. It is possible that a query, or in 3D at least one of three queries of a beam, is not completed within the timeslot. In these cases, the beam is also not considered to be deliverable. However, we observe these queries to take only milliseconds to complete. Depending on actual timings, requested lists of 250 beams often may be completely checked and answered.

5 Data

We evaluate on a small amount of seven sets of real patient motion data (966670 data points) which was previously published[5] and extend our dataset with synthetic data (4893007 data points) for our machine learning approach. In this section, we report on our available datasets and the limitations of synthetic data.

The respiratory motion data consists of 3 dimensional data points that are timestamped. Furthermore, we generated 40 synthetic motion curves approximating idealized respiratory motion with relatively simple changes and only in 1D along the superior-inferior axis. For each of these patients we provide[1] the corresponding line of models generated through the OMC process; for the real patients also in all three dimensions. A sample of this dataset is visualized in Figure 6. All lines of models come with their data logs, so that they can be easily visualized and analyzed. Furthermore, we provide beam lists for both 1D and 3D cases.

Generating patient breathing data, however, is not trivial. It is important to consider that not all situations can be well covered as synthetic data is much smoother and easier to align to for the OMC process, and resulting curves are simpler and distinct. Real breathing changes more often, is irregular,

[1] `https://doi.org/10.6084/m9.figshare.25061336`

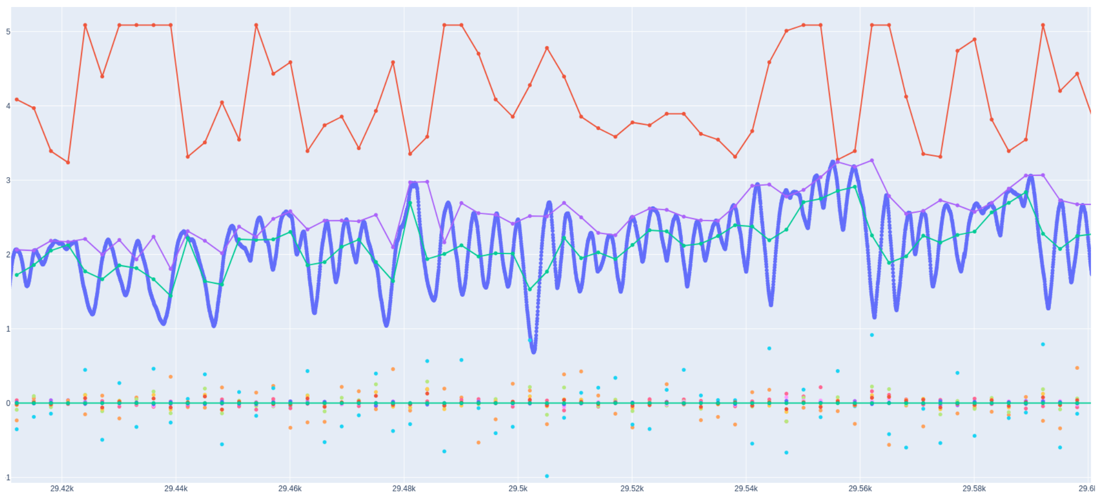


Figure 6: Partial superior-inferior portion of the breathing curve of patient DB126-Fx1 in blue along time on the x axis. Model variable values of period as red line, base as green line through the middle of the blue line, with the sum formula values c0, c1, c2, c3 and s0, s1, s2, s3 dotted. Illustrated with an expectation query of the statistical model checker for the minimal and maximal curve point by the purple line atop the blue line and the green line along 0 on the y axis.

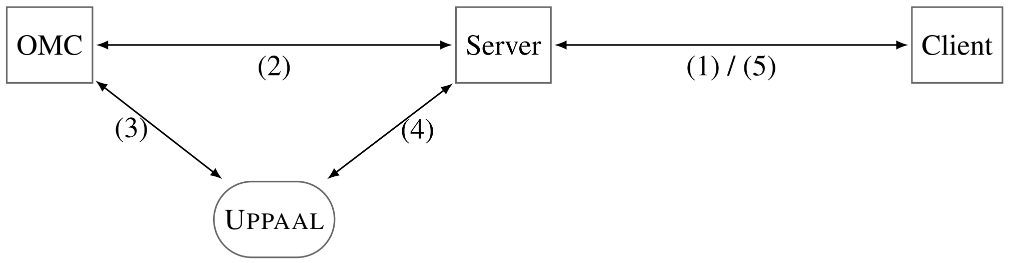


Figure 7: Architecture diagram of the software systems that we employ for our experiments. Each temporal slice consists of: (1) Client sends beam list, (2) Server uses Online Model Checking (OMC) to receive models predicting respiratory motion, (3) OMC uses UPPAAL for model creation and verification, (4) Server starts UPPAAL processes to verify beams, (5) Server sends list of beam results.

and has lots of small changes. This can be seen in Figure 6, which is an excerpt of under four minutes. There, the OMC process is constantly changing its predictions and beams differ in their verification results. Thus, it is important to use not only synthetic data to make sure observed behavior is realistic.

Compared to breathing data, beam lists are relatively simple to generate, both for real movement paths and also synthetically. For synthetic beam lists we can use existing beam lists as input to generate new lists. Each beam is distinct from the next, thus we do not need to consider changing movements along a curve. Thus, new lists just need to have the same distribution within the threshold and time values with regards to minimal, maximal, and average values overall and within quantiles.

6 Beam Checking Application

Our system needs to synchronize between the online model checking (OMC) process that generates the models themselves, the treatment robot, and our own verification of possible beams to apply. The general architecture is explained in this section and visualized in Figure 7.

The client is externalized as an application that connects over network and sends a current list of beams that are outstanding. In turn, we send a list of checked beams with their success probabilities until the updated beam list is empty. Having the client networked is a proven way for separation of concerns and allows the—possibly pre-existing—client to implement its own, e.g., safety equipment to cover cases such as sudden changes by coughing that cannot fully be anticipated by software. However, this separation induces the need to properly synchronize between these and the state of the session.

The OMC application is run directly by our software and monitored for new model files and its verification of those. It runs self-contained and does not pose requirements on the application. We currently use pre-existing lists of breathing motions that would in reality need live data. The self-validation of the OMC process also means that if verification is not passed for one model, it can recover on one of the next intervals based on updated sensor data, but until it recovers there is no current model against which to verify beams. Thus, we cannot make any recommendation within this gap and it follows that the client would need to pause the treatment, failing on the safe side.

Whenever we receive a new model from the OMC process, we execute a new set of beam verification queries, which check per beam the probability that it never violates its threshold within the timescope. We align the OMC time slots with our own verification calls, allowing a three second time slot for our own tasks and thusly verifing beams regularly, similar to the OMC process itself. However, the results of this checking process is not a new model but a recommendation for or against those beams. If, e.g., a beam failed to verify within the time slot, either by the property itself or by the end of the verification interval, it is not part of the recommended set of viable beams corresponding to this time slot.

In order to gain as much information as possible within this interval, we execute multiple verification queries in parallel. Still, it is likely that we may not have enough time to verify all remaining beams, so we prefer certain beams. First of all, we prioritize the currently running beam if there is one, but also all beams that have already been started. This reduces overhead of beam changes on the treatment robot and increases application quality, because the dosages are affected by application times and cannot precisely be controlled by starts and stops. Second, we select well applicable beams by offering a mixture of short and long applied beams, but also consider their thresholds. Beams with high thresholds, but shorter running times are clearly easier to apply even if breathing is not calm, slow, or steady. Thus for future experiments, we intend to implement a feedback loop that further prioritizes beams according to the actual current breathing situation rather than static properties.

As introduced, our architecture can be characterized as loosely coupled as we kept three separated systems in our architecture. Despite the synchronization issues this architecture necessitates, the separated concerns helped with the implementation of the software. By using the pre-existing OMC process, we only needed to introduce minimal changes, i.e., updating the code base to run within a more modern environment, including adding Java synchronization primitives and exporting the actual model files. Similarly, the client is separated from our application. Communication is realized over standard input/output streams of the process in case of the OMC process or over a TCP socket in case of the client. Instead of constraining us to a single programming language, we have specified the file formats of the transmitted data. The OMC application transmits simple log messages and UPPAAL models in their already specified file format, while the client sends and receives CSV tables with specified attributes.

The server application software is written in Rust. The beam verification queries are executed via the UPPAAL command line interface with the result parsed from its standard output, compared to the OMC software using UPPAAL's in-process library API. The approach via external processes was taken because it allows us to terminate verification processes according to our timeouts as required, which is not possible via the library itself. Furthermore, the independent processes do not require synchronized access to a singular verification server instance that the library allows to access, when evaluating multiple different queries as with beams.

For the client, we extend our in-house treatment planning Java application. We determine beam collision with the ultrasound robot by applying a projection based approach using a distance transform[8]. Here, we can also specify a static safety margin which accounts for expected target motion. Note however, that this safety margin needs to be specified before the treatment and leads to an elimination of additional beam directions. Therefore, that treatment plan quality degrades with increasing safety margin.

We extend our approach to calculate beam thresholds by computing a projection for every beam translation in the discretized motion trace. The treatment simulation starts by computing a beam order which is time optimal if the target is static, i.e., it minimizes the robot motion. Afterwards, the client cycles through the following steps: The beam list with corresponding thresholds is sent to the server for evaluation whether delivery will be feasible. After receiving the response from the server, the first feasible beam is selected, the robot is moved in position and delivery is started. Before the next evaluation cycle starts, the time optimal beam order is updated with respect to the current position of the robot. During beam delivery, it is ensured that the beam is collision free and otherwise beam delivery is halted and the cycle is started again.

7 Experiments

In our evaluation we first show preliminary results of our approach. We evaluate on 5 patient geometries for which the target tumor is located in the liver. Since treatment planning typically involves the selection of a subset of beams from a large set of randomized candidate beams, we repeat each experiment 30 times with different candidate beam sets to obtain statistically meaningful results. We report significance according to the Student's t-test with p-values smaller than 0.01.

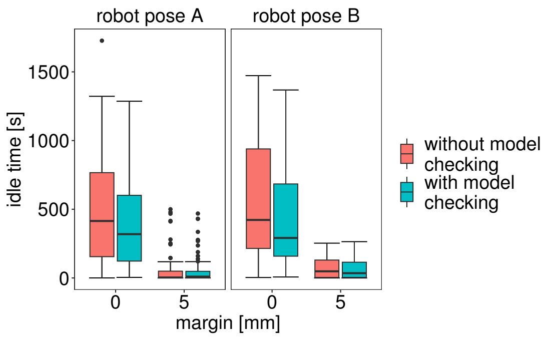
We use a motion trace of captured target motion on a previous CyberKnife treatment[5], where the breathing pattern showed a large change over time. While the particular motion pattern is not representative for every patient, it represents a challenging real-world case for ultrasound guided radiation therapy.

To evaluate the impact of our online model checking approach, we simulate complete treatments. We compare a delivery of beams where each beam is scheduled either according to the static beam list or according to our OMC approach. When a scheduled beam cannot be delivered due to the target motion, we pause the beam delivery until the beam is not blocked. We sum up this idle time to draw conclusions on the quality of our approach. Additionally, we track how often beam delivery needs to be interrupted.

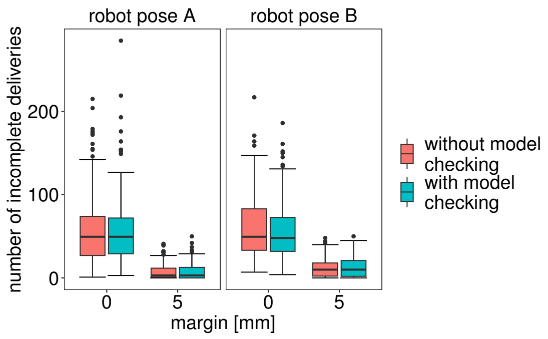
Additionally, we evaluate whether the pose of the ultrasound robot influences the results, and whether an additional safety margin could be used to reduce the idle time and the occurrences of beam collisions. Here, we apply a safety margin of 5 mm to the ultrasound robot during treatment planning and not consider beams which are too close to the ultrasound robot. Thereby, small motion of the target would not lead to a collision during treatment. However, this additional margin impacts the treatment plan quality since fewer beams are available during the treatment planning process[8].

Our results in Figure 8a show that the average idle time decreases from 418.44 s to 304.96 s for no margin and from 25.93 s to 22.35 s for a margin 5 mm. This represents a reduction in idle time of 37.21 % and 16.02 %, respectively. While the difference is significant for no margin ($p < 0.004$) it is not significant when using 5 mm margin ($p = 0.22$). While the pose of the robot can change the resulting idle time in some cases, e.g., from 44.66 s to 76.95 s for 5 mm margin and no OMC ($p = 0.001$), the difference is otherwise not significant on the average distribution ($p > 0.02$). When evaluating the number of times that a beam could not be delivered shown in Figure 8b, we do not observe a significant difference between our OMC approach and the static beam ordering ($p > 0.38$).

As Figures 8 show, increasing the margin around the ultrasound robot also decreases the number of collision events and the resulting idle time. Still, Figure 8a shows that also when introducing the margin, the idle time is even lower with our OMC approach. Crucially, the target coverage decreases



(a) idle time



(b) incomplete deliveries

Figure 8: Summarized idle time (a) and number of incomplete beam deliveries (b) per simulation for the different approaches for 5 patients and motion trace DB126_Fx1. Experiments are repeated 30 times with different beam sets to increase statistical significance.

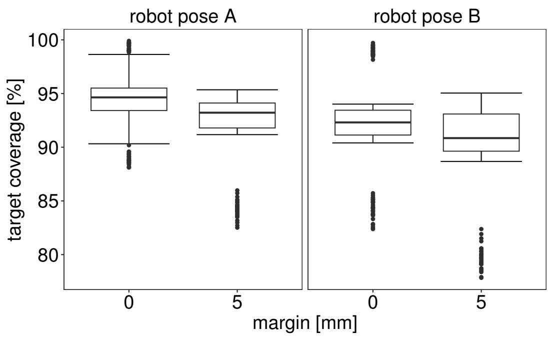


Figure 9: Example of resulting target coverage for 5 patients. Experiments are repeated 30 times with different beam sets to increase statistical significance.

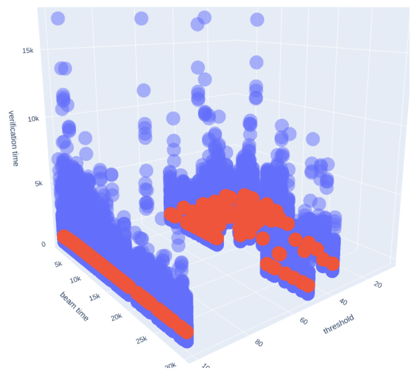


Figure 10: Predictions in red, actual data in blue. Note the difference in transparency: The blue circles are much more varied than the red.

when introducing a margin as Figure 9 shows. The target coverage refers to the proportion of the target which receives at least the prescribed dose and is an important clinical goal which influences the clinical outcome. Therefore, the safety margin is degrading the treatment quality, while OMC does not negatively influence the delivered dose as per Figure 8b. Note also that extreme outliers exist for certain patients with substantially worse coverage when applying an increased safety margin.

8 AI Enhancements

We tried to extend the information we can gain from our data using machine learning methods. First, we tried an experiment on beam verification time learning. Second, we tried an experiment on breathing pattern classification. Both would allow us to change the prioritization of beams to ensure a more fitting list of verification results or even a larger list of verification results when we exclude beams that are not likely to succeed in verification by failing either time requirements or anticipated threshold violation. Although both experiments failed, we briefly report in this section their respective setting and outcome. These experiments were implemented in Python to make use of the widely known *scikit-learn*[10] library that provides good default implementations of common learning algorithms.

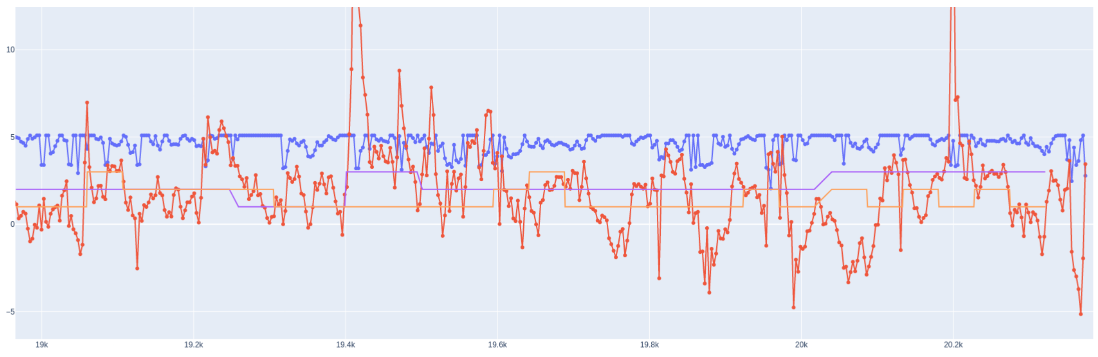


Figure 11: Attempt at classification. Excerpt from patient DB114-Fx1 represented by the model variables base and period in red and blue dotted lines, respectively. Classification categories range from 1 to 3. Predicted category in yellow, truth in purple.

In the first experiment, learning verification time, we first used our synthetic data. Then, we varied queries and, third, replaced the synthetic data by real data. In all cases we applied a beam list to a single patient dataset, but we also tried to scale the number of beams from the typical 250 to 2500. Regarding the queries, we tried to vary between different kinds of queries, e.g., existential and universal, but did not notice significant differences. The predictions were consistently at the lower end of verification time with more visible noise in the actual verification times. This can be seen visually in Figure 10. Thus, this learning approach does not yield any opportunity for priorization. Our explanation is that the synthetic data together with rather fast queries, that complete within a few hundred milliseconds, is rather unlearnable noise, because there is no discernable pattern within the parameters for longer durations. Also, simply increasing the number of queries from about 250 to about 2500 to have more data to learn may not help when the overall time becomes larger faster than any results are improving. Based on the experiments with real data, we could not make predictions on verification time that were significant enough to be of any help.

In the second experiment, we manually labeled a full motion dataset with three categories representing different levels of breathing periods, from calmer to more irregular. We applied a multi-layer perceptron classifier and varied the solver configuration with the options that *scikit-learn* offers. As can be seen in Figure 11 almost all classifications are wrong, specifically when the period is spiking the level is classified as lower. Based on the classification of a real breathing sequence, the prediction does not yield the desired results of differentiating more irregular from calmer breathing periods. Especially extreme movements are completely removed from the difficulty class. Thus, a better approach for classification is an algorithm specialised to detect more extreme cases in the movement data. We refrained from integrating the results of our AI experiments in our beam scheduling application.

9 Summary

Robotic radiation therapy with robotic ultrasound guidance represents an interesting area of application for OMC. We described and implemented an approach for beam scheduling which verifies feasibility of beam delivery during the treatment with OMC. Furthermore, we provide data and generated models. Our preliminary results show that, avoiding beam collision, reductions in the idle time ranging from 16.02 % to 37.21 % are possible. Comparatively, a naive machine learning approach to predict beam delivery feasibility does not achieve the same performance as OMC while also being harder to reason about.

A Model declarations

Global declarations

```
const double accuracy = 100.0;

const double period = 3469.0;
const double drift = 0.0;

double base = 2.5019;
double a[4] = { -0.1959, 0.0295, -0.0022, -0.0169 };
double b[4] = { -0.4023, 0.0294, 0.033, 0.013 };

double v[4];
double result;

double time;
broadcast chan step;

double frequency = 2 * 3.14159265358979323846 / period;
```

System declarations

```
Clock = Timer(38);

const double accrate = (100.0 - accuracy) / 15.0;

CMP = CoeffModifier(frequency, 1.0 * accrate * 0.0001, 10, 1000);
CMB = CoeffModifier(base, 1.0 * accrate * 0.25, 10, 1000);
TM1 = TermModifier(a[0], b[0], 1.0 * accrate * 0.1, 10, 1000);
TM2 = TermModifier(a[1], b[1], 1.0 * accrate * 0.1, 10, 1000);
TM3 = TermModifier(a[2], b[2], 1.0 * accrate * 0.1, 10, 1000);
TM4 = TermModifier(a[3], b[3], 1.0 * accrate * 0.1, 10, 1000);

First = FSTerm(v[0],a[0],b[0],frequency,1);
Second = FSTerm(v[1],a[1],b[1],frequency,2);
Third = FSTerm(v[2],a[2],b[2],frequency,3);
Fourth = FSTerm(v[3],a[3],b[3],frequency,4);

TermSummer = Summer(result,v,base);

system Clock, First, Second, Third, Fourth, TermSummer, CMP, CMB, TM1, TM2, TM3, TM4;
```

Automaton Parameters

FSTerm `double &v, double &a, double &b, double &freq, int k`

Summer `double &s, double &v[4], double &offset`

Timer `int rate`

TermModifier `double &a, double &b, double rate, int minWait, int maxWait`

CoeffModifier `double &c, double rate, int minWait, int maxWait`

Automaton Declarations

Timer `clock dt;`

TermModifier `meta double m; clock t;`

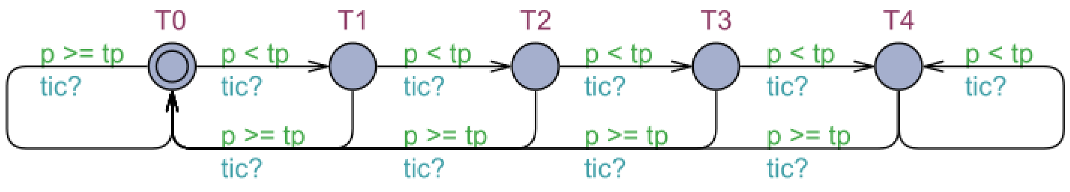
CoeffModifier `clock t;`

B Evaluation model

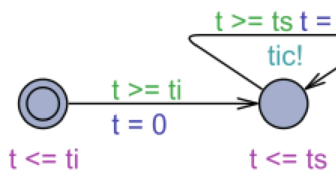
The evaluation model is directly used as per its introduction in Figure 7.6 of [12], and shown here in Figure 12. It is queried by `E<> ((A1.T3 || A1.T4) && (A2.T3 || A2.T4) && (A3.T3 || A3.T4))` to check if a high enough error tier is reached through repeated condition violation. The probability p that is referenced is obtained by querying the motion model using the query

$$Pr[\diamond_{\leq t_o - t_m + t_+} t_o - t_- \leq t_p \leq t_o + t_+ \wedge x_0 - x_- \leq x_p \leq x_o + x_+],$$

where t_m is the model's creation time, $t_o > t_m$ the later observation time with x_o the observed values, t_+, t_-, x_+, x_- range parameters defining the bounding box around the observed (t_o, x_o) and predicted (t_p, x_p).



(a) Tier template



(b) Timer template

Figure 12: The Tier template is instantiated per data series, the Timer template is instantiated once. For each data series, p is the current validity probability and tp is the corresponding threshold. The probability is estimated by UPPAAL-SMC that within an observed time period, observed values are predicted by the original model[12].

References

[1] Sven-Thomas Antoni, Jonas Rinast, Xintao Ma, Sibylle Schupp & Alexander Schlaefer (2016): *Online model checking for monitoring surrogate-based respiratory motion tracking in radiation therapy*. International Journal of Computer Assisted Radiology and Surgery 11(11), pp. 2085–2096, doi:10.1007/s11548-016-1423-2.

[2] Gerd Behrmann, Alexandre David & Kim G. Larsen (2004): *A Tutorial on* UPPAAL. In Marco Bernardo & Flavio Corradini, editors: Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, LNCS 3185, Springer–Verlag, pp. 200–236, doi:10.1007/978-3-540-30080-9_7.

[3] Lei Bu, Qixin Wang, Xin Chen, Linzhang Wang, Tian Zhang, Jianhua Zhao & Xuandong Li (2011): *Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior*. SIGBED Rev. 8(2), p. 7–10, doi:10.1145/2000367.2000368.

[4] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis & Danny Bøgsted Poulsen (2015): UPPAAL *SMC tutorial*. International Journal on Software Tools for Technology Transfer 17(4), pp. 397–415, doi:10.1007/s10009-014-0361-y.

[5] Floris Ernst, Robert Dürichen, Alexander Schlaefer & Achim Schweikard (2013): *Evaluating and comparing algorithms for respiratory motion prediction*. Phys Med Biol 58, p. 3911 – 3929, doi:10.1088/0031-9155/58/11/3911.

[6] Stefan Gerlach, Theresa Hofmann, Christoph Fürweger & Alexander Schlaefer (2022): *AI-based optimization for US-guided radiation therapy of the prostate*. Int J Comput Ass Rad 17, p. 2023 – 2032, doi:10.1007/s11548-022-02664-6.

[7] Stefan Gerlach, Theresa Hofmann, Christoph Fürweger & Alexander Schlaefer (2023): *Towards fast adaptive replanning by constrained reoptimization for intra-fractional non-periodic motion during robotic SBRT*. Med Phys 50, p. 4613 – 4622, doi:10.15480/882.5073.

[8] Stefan Gerlach, Ivo Kuhlemann, Philipp Jauer, Ralf Bruder, Floris Ernst, Christoph Fürweger & Alexander Schlaefer (2017): *Robotic ultrasound-guided SBRT of the prostate: feasibility with respect to plan quality*. Int J Comput Ass Rad 12, p. 149 – 159, doi:10.1007/s11548-016-1455-7.

[9] Tao Li, Feng Tan, Qixin Wang, Lei Bu, Jian-Nong Cao & Xue Liu (2014): *From Offline toward Real Time: A Hybrid Systems Model Checking and CPS Codesign Approach for Medical Device Plug-and-Play Collaborations*. IEEE Transactions on Parallel and Distributed Systems 25(3), pp. 642–652, doi:10.1109/TPDS.2013.50.

[10] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot & Édouard Duchesnay (2011): *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research 12, pp. 2825–2830, doi:10.5555/1953048.2078195.

[11] Mona Qanadilo, Sufyan Samara & Yuhong Zhao (2013): *Accelerating Online Model Checking*. In: 2013 Sixth Latin-American Symposium on Dependable Computing, pp. 40–47, doi:10.1109/LADC.2013.20.

[12] Jonas Rinast (2015): *An online model-checking framework for timed automata*, doi:10.15480/882.1253. Available at `http://tubdok.tub.tuhh.de/handle/11420/1256`.

[13] Gerald Sauter, Henning Dierks, Martin Fränzle & Michael R. Hansen (2009): *Lightweight hybrid model checking facilitating online prediction of temporal properties*. In: Proceedings of the 21st Nordic Workshop on Programming Theory, 21, DTU Informatik, Danmarks Tekniske Universitet, pp. 20–22.

[14] Alexander Schlaefer & Achim Schweikard (2008): *Stepwise multi-criteria optimization for robotic radiosurgery*. Med Phys 35, p. 2094 – 2103, doi:10.1118/1.2900716.

[15] Matthias Schlüter, Christoph Fürweger & Alexander Schlaefer (2019): *Optimizing robot motion for robotic ultrasound-guided radiation therapy*. Phys Med Biol 64, doi:10.1088/1361-6560/ab3bfb.

[16] Krishna Sudhakar, Yuhong Zhao & Franz-Josef Rammig (2016): *Efficient integration of online model checking into a small-footprint real-time operating system*. *Concurrency and Computation: Practice and Experience* 28(14), pp. 3773–3797, doi:10.1002/cpe.3712.

[17] Yuhong Zhao & Franz Rammig (2012): *Online Model Checking for Dependable Real-Time Systems*. In: *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 154–161, doi:10.1109/ISORC.2012.28.