Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Model Checking the FlexRay startup phase

S. Cranen

12/01

ISSN 0926-4515

Computer Science Reports 12-01
Eindhoven, January 2012

Model checking the FlexRay startup phase

S. Cranen
s.cranen@tue.nl

Eindhoven University of Technology

Abstract. This report describes a discrete-time model of the startup phase of a FlexRay network. The startup behaviour of this network is analysed in the presence of several faults. It is shown that in certain cases a faulty node can prevent the network from communicating altogether. One previously unknown scenario is uncovered.

1 Introduction

In the year 2000, a consortium was established with the goal to design a new, time-triggered communication protocol for use in the automotive industry that would outperform CAN and TTP in both speed and reliability. At the end of 2009, the consortium was disbanded, leaving a final version of a time-triggered protocol called FlexRay. The protocol definition is currently being transformed into an ISO standard.

Already in 2006, the first commercially available cars were equipped with FlexRay networks, enabling new algorithms for vehicle control because of its higher bandwidth.

Since FlexRay will be the basis for communication in many vehicles to come, we would like to establish that the protocol itself is inherently correct, *i.e.*, that implementing a system according to the latest specification leads to a system that behaves predictably. We restrict ourselves to analysing the startup behaviour of FlexRay networks.

As the specification has been declared final, there is no way to avoid any unwanted behaviour that might occur (that is, any behaviour that an end user might find inconvenient). For any such behaviour we would therefore also like to know whether an external party such as a so-called central bus guardian can be used to improve such behaviour without having to alter the protocol.

In this report, we start by giving a brief overview of the FlexRay protocol. We then discuss some of the research done towards verifying parts of the FlexRay protocol, and we explain how our research compares. After this, we give a more detailed description of the startup phase of a FlexRay network, which is the part of the protocol that we wish to analyse. We show how the startup phase can be modelled using the mCRL2 language. We then analyse the startup behaviour in several scenarios with noisy channels and faulty nodes. To conclude, we discuss the results and suggest how the startup behaviour of FlexRay networks might be improved by a central bus guardian.

2 The FlexRay protocol

We base our analysis on the 2.1A version of the protocol [8]. For those parts of the protocol that this report discusses in detail, the publicly available 3.0.1 version is identical.

The aim of the protocol, as stated in the accompanying requirements document [9], is to provide both 'deterministic' communication and 'on-demand' communication. Essentially, this means that some bandwidth is reserved for communication between specific nodes in the network, and some bandwidth is assigned at runtime to nodes that wish to send data.

All scheduling in the protocol is based on a shared time base, which is established through a clock synchronisation mechanism that is implemented on the same physical channel as the communication uses. In the 3.0.1 specification, additional means of clock synchronisation are added, namely synchronisation on an externally supplied clock, or synchronisation on another FlexRay network.

To communicate over a FlexRay network, the network must first be brought to a state in which it is synchronised, and aware of the communication schedule and the current position in that schedule. Firstly, the network is woken up; a special symbol is sent over the channel that causes nodes to switch to an active mode.

When woken up, a distinction is made between *coldstart* nodes and non-coldstart nodes. Coldstart nodes will attempt to initiate communication on a silent bus when woken up, where non-coldstart nodes will wait until they detect ongoing communication and then integrate. For each node (coldstart or non-coldstart) we call the period in which it is awake and trying to establish communication or trying to integrate into existing communication the *startup phase* of that node.

During this startup phase, clock synchronisation is initialised and some consistency checks are done. When startup is completed, all nodes in the network should have reached agreement on what the global time is. Communication proceeds according to a schedule, of which one execution is called a *communication cycle*. The term cycle is somewhat confusing here, because nodes keep track of a cycle counter, which is reset periodically (with a period smaller than 65 cycles), and which must correspond to the other nodes' cycle counters. Hence the schedule is only truly cyclical in this period of cycles. The structure of a communication cycle is however always the same, and we will therefore use the term communication schedule to refer to this structure.

The communication schedule consists of two parts: the so-called *static segment* and the *dynamic segment*. The lengths of these two segments are fixed. The communication in the static segment is based on *time-division multiple access* (TDMA). Every node has time slots assigned to it in which it may send, while all other nodes will attempt to receive the message it sends.

The dynamic segment is also TDMA based, the difference being that slots are smaller and hence called 'minislots'. They are not assigned to a specific node; instead, they are assigned to nodes based on their priority. The first minislot in the dynamic segment is allocated to the highest priority node. A node can use a series of minislots (possibly 0) to send its on-demand data, followed by a single minislot that indicates the end of the transmission. The next minislot is then assigned to the highest priority node that did not yet send anything. As the length of the dynamic segment is fixed, it may be the case that no more bandwidth is available for lower priority nodes if higher priority nodes claim many minislots.

The FlexRay protocol states that nodes may be connected to at most two *channels*. A channel is simply a communication medium such as a copper wire or optical fibre (depending on hardware support). The two channels may be used to create redundancy, or to connect to two different sets of nodes.

3 Related work

The FlexRay protocol has been studied quite extensively, from different perspectives. In this section we give a brief overview of previous studies known to us, and describe the aspects that these studies cover.

The German Verisoft project[1] is a project in the automotive realm that has the verification of the FlexRay protocol as a sub-goal. Within this project, Kühnel et al. aim to provide a framework in which distributed applications can be verified if they use a combination of an OSEKTime compliant (real-time) operating system, FTCom (a fault-tolerant communication layer for OSEKTime operating systems) and FlexRay communication [17]. They use the FOCUS [5] language as a basis for their toolkit. A model is created on a specification that is "based on" the FlexRay protocol specification version 2.0. They apply the following abstractions to simplify their model [16]:

1. All clocks are synchronized, i.e. clock synchronization is not modelled.
2. Start-up behaviour is not modelled (because the clocks are synchronised).
3. The coding/decoding process is not modelled.
4. Bus guardians are not modelled.
5. Only the static segment of a communication cycle is modelled, not the dynamic segment.
6. Slots are assumed to have a size of one tick in FOCUS.
7. Lossiness of the channel is not modelled.
8. Single-channel nodes are modelled.

The above lists the aspects of FlexRay that are not modelled, but we could not find a textual explanation of the part of FlexRay that is incorporated in the model. Instead, a specification in FOCUS is given directly, and it is shown—using a theorem prover—that the used interface specification of the FlexRay component is indeed refined by the presented model [22].

The clock synchronization mentioned in abstraction 1 is a modification of a clock synchronisation protocol described by Lundelius and Lynch [18]. Barsotti et al. have verified (amongst other protocols) the latter [1,11] using a combination of

[1] http://www.verisoft.de/SubProject6.html

a theorem prover and an SMT[2] solver. Zhang notes, however, that the correctness of the FlexRay clock synchronisation protocol does not trivially follow from the correctness of Lundelius and Lynch's algorithm [25].

The start-up behaviour (not modelled because of abstraction 2) is addressed by Malinsky in [19,20]. He uses UPPAAL to create a timed-automata representation of a system consisting of two *coldstart* nodes and one non-coldstart node. Using a few different settings for a number of FlexRay parameters, this system is checked for deadlock, and it is checked that the system starts up normally. It should be noted that this setup is not a valid FlexRay setup, because in a network with three nodes all nodes must be coldstart nodes, according to [8]. However, the requirements document [9] does state that startup must succeed when, due to a fault, only two coldstart nodes are active.

Local bus guardians [7] are considered by Zhang, who proves three functional properties under the assumption of synchronised clocks [26]. This partially addresses abstraction 4, although central bus guardians [6] are not taken into account. Zhang does however mention research done towards systems employing a central bus guardian in another industry standard, Time-Triggered Architecture[3].

Pop et al. have looked into properties related to the dynamic segment in a FlexRay communication cycle as meant in abstraction 5. Their analysis is however of a rather different kind, as they provide a schedulability analysis based on a correctly working protocol [21]. Such results can be extremely useful in proving correctness of a distributed application, as it may guarantee that—under given circumstances—data will travel through the system.

Botaschanjan et al. present a methodology that uses the framework developed in the Verisoft project [4]. The method aims at formally modelling tasks (a concept taken from the automotive industry) in the FOCUS tool. The application logic can then be verified, while automatic, verified code generation should ensure that these properties still hold in the deployed system. Later this approach is refined and formalised [3].

Steiner uses the SAL model checker to find failures in the startup protocol [23,24]. He identifies a scenario in which the system does not start up due to a single fail-silent node.

In this report, we try to establish some correctness results on the discrete behaviour during the startup phase of the FlexRay protocol. This aim resembles that of Malinský, but we do not focus on modelling timing properties. The approach of Steiner et al. is comparable, but their model is hand-crafted and therefore does not have an obvious one-to-one correspondence with the specification. The time model chosen in their model is also coarser than the one we present. Our aim is to create a model that can be easily seen to correspond to the specification document, and of which we can describe fairly accurately the characteristics that do not match reality.

[2] Satisfiability Modulo Theories

[3] http://www.vmars.tuwien.ac.at/projects/tta

4 The FlexRay startup phase

We study the behaviour of a FlexRay network in the startup phase. The startup phase starts when a node is awake (*i.e.*, it is powered on and ready to send and receive), and ends when the node has succesfully integrated into the network or when it has done a number of unsuccessful integration attempts.

Our aim is to create a model that allows us to analyse the state behaviour of the FlexRay protocol during startup. We focus on creating a model that has a direct link with the SDL models in the FlexRay protocol specification.

We wish to assess the correctness of the FlexRay startup protocol itself. The protocol description is not very clear about what correct operation entails, so we discuss this issue separately in Section 4.2.

The model that we provide later in this document is not complete; like the models mentioned in Section 3, we must simplify the problem to make it tractable. The applied simplifications are discussed in Section 4.3. Finally, we discuss the model itself in Section 4.4.

4.1 Operation

The startup phase in a FlexRay network is initiated by so-called *coldstart* nodes. These nodes have the ability to start communication on a network. Regular nodes must wait for communication to start on the network, after which they may join in. Coldstart nodes send special 'startup frames' on fixed locations in the schedule. These startup frames are the only frames that are sent during startup of a network, and they are sent every cycle.

All nodes start listening to the bus as soon as they are awake. If a listening node decodes a startup frame header or a collision avoidance symbol (CAS) from the bus, then it will assume that communication has already started and it will try to integrate into the schedule.

Integrating into ongoing communication is done by first waiting for the same startup frame header in two adjacent communication cycles, so the node's clock can be adjusted to account for any relative clock drift that the sender might have. Once it has adjusted its clock speed, it will listen for a few more cycles to see whether communication is indeed still going as expected, and eventually it will join in.

If a node does not hear anything on the bus for two communication cycles, it sends a collision avoidance symbol, after which it starts sending startup frames according to its schedule. The CAS is always sent at a specific point in a communication cycle, so, effectively, every node waits for a different amount of time after sending a CAS before sending a startup frame. It is this timing difference that implements a leader election protocol: should more than one node (almost) simultaneously send a CAS, then the first of them to send a startup frame will cause the others to stop sending. These nodes will then once more listen to the bus, and integrate to the remaining node.

After a node has successfully started communicating on the bus, it observes the traffic for another two cycles and restarts the whole procedure if it does

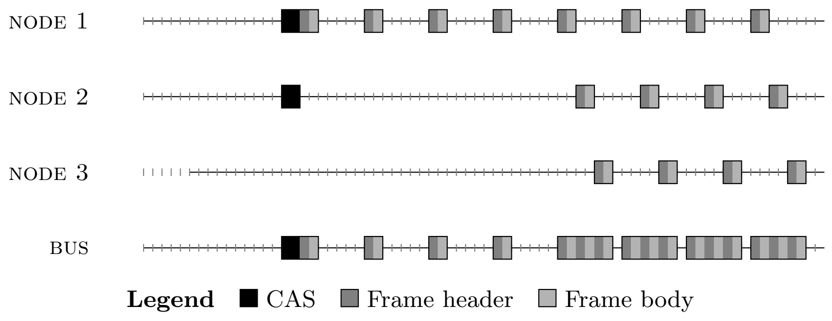

Fig. 1. Three nodes starting up. Black is CAS, dark grey is frame header and light grey is frame body. The BUS line is the combined signal of the three nodes.

not see the other parties anymore. The network traffic of a successful startup is depicted schematically in Figure 1.

In our report, we assume that all nodes are coldstart nodes, and hence we will sometimes talk about ‘nodes’, rather than ‘coldstart nodes’. Furthermore, we assume that all frames are in fact startup frames, so again we do not distinguish between the two.

4.2 Correctness

Although the FlexRay specification documents do not say what correct operation means, we do find the following in the requirements document [9]:

> *To say that “a cluster is able to start up in the presence of a fault” has a meaning that depends on the type of fault.*
> - *Fault class 1: The fault is associated to a channel or a star: All nodes which are intended to participate in communication and are connected to the other channel reach a state where they communicate to one another as scheduled. They reach this state within a defined maximum time.*
> - *Fault class 2: The fault is associated to a node:* All *fault-free nodes which are intended to participate in communication reach a state where they communicate to one another as scheduled. They reach this state within a defined maximum time.*
> - *Fault class 3: Transient fault: All nodes which are intended to participate in communication reach a state where they communicate to one another as scheduled. They reach this state within a defined maximum time (For a value see the requirements specification).*

The requirement then gives some faults that FlexRay networks should be robust against. In some cases, the use of both FlexRay channels is required to ensure correct operation. The scope of this document only covers single-channel networks, and therefore all faults in class 1 are considered fatal (if the only used channel is not working, then no communication can take place).

The class 2 and 3 faults within the scope of our investigations are the following (we again quote, and numbering corresponds to the numbering in [9]).

3 *A single I/O signal/pin connecting two of the components communication module, bus guardian or bus driver has become disconnected inside one of the node in the cluster. e.g. bus driver's Transmit Enable input disconnected.*

3b *signal wave form degraded on one channel [only has to work in certain circumstances].*

4 *A single arbitrary node in the cluster is not communicating on all attached channels for whatever reason. It doesn't transmit anything, or it starts to transmit somewhat later than the other nodes.*

5 *A single clock oscillator in the cluster erroneously runs at the wrong frequency. [...]*

6 *A node (e.g. coldstart node) cannot receive any communication element on all its attached channels.*

8 *A star cannot receive any communication element of a certain branch [only has to work in certain circumstances].*

9 *A periodic reset event is present in a node.*

12 *One single bit of a communication element on one channel flips.*

13 *For a given time of less than one frame length, all present channels are forced to an arbitrary pattern.*

14 *A bus driver in a node cannot receive anything.*

15 *A bus driver in a node cannot transmit anything.*

17 *A coldstart node sends sporadically CASs. After occurrence, the fault does not manifest itself for at least 10 communication cycles.*

18 *No node is operational except of 2 fault free nodes and these two nodes are assigned to perform startup ("coldstart nodes"). (Req ID 326)*

Our aim is to verify that in these cases, the correctly functioning nodes in a FlexRay network will indeed start up as usual. We saw before that, according to the requirement, startup has succeeded when all non-faulty nodes communicate to one another as scheduled. This will never be the case in for instance fault scenario *17*, as the communication will be periodically interrupted by the CAS.

We reinterpret the definition of correct startup to mean that on all nodes the startup protocol has terminated successfully, and during one cycle in which no startup protocol is active anymore, every frame that is sent by a non-faulty node is received by all other non-faulty nodes, unless a faulty node or transient fault prevents reception.

4.3 Simplifications

Although our choice of modelling language would permit us to exactly describe every detail of the FlexRay protocol, we must abstract away some of the complexity in order to be able to automatically analyse our model. The aim is to have a model of which the behaviours are a subset of the behaviours that a real system might display. Note that this means that proving the model correct is not enough to conclude that the actual system will behave correctly. However, any unwanted behaviour that is detected in the model will indeed be observable in the actual system.

Important but straightforward restrictions are that in our model, only a single channel is used, each node is assigned a single static slot, and all frames are startup frames. The dynamic segment remains unused.

Two concepts that require more complicated measures are the notions of time and data. We discuss these in more detail, and try to give an intuition about the implications of the simplifications for the accuracy of our model.

Time In [20], Malinský and Novák present a model of the startup phase of a FlexRay network. Their approach is to make no assumption about the correct working of the clock synchronisation protocol, and they subsequently show that several parameters of their model can be adjusted in such a way that the network model does not properly start up.

Our approach is fundamentally different in a number of ways. We assume a discrete-time global clock, *i.e.*, we assume that for all nodes in the network the clock drift is zero and that the nodes' timed behaviour can be modelled using time slots. The advantage of this approach is that the clock synchronisation process need not be modelled, which is a great advantage indeed as it is a very data-intensive process. The downside is that the model becomes less realistic, as many scenarios are not permitted by our model (namely the ones in which the clocks of the nodes are not synchronised). However, because we only discard behaviour by making this assumption, any faulty behaviour in the model should still correspond to faulty behaviour in the real system.

The resolution for time slots is chosen to be one bit length (`gdBit` in [8]). When clock synchronisation works correctly, the external behaviour of a node can be modelled using this time base, as all lengths of symbols sent over the bus are defined (either directly or indirectly) as multiples of one bit length.

Another issue concerning time is caused by the semantics of SDL. Signals are sent to processes, which store them in queues. Each such queue is partially visible, the invisible part modelling signals that have not reached their destination yet, and the visible part modelling those signals that are ready to be processed by the receiver.

We assume that signals are delivered immediately, thus avoiding the need for the invisible part of the queue. The need for the visible part of the queue stems from the fact that a process can be busy performing a calculation when events arrive. We assume that calculations are also done instantly. Since the queues in the SDL semantics are FIFO, we can now simply process events when they come in.

This abstraction comes at the cost of modelling only part of the possible behaviour of the real system. By synchronising the bus communication per bit, we cannot detect any faults in the clock synchronisation protocol. This however is outside the scope of our investigations.

It is more difficult to assess what part of the state behaviour we might have excluded by this synchronisation. Obviously, we cannot model a system in which one node starts sending a symbol less than a bit time after another node; a scenario that might occur when more than one node is sending a CAS symbol. It is unclear to what extent this may influence the behaviour of the system.

Data During the startup phase, the only data communicated over the bus are collision avoidance symbols and startup frames. As mentioned before, our time model assumes a global clock with a 1-bit resolution, so these symbols are encoded one bit at a time.

We do not wish to model the sending of actual bit patterns, as that would lead to an enormous amount of allowed combinations. Instead, we model the relevant symbols by using an encoding that models some relevant properties.

The symbols that are relevant during the startup procedure are the following.

FRAME_HEADER(id) A (startup) frame header sent by node *id*. According to the specification, the frame header would carry information such as the frame identifier, the length of the payload that follows, a CRC, the cycle counter and some indicator bits.

FRAME_BODY(id) The body of a (startup) frame sent by node *id*. The body in reality carries the payload of a frame and a CRC checksum.

Both the frame header and the frame body contain a CRC checksum. This check is in place to detect corrupted frames. We make the assumption that this check is flawless, and model this by sending the *id* of the sender along with every bit in a frame. This enables the receiver to decide whether a string of received bits forms a valid frame.

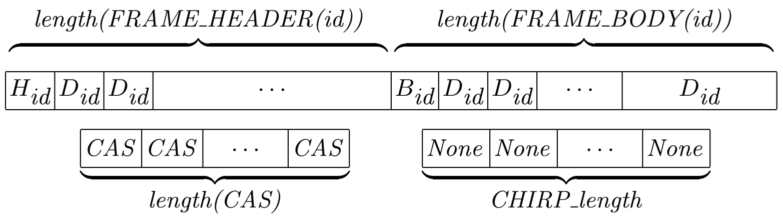

Fig. 2. Encoding of symbols on the bus.

The following two symbols are much simpler, but pose another technical difficulty: these symbols are defined as a period during which the bus is in a certain state. The lengths of these periods are given in microseconds, so the symbols cannot in general be seen as bit patterns. However, the periods are both multiples of *gdBit*, which is the length of one bit. In our model, we will therefore model them as bit patterns, because our time model allows it.

CAS A collision avoidance symbol, the length of this symbol is defined by the protocol constants *cdCASRxLowMin* and *cdCAS*, which are set to 29 *gdBit* and 30 *gdBit*, respectively.

CHIRP The channel idle recognition point (CHIRP) is defined as an offset relative to the last time activity was seen on the channel. The length of this symbol is defined by the protocol constant *cChannelIdleDelimiter*, which is defined to be 11 *gdBit*.

The mCRL2 encoding of these symbols is schematically depicted in Figure 2. It is clear that this encoding does not allow us to model certain corner cases; every scenario in which frame data from two different sources is accidentally interpreted as a valid frame cannot be modelled, for instance. This is because in our model the recipient of a bit can identify the sender of that bit, which corresponds to the assumption that the CRC check can detect any form of data corruption.

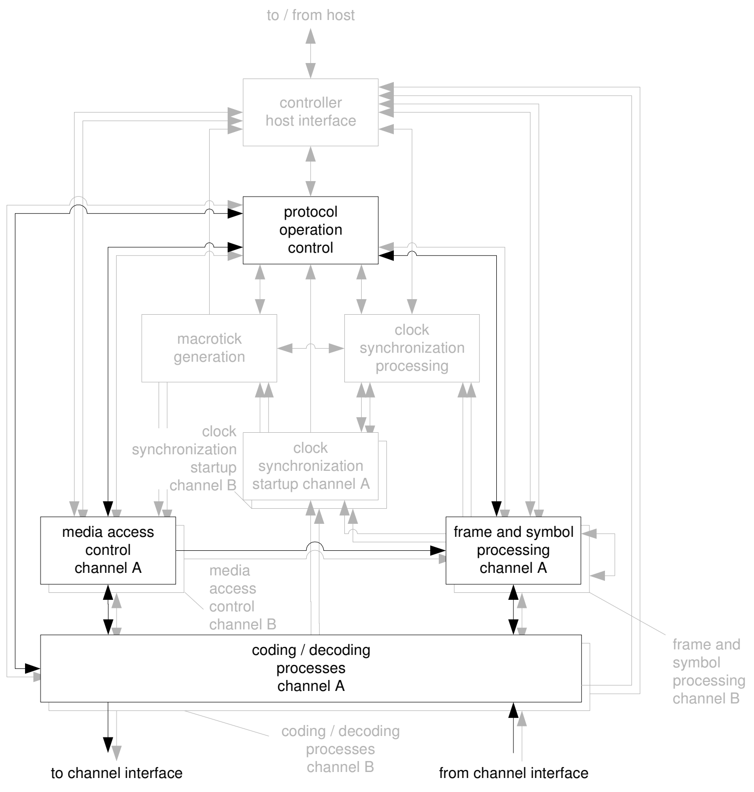
to / from host
controller host interface
protocol operation control
macrotick generation
clock synchronization processing
clock synchronization startup channel B
clock synchronization startup channel A
media access control channel A
frame and symbol processing channel A
media access control channel B
coding / decoding processes channel A
frame and symbol processing channel B
coding / decoding processes channel B
to channel interface
from channel interface

Fig. 3. The context of the POC process. Grey areas are not modelled.

4.4 Model

We use the mCRL2 modelling language [12,14] to model a small FlexRay network. In the following, we assume a basic knowledge of mCRL2, and explain how our model relates to the FlexRay protocol specification.

Structure

We model a network of three coldstart nodes during the startup phase. The structure of a single node, as presented in the FlexRay specification, is shown in figure 3. We will only model the Process Operation Control component in detail, and the simplifications described in the previous section allow us to not model the components shown in grey altogether. For the remaining components, a very abstract model is created.

Figure 4 shows a high level model of our system. Three nodes and one bus are each modelled as a separate process. The processes run in parallel, and communicate by synchronising actions. Each process has a notion of a clock tick, which is modelled using an action. We require that if a clock tick occurs in one process, it must simultaneously occur in all other processes. This type of synchronisation is called barrier synchronisation; the dotted line represents the barrier.

The `Put` and `Get` actions occur between clock ticks, and define what a process writes and reads in the current time slot. The system will in each time slot first perform three `Put` actions, one for each node, followed by three `Get` actions. In a fault-free scenario, the data that is read by each node will be the same; it will be the combination of the data that was written by the three nodes.

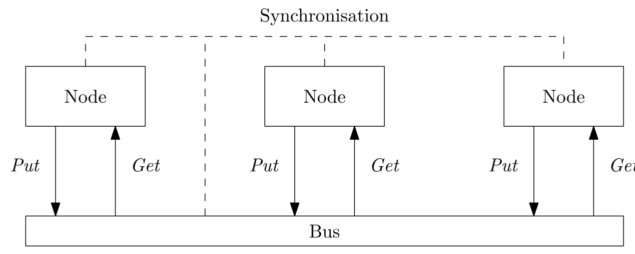

Fig. 4. The system model consists of three nodes and one bus, that communicate by `Put` and `Get` actions.

A more detailed view of the `Node` and `Bus` processes is given in Figure 5. Their interface consists of the actions involved in the barrier (again indicated by the dotted line), and the actions named `put`, `get`, `put′` and `get′`. The first two represent a node respectively writing to or reading from the bus. The last two represent the bus receiving data from a node and providing data to a node. The `put` and `put′` together form the `Put` action from Figure 4 when they occur simultaneously. Moreover, we do not allow `put` or `put′` to occur on their own;

a node can only read data from the bus if the bus is providing that data, and likewise, a node can only write data to the bus if the bus is receiving it. This type of synchronisation is a widely used technique to model communication between processes running in parallel.

The `Node` process consists of three subprocesses that run in parallel: the CODEC, MAC and POC. These components correspond closely with the SDL processes described in [8]. Figure 5 also shows the flow of data between the components. The names on the arrows are the actions involved in the communication between components (which is again modelled in the same way as the communication between the nodes and the bus).

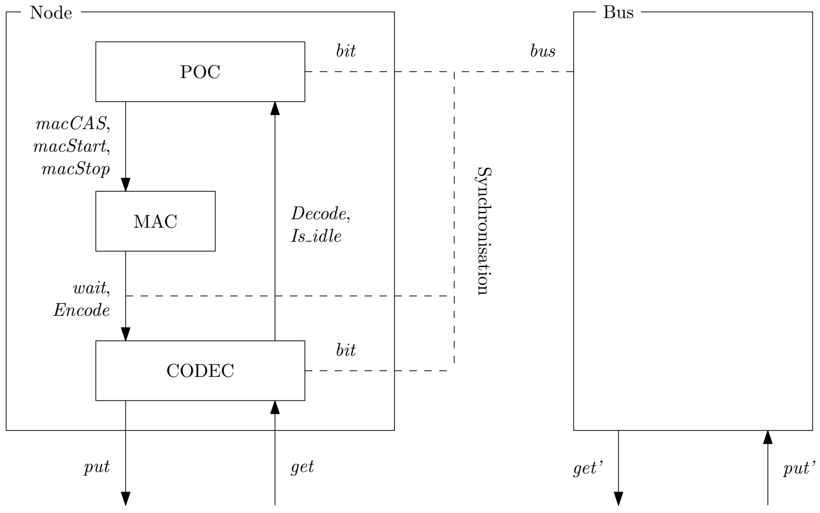

Fig. 5. Detail of the `Node` and `Bus` processes.

Figure 6 shows the mCRL2 code that corresponds to the structure of Figures 4 and 5.

For a single node, the POC (*process operation control*) process is the process that drives the startup protocol. It is defined in terms of SDL macros in [8] Chapter 7.2. Figure 7 shows the parts of the POC process that are modelled.

Bus model

We wish to verify properties of a network with deaf nodes (nodes that cannot read anything from the bus) and mute nodes (nodes that cannot write anything to the bus). The easiest way to deal with this is to see such faulty behaviour as a trait of the physical bus.

```
proc Node(id: Sender, faulty: Bool) =
  allow({...},
    comm({encode|encode->Encode,
          decode|decode->Decode,
          is_idle|is_idle->Is_idle},
      CODEC(id) ||
      POC(id, 0, faulty) ||
      MAC(id, 0, false)
  ));

init
  allow({...},
    comm({...},
      Bus({}, {}, 0, 0, NONE) ||
      Node(S0, false) ||
      Node(S1, false) ||
      Node(S2, false)
  ));
```

Fig. 6. The mCRL2 code that implements the structure of Figures 4 and 5. Ellipses indicate omitted text.

The physical bus is modelled as a process that reads, per time slot, a signal from all connected nodes that are not mute and delivers the combination of all those signals back to all connected nodes that are not deaf.

```
proc Bus(r, w: Sender, s: Signal) =
  (r <= NODES + 1) -> (
    sum s': Signal . put'(r, s') .
      Bus(r + 1, w, if(r == mute, s, combine(s, s')))
  ) <> (
    (w <= NODES + 1) -> (
      get'(w, if(w == deaf, NONE, s)) . Bus(r, w + 1, s)
    ) <> (
      bus(s) . Bus(1, 1, NONE)
    )
  );
```

If a node is not sending, it is modelled as a node that is sending silence. The combination of a signal and silence is that signal. The combination of two signals is defined to be noise.

When a bit has been sent and received in this manner, the `bus` action marks the progress of time and the process repeats itself (it is part of the clock tick barrier).

In the model in appendix C, two variants of the bus are included. One of them reads in an arbitrary order, and one reads in a fixed order. The latter was used for verification purposes (because it is the more general model), but the model as listed above was used to generate the traces in section 5.1, as it is

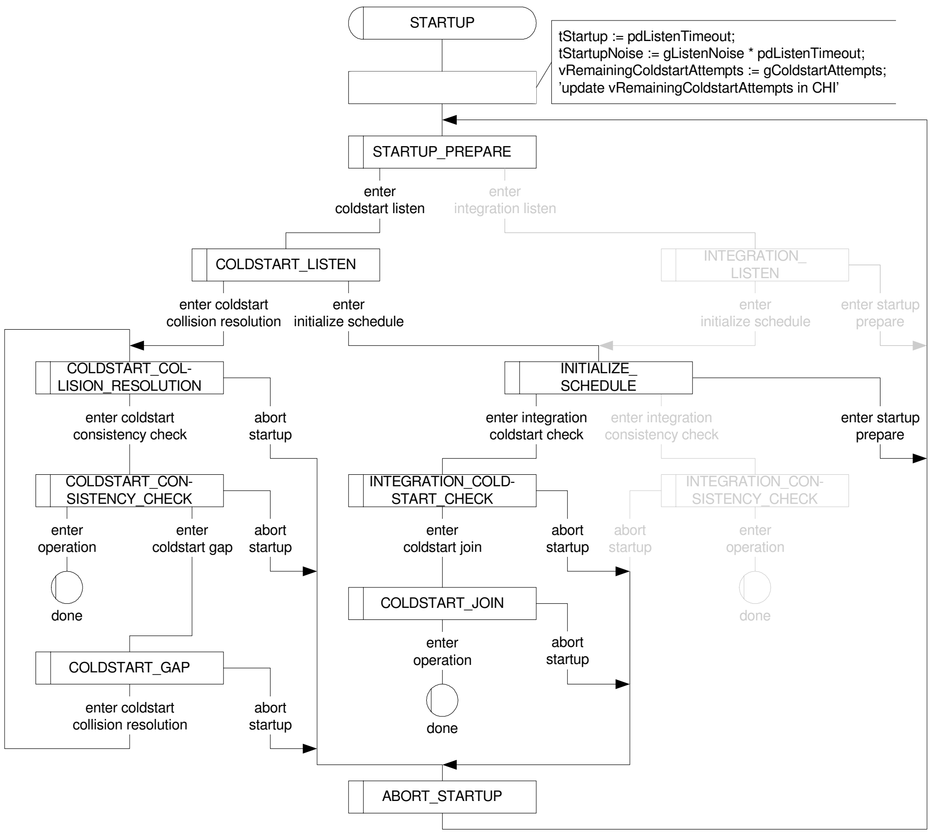
STARTUP
tStartup := pdListenTimeout;
tStartupNoise := gListenNoise * pdListenTimeout;
vRemainingColdstartAttempts := gColdstartAttempts;
'update vRemainingColdstartAttempts in CHI'
STARTUP_PREPARE
enter coldstart listen
enter integration listen
COLDSTART_LISTEN
INTEGRATION_ LISTEN
enter coldstart collision resolution
enter initialize schedule
enter initialize schedule
enter startup prepare
COLDSTART_COL-LISION_RESOLUTION
INITIALIZE_ SCHEDULE
enter coldstart consistency check
abort startup
enter integration coldstart check
enter integration consistency check
enter startup prepare
COLDSTART_CON-SISTENCY_CHECK
INTEGRATION_COLD-START_CHECK
INTEGRATION_CON-SISTENCY_CHECK
enter operation
enter coldstart gap
abort startup
enter coldstart join
abort startup
abort startup
enter operation
done
done
COLDSTART_JOIN
COLDSTART_GAP
enter operation
abort startup
done
enter coldstart collision resolution
abort startup
ABORT_STARTUP

Fig. 7. The POC startup phase, taken from [8]. Grey parts are not included in our model.

quicker. The reason the model that performs reads and writes in an arbitrary order is more time-consuming to generate is that it generates more states, and uses set operations where the ordered model needs only enumerated and integral types.

POC model

Because the FlexRay protocol specification is already quite formal (in the sense that a systematic notation is used to describe processes), we set out to create our model in a way that remains as close to the specification as possible. Because a large part of the specification is abstracted away, we cannot use an automated conversion to an mCRL2 model, but where our model attains the same degree of detail as the original specification, we attempt to make our translation as systematic as possible. We have taken the diagrams from Chapter 7.2 in [8] as the basis for our model. The semantics of these diagrams is given in [15].

To create a model that would not inevitably blow up to unreasonable proportions, we use a slightly simplified version of these semantics. We assume that the calculations in the diagrams do not take time; moving from one state to the next can then be seen as an instantaneous change. Secondly, we assume that signals do not take time to be delivered. In [15], event queues are used to capture the semantics of these phenomena. In a finite-state model, this would lead to a very large state space, as the state of the queues must be taken into account. Our simplifications eliminate the need for queues. The downside is that not all possible sequences of events that may occur in reality are modelled; for instance, a signal from another process never arrives when the receiving process is busy.

We note here that the FlexRay specification does not claim to use the precise SDL semantics (in fact, it claims that it may not do so). Given that the specification is at the detail level of a reference implementation, this is not surprising, as the use of event queues would be infeasible in a hardware implementation. The specification does however not give any guidelines on how to interpret the diagrams, which is why we have taken the official SDL semantics as a starting point.

Using the simplified semantics, the SDL diagrams in Chapter 7.2 of [8] were translated into mCRL2 code (using a slightly altered syntax, see appendix A). As an example, we show the part of the mCRL2 code that models the 'coldstart collision resolution' diagram in the specification (Figure 7–12 in [8]):

```
state ColdstartCollisionResolution(timer: Nat) =
(
  % Inputs from CODEC
  decode(CAS) . bit . AbortStartup()
+ sum id': Sender . decode(FRAME_HEADER(id')) .
                    hdr(id, id') . AbortStartup()
+ sum id': Sender . decode(FRAME(id')) .
                    ColdstartCollisionResolution()

  % Wait for four cycles
+ (timer < FRM_START(id) + CYCLE_length * 4 - 1) ->
  bit . ColdstartCollisionResolution(timer=timer + 1)
+ (timer >= FRM_START(id) + CYCLE_length * 4 - 1) ->
  bit . cs_cons . ColdstartConsistencyCheck(0)
)
```

Looking at SDL diagram 7–12, we see a single state in which the POC may respond to a number of different events:

– *SyncCalcResult.* This event is emitted by the clock synchronisation process, just before a new communication cycle starts. We assume a global clock with a resolution of one bit, of which the ticks are modelled by the `bit` actions. Rather than waiting for four *SyncCalcResult* events, we instead look at the global clock to decide when four cycles have passed.
– *header received on A/B.* This event is emitted by CODEC right after a frame header has been received. It is modelled using the `decode` action with a `FRAME_HEADER` parameter.

– *symbol decoded on A/B.* This event is emitted by CODEC right after a collision avoidance symbol or media test symbol is decoded. It is again modelled using the `decode` action, this time with a `CAS` parameter.

In the mCRL2 model, we need to absorb `decode(FRAME(id'))` events (not doing so would prevent the source from sending the event, thus potentially leading to deadlock states). In the semantics of SDL [15], this corresponds to discarding an event from the event queue when it cannot be processed.

To allow time to progress in a state, we must additionally allow the `bit` action to occur. The above piece of code implements a timeout of four cycles by letting time progress for four cycles, and then moving to the 'coldstart consistency check' state. If a CAS or frame header is received before the timeout occurs, the startup is aborted.

MAC

Media access control is modelled by the mCRL2 process below. It starts in inactive mode (not shown) and can then receive `macCAS`, `macStart` and `macStop` commands.

```
proc MAC(id: Sender, togo: Int, active: Bool) =
  macCAS . encode(CAS) .
    MAC(active=true, togo=FRM_START(id) + length(CAS))
+ macStart . MAC(active=true, togo=FRM_START(id))
+ macStop . wait . MAC(active=false)
+ active -> (
    (togo > 0) -> (
      wait . MAC(togo=togo - 1)
    ) <> (
      encode(FRAME_HEADER(id)) . MAC(togo=CYCLE_length)
    )
  ) <> wait . MAC();
```

When a `macCAS` command is received, it requests the CODEC to send a collision avoidance symbol and then switches to active mode. In the specification, MAC waits for one macrotick before sending the CAS, but it seems that the waiting for the macrotick is not intended as a delay, but merely as a check that the clock synchronisation process has started. Since we assume a global clock, we do not need to model this delay.

When a `macStart` command is received, the MAC proceeds to active mode, and when it receives a `macStop` command, it goes back to inactive mode. In active mode, MAC periodically requests CODEC to encode a frame.

CODEC

The CODEC is modelled as a process that either reads from or writes to the bus. When in reading mode, it processes bits it reads from the bus, and writes silence to the bus (*i.e.*, it does not write anything to the bus). When it is in writing mode, bits it reads from the bus are ignored, and it writes an encoding of the last requested symbol to the bus.

Remainder

It can be seen from the context diagrams in [8] that the POC process communicates with all other processes during the startup process. We covered MAC and CODEC, and we argue that we may safely omit models for the remaining processes.

By assuming a global clock, we can avoid modelling the macrotick generation, clock synchronisation startup and clock synchronisation processes. Furthermore we assume that nodes are never in coldstart inhibit mode. This eliminates the influence of the controller host interface, so we can also omit a model for that process. During startup, only the clock synchronisation depends on events generated by frame and symbol processing, so this process is also ignored.

5 Verification

From the viewpoint of the POC, the faults mentioned in Section 4.2 can be seen as instances of a few general problems that may occur. Either a node is not able to send anything, a node is not able to receive anything, or the bus misbehaves in such a way that symbols are not always transmitted correctly. Only the periodic resetting of a node requires the node to display slightly more complicated behaviour.

Since for the POC a noisy signal is observably equal to no signal at all (the CODEC simply does not generate events), we model a limited set of scenarios. Each of the descriptions below describes two scenarios: one in which the node with the lowest identifier is the faulty node, and one in which another node is faulty. This is necessary because the protocol relies on a leader election mechanism that is not quite symmetric: although the process descriptions for startup are the same for every node, the leader that will be elected depends on the configuration of the nodes. The candidate configured with the lowest identifier will be elected as leader. At least one failure scenario (*viz.* the resetting node scenario below) is known [23] that is only possible if the node with the lowest identifier is the faulty node.

In this manner, the following categories of scenarios are modelled.

Two nodes A faulty node does not switch on at all, so effectively there are only two nodes present in the network.

Silent node A faulty node is not able to send anything. Although we do model this separately, we note that this scenario is equivalent to the two-node scenario if we are not interested in the behaviour of the faulty node. We include this scenario because it shows that the silent node is still able to integrate into the communication correctly, albeit in a read-only mode.

Deaf node A faulty node does not receive anything.

Resetting node A node resets itself periodically.

Noisy channel Signals sent by nodes are corrupted on the channel. We use a noise model that consisting of a burst length and a maximum backoff time. The burst length determines the maximum number of sequential bits that are corrupted, the maximum backoff time determines the maximum number of sequential bits that pass through the channel unaltered.
Due to practical limitations, we were only able to model this scenario in a two-node scenario.

For each of these scenarios, we check that the correctly functioning nodes start up. We do this by checking three properties. The first is absense of deadlock; our model is constructed in such a way that we do not expect a deadlock to occur (we always allow time to progress, so a deadlock would indicate an error in the model).

Absence of deadlock is checked while generating the statespace. The other two properties are formulated in the first order modal µ-calculus (see, *e.g.*, [13]). For brevity, we use mathematical syntax rather than concrete mCRL2 µ-calculus syntax, and extra statements to help the mCRL2 toolset (*e.g.*, to prevent quantifiers from being expanded forever) are left out. The concrete formulae are given in Appendix B. It is important to note that these formulae only represent the intended properties correctly if the system they are checked on is deadlock free, as otherwise the $[\mathbf{true}]\varphi$ subformulae might trivially hold.

The second property asserts that eventually all correctly functioning nodes enter normal operation exactly once, an event that is flagged by the *enter_operation* action. It is expressed by the formula in Figure 8, in which N is the total number of nodes and C is the set of correctly functioning nodes.

$$
\begin{array}{l}
\mu X(r : 2^{\mathbb{N}} = C)\,.\\
(\\
\qquad r \neq \emptyset\\
\quad \wedge\ (\forall i : \mathbb{N}\,.\,(i \in r \Rightarrow [\mathit{enter\_operation}(i)]X(r \setminus \{i\})))\\
\quad \wedge\ [\neg\exists i : \mathbb{N}\,.\,\mathit{enter\_operation}(i)]X(r)\\
)\\
\vee\\
(\\
\qquad r = \emptyset\\
\quad \wedge\ \nu Y\,.\,[\exists i : \mathbb{N}\,.\,i \in C \wedge \mathit{enter\_operation}(i)]\mathbf{false}\\
\qquad\qquad \wedge[\mathbf{true}]Y\\
)
\end{array}
$$

Fig. 8. Eventual startup, expressed as a first order µ-calculus formula.

The last property says that eventually all correctly functioning nodes will keep receiving each others messages. Even though our model is not intended to model the ongoing traffic after startup, we have constructed our model in such a way that this property should hold. If this property does not hold, then it

$$
\begin{array}{l}
\mu X \,.\, [\mathbf{true}]X \vee \\
(\\
\quad \nu Y(r : 2^{\mathbb{N}} = C, s : \mathit{Symbol} = \texttt{FRAME\_HEADER}(\mathit{head}(C)))\,. \\
\quad (\\
\qquad \mu Z\,. \\
\qquad ((\\
\qquad\qquad r \neq \emptyset \\
\qquad\quad \wedge\ (\forall i : \mathbb{N}, s' : \mathit{Symbol}\,. \\
\qquad\qquad\quad [\mathit{Decode}(i, s')](\,(i \in C \wedge s = s' \wedge i \in r \wedge Y(r \setminus \{i\}, s)) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (i \notin C \wedge Z)) \\
\qquad\qquad) \\
\qquad\quad \wedge\ [\neg \exists i : \mathbb{N}, s' : \mathit{Symbol}\,.\, \mathit{Decode}(i, s')]Z \\
\qquad\ ) \vee (\\
\qquad\qquad r = \emptyset \\
\qquad\quad \wedge\ Y(C \setminus \mathit{nextsender}(s), \mathit{nextsymbol}(s)) \\
\qquad)) \;) \;)
\end{array}
$$

Fig. 9. Eventual communication, expressed as a first order µ-calculus formula.

is likely that the nodes did not synchronise correctly. The formula in Figure 9 states this property formally.

Verification of these properties is done by linearising the mCRL2 specification and combining it with the formulae to form parameterised Boolean equation systems. These are instantiated to Boolean equation systems, which are in turn reduced modulo stuttering equivalence on parity games. The resulting smaller equation systems are then solved. A description of this procedure can be found in [10]. We use the July 2011 release of the mCRL2 toolset.

We note that it is also possible to check eventual startup and eventual communication manually. By hiding all actions but *enter_operation* and then reducing the statespace using branching bisimulation, the first property can be checked. The second property can be checked manually by hiding all but *Decode*, reducing the statespace using branching bisimulation and then manually inspecting all strongly connected components.

5.1 Results

No faulty nodes The statespace is deadlock free, and both properties hold on the system.

Two nodes The statespace is deadlock free, and both properties hold on the system.

Silent node The statespace is deadlock free, and both properties hold on the system.
Manually inspecting the branching-bisimulation reduced statespace reveals that the failing node can in this case enter normal operation using the wrong schedule (see Figure 10). The clock synchronisation process will allow this scenario, and frame and symbol processing will also not detect the mistake

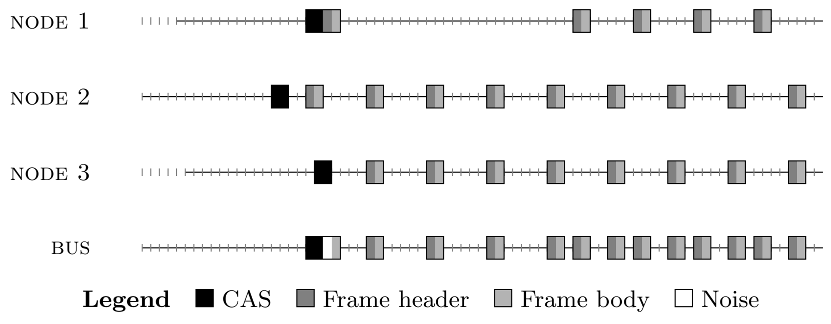

Fig. 10. Node 2 is mute, and can therefore start operation out of sync.

while the startup protocol has not finished. The mistake is harmless, however, because the silent node cannot disturb ongoing communication. As soon as normal operation is entered, the clock correction process or the frame and symbol processing process of the faulty node will notice the error. A next attempt to integrate will succeed, because there is then already ongoing traffic.

Notice that the above scenario is possible because a process is not able to read while writing (as can be seen from Figure 3-18 in [8]).

Deaf node The statespace is deadlock free, but neither of the µ-calculus properties hold.

In case of a deaf node, there is a possibility of the network not starting at all. Figure 11 shows such a scenario.

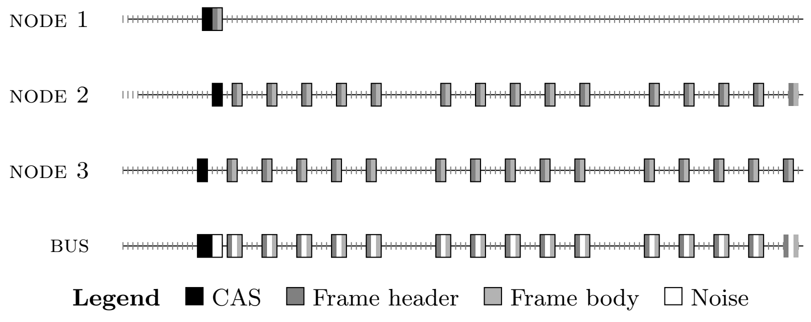

Fig. 11. Node 2 is deaf, and prevents other nodes from starting up.

The deaf node can choose to align its frames with those of another startup node, causing only the headers of the other node to be readable on the bus. The non-faulty node that is broadcasting startup frames will not detect that every sent frame is corrupted by the faulty node. Because the non-faulty

node's frame headers are untouched, all other nodes will wait until it gives up after the maximum number of startup attempts.
When there are more than three coldstart nodes in the system, the remaining nodes will be able to start normally after that. In Figure 11 however, there are only three, and the remaining coldstart node cannot start the system by itself. In this case, the entire network fails to start.
This scenario again is due to the fact that the CODEC cannot read and write simultaneously.

Resetting node The statespace is deadlock free, but neither of the µ-calculus properties hold.
Although this scenario was already known (it was described in [24]), the emergence of the trace in Figure 12 gives us confidence that our model is correct. The trace shows that the leading node may cause startup of the network to fail by resetting itself every time it has sent a frame. In fact, it would just have to send the frame header, but the way we modelled our reset behaviour does not allow this.

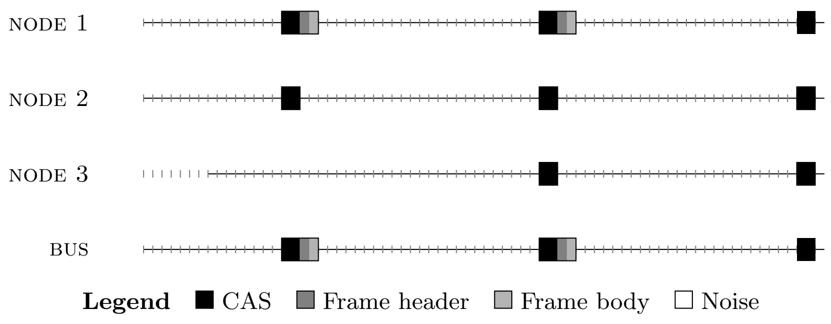

Fig. 12. Node 1 resets periodically, preventing the other nodes from starting up.

It should be noted that in this scenario it is required that node 1 be the faulty node, which is not necessary in the scenarios for deaf and mute nodes.

Noisy channel The results depend very much on the parameters of the noise model.
For an arbitrary noise pattern, it is obvious that the system will not start up. The channel could simply decide to corrupt all traffic going through it. The noise model we chose guarantees that some information will come through. Checking exactly for which values of maximum burst size and maximum backoff period the system starts correctly is too big a task, but simply trying a few settings soon gives an idea of how robust the system is. We made the following observation.
If there is noise on the channel for too long while nodes are trying to commence the startup procedure, then obviously startup may fail. The interesting scenarios are those in which some information can be communicated.

However, if the minimum backoff time is less than the time needed for fault-free startup, then one of the sync frames of the leading coldstart nodes can always be corrupted, causing either the schedule initialisation or the consistency check of the other nodes to fail.
If the presence of noise is the only anomaly in the system, then the minimum backoff time being at least the time required for fault-free startup is enough to guarantee that the system will come up.

Techniques

We created a model that is very close to the original specification. As a consequence the model is rather complex, and its state space is large for realistic values of the various parameters.

The results in this paper are obtained by explicitly generating the statespaces of systems in which these parameters have been set to (unrealistically) small values. The resulting statespaces are small enough to minimise modulo branching bisimulation, and the reduced statespaces are in turn small enough to check manually.

A natural choice when dealing with extremely large systems is to use symbolic model checking. In our case, however, the bottleneck does not seem to be in the memory needed to store the statespace, but in the time needed to generate it; a problem that cannot be overcome by using for instance the symbolic model checker *LTSMIN*[2]. Using this tool we tried to check that the *startup_failed* action does not occur in a fault-free setting. However, after 9 days of computing, only 2.2 billion states have been checked. Attempts to speed up the state space generation by reducing the size of the process equations did not lead to better results, and different search strategies (BFS, chaining) seem to perform equally poor.

The µ-calculus properties were checked by using parity game reduction and parity game solvers, as described in [10]. Although the parity game reduction did speed up the solving process significantly, the real bottleneck was generating the game (or, equivalently, boolean equation system) itself, taking over three days for the second µ-calculus property for the scenario with a resetting leader node.

Specification

As was already mentioned in [23], the absense of a fault hypothesis makes it difficult to assess whether the specification specifies a correctly operating system.

Modelling the specification is difficult, because the specification is given only in a high level of detail. This is ideal for implementing the system, because the mapping to an implementation is very straightforward. However, it makes it virtually impossible to abstract away implementation details: due to implementation considerations, it seems, the specification lacks a hierarchical structure; functionality like clock synchronisation is spread out over multiple components. In this sense, the specification does not so much specify a general networking protocol, but rather a reference implementation of one. A lack of explanation on a more abstract level makes it extremely difficult to understand what the exact functionality of the protocol is, and makes it hard to assess whether simplifications of a model are valid abstractions.

6 Conclusion

We have modelled a large part of the FlexRay startup protocol using a discrete time model. We presented a notion of correct operation of the startup protocol, and mechanically verified that in a number of fault scenarios the startup protocol operates correctly. The model revealed two scenarios in which a network does not start up; one of these scenarios was previously unknown.

References

1. D. Barsotti, L.P. Nieto, and A. Tiu. Verification of Clock Synchronization Algorithms: Experiments on a combination of deductive tools. *Formal Aspects of Computing*, 19(3):321–341, 2007.
2. S. Blom, J. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.
3. J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, and M. Spichkova. On the correctness of upper layers of automotive systems. *Formal Aspects of Computing*, 20(6):637–662, 2008.
4. J. Botaschanjan, A. Gruler, A. Harhurin, L. Kof, M. Spichkova, and D. Trachtenherz. Towards modularized verification of distributed time-triggered systems. *Lecture Notes in Computer Science*, 4085:163, 2006.
5. M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Verlag, 2001.
6. FlexRay consortium. *FlexRay Preliminary Central Bus Guardian Specification v2.0.9*, 2005.
7. FlexRay consortium. *FlexRay Preliminary Node Local Bus Guardian Specification v2.0.9*, 2005.
8. FlexRay consortium. *FlexRay Protocol Specification v2.1 rev. A*, 2005.
9. FlexRay consortium. *FlexRay Requirements Specification v2.1*, 2005.
10. S. Cranen, J.J.A. Keiren, and T.A.C. Willemse. Stuttering mostly speeds up solving parity games. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 207–221. Springer Berlin / Heidelberg, 2011.
11. P. Fontaine, K. Gupta, J.Y. Marion, S. Merz, L.P. Nieto, and A. Tiu. Towards a combination of heterogeneous deductive tools for system verification: A case study on clock synchronization. In *APPSEM Workshop*. Citeseer, 2005.
12. J.F. Groote, J.J.A. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T.A.C. Willemse, et al. The mCRL2 toolset. In *WASDeTT*. Citeseer, 2008.
13. J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. In *AMAST*, pages 74–90, 1998.

14. J.F. Groote, A. Mathijssen, M.A. Reniers, Y.S. Usenko, and M. van Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process algebra for parallel and distributed processing.* CRC Press, 2009.
15. ITU-T. Recommendation Z.100: Specification and Description Language (SDL), 1999.
16. C. Kühnel and M. Spichkova. FlexRay und FTCom: Formale Spezifikation in FOCUS. Technical report, Technische Universität München, 2006.
17. C. Kühnel and M. Spichkova. Upcoming Automotive Standards for Fault-Tolerant Communication: FlexRay and OSEKTime FTCom. In *EFTS 2006 International Workshop on Engineering of Fault Tolerant Systems. Universite du Luxembourg, CSC: Computer Science and Communication*, 2006.
18. J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, page 88. ACM, 1984.
19. J. Malinský. The application of the timed automata for flexray start-up testing. In *Proceedings of the 13th Workshop on ADC Modelling and Testing (TC4), Florence, ITALY*, 2008.
20. J. Malinský and J. Novák. Verification of flexray start-up mechanism by timed automata. *Metrology and Measurement Systems*, XVII(3):461–480, 2010.
21. T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.
22. M. Spichkova. FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study. Technical report, Technische Universität München, 2006.
23. W. Steiner. An assessment of FlexRay 2.0 (safety aspects). Technical Report 45/2005, Technische Universität Wien, 2005.
24. W. Steiner. Model-checking studies of the flexray startup algorithm. Technical report, Technische Universität Wien, 2005.
25. B. Zhang. On the Formal Verification of the FlexRay Communication Protocol. *Automatic Verification of Critical Systems (AVoCS)*, pages 184–189, 2006.
26. B. Zhang. Specifying and Verifying Timing Properties of a Time-triggered Protocol for In-vehicle Communication. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 467–472. IEEE Computer Society, 2008.

A Modified mCRL2 syntax

In order to create a more structured specification, an extension of the mCRL2 syntax was used. This syntax is not meant to provide a semantic extension of the language, but merely to syntactically abbreviate some common constructs so as to remove clutter. We therefore merely give an explanation of how to read the syntax, but do not recommend using it in any other setting.

The syntax is best explained using an example. Consider the following specification:

```
proc X = nested(n: Nat) {
  initial state A =
    a(n) . B(0);
  state B(m: Nat) =
    m < 10 -> b . B(m + 1)
  + m == 10 -> b . X(n + 1)
}
```

This specification is translated to the following, pure mCRL2 specification.

```
proc X(n: Nat) = X'A(n: Nat);
proc X'A(n: Nat) = a(n) . X'B(0);
proc X'B(n: Nat, m: Nat) =
  m < 10 -> b . X'B(n, m + 1)
+ m == 10 -> b . X(n + 1);
```

The intuition is that every ***nested*** block defines some parameters that are shared by the states defined in it. The states are mCRL2 process specifications, and can therefore again be defined in terms of nested blocks.

A state may add parameters to the list provided in its associated ***nested*** statement. Duplicate parameter names are not allowed. When a process or state name is used within a state, a name lookup is done first in the scope of the current ***nested*** block, then in the one above that etc.

Obviously, name clashes are a big problem in this construction. The models presented in this document therefore use unique variable and process names in such a way that the translation scheme does not cause any ambiguity.

B Concrete µ-calculus formulae

```
      Bus || Noise || Resetter ||
      ${TYPE1}Node(1) || ${TYPE2}Node(2) || ${TYPE3}Node(3)
  ));

% Mappings for the mu-calculus formulae.
map nextin: Nat # List(Nat) -> Sender;
    inornext: Nat # List(Nat) -> Sender;
    finite: Symbol -> Bool;
    nextsym: Symbol # List(Nat) -> Symbol;
    correct_nodes: Sender -> List(Sender);
    CORRECT_NODES: List(Sender);
    remove: List(Sender) # Sender -> List(Sender);
var s, s2: Sender;
    l: List(Sender);
    sym: Symbol;
eqn nextin(s, l) = inornext((s + 1) mod (NODES + 1), l);
    l != [] && s in l -> inornext(s, l) = s;
    l != [] && !(s in l) -> inornext(s, l) = nextin(s, l);
    finite(sym) = (isHeader(sym) || isFrame(sym)) => s(sym) <= NODES;
    nextsym(FRAME_HEADER(s), l) = FRAME(s);
    nextsym(FRAME(s), l) = FRAME_HEADER(nextin(s, l));
    s <= NODES -> correct_nodes(s) =
      if(!(s in {reset_node, deaf_node, mute_node}),
         s |> correct_nodes(s + 1),
         correct_nodes(s + 1));
    s > NODES -> correct_nodes(s) = [];
    CORRECT_NODES = correct_nodes(0);
    remove(s2 |> l, s) = if(s2 == s, l, s2 |> remove(l, s));
    remove([], s) = [];
```

B.1 Eventually all nodes will start up

```
mu X(s: List(Sender) = CORRECT_NODES) .
  (val(s != [])
   && (forall i: Sender .
        (val(i in s) => ([enter_operation(i)]X(remove(s, i)))))
   && [!exists i: Sender .
        (val(i <= NODES) && enter_operation(i))]X(s)
  )
||(val(s == []) &&
   nu Y .
      [exists i: Sender .
        val(i <= NODES && i in CORRECT_NODES) &&
        enter_operation(i)]false
   && [true]Y
  )
```

B.2 Eventually all nodes will communicate

```
mu X . [true]X || (
  nu Y(togo: List(Sender) = tail(CORRECT_NODES),
       s: Symbol = FRAME_HEADER(head(CORRECT_NODES))) .
    mu Z .
    (val(togo != [])
     && (forall i: Sender .
          (val(i in CORRECT_NODES) &&
            (([Decode(i, s)]
              (
                Y(remove(togo, i), s) &&
                val(i in togo)
              )
             ) &&
             forall s': Symbol .
              (val(finite(s') && s != s') =>
               [Decode(i, s')]false
              )
            )
          )
        || (!val(i in CORRECT_NODES) &&
            (forall s': Symbol .
              (val(finite(s')) => [Decode(i, s')]Z))
         ))
     && ([!exists i: Sender, s': Symbol .
            (val(i <= NODES && finite(s')) &&
            Decode(i, s'))]Z)
    )
  ||(val(togo == [])
     && Y(remove(CORRECT_NODES, s(nextsym(s, CORRECT_NODES))),
          nextsym(s, CORRECT_NODES))
    )
)
```

C mCRL2 specification

Below follows the integral mCRL2 specification for the FlexRay startup protocol. Some values in the model are parameterised; they are included in the specification using environment variable syntax (‘`${NAME}`’). Replacing these values by a concrete value of the appropriate sort yields an instance that, after preprocessing, can be parsed by the mCRL2 toolset.

```
% Additional rules for if, to reduce the size of the linearised process.
var a, b: Bool;
eqn if(a, false, true) = !a;
    if(a, b, true) = a => b;
    if(a, b, false) = a && b;
    if(a, true, b) = a || b;

% User defined sorts
sort Sender = Nat;
     Signal = struct NONE?isNone
                   | NOISE?isNoise
                   | DATA_BIT(s: Sender)?isData
                   | CAS_BIT?isCAS
                   | FIRST_HEADER_BIT(s: Sender)?isHeader
                   | FIRST_BODY_BIT(s: Sender)?isBody;
     Symbol = struct CHIRP
                   | IDLE_END
                   | CAS
                   | FRAME(s: Sender)?isFrame
                   | FRAME_HEADER(s: Sender)?isHeader
                   | NOTHING;

% Model parameters
map deaf_node,
    mute_node,
    reset_node: Sender;
    MIN_DELAY,
    MAX_DELAY: Nat;
    CHIRP_length: Nat;
    NIT_length,
    CYCLE_length: Pos;
    length: Symbol -> Nat;
    noise_max_burst,
    noise_min_backoff,
    noise_max_backoff: Nat;
eqn deaf_node = ${DEAF};       % Which is the deaf node (0 to disable)
    mute_node = ${MUTE};       % Which is the mute node (0 to disable)
    reset_node = ${RESET};     % Which is the resetting node (0 to disable)
    MIN_DELAY = ${MIN_DELAY}; % Min and max delay of delayed nodes
    MAX_DELAY = ${MAX_DELAY};
    CHIRP_length = ${CHIRP};   % Lengths (in bits) of CHIRP, NIT, CAS,
    NIT_length = ${NIT};       %   frame header and frame
    length(CAS) = ${CAS};
    length(FRAME_HEADER(id)) = ${HDR};
    length(FRAME(id)) = ${HDR} + ${PAYLOAD};
    length(NOTHING) = 0;
    noise_max_burst = ${NOISE_MAX_BURST};      % Noise model
    noise_min_backoff = ${NOISE_MIN_BACKOFF};
    noise_max_backoff = ${NOISE_MAX_BACKOFF};

% Model constants and helper functions
map NODES: Pos;
    FRM_START: Sender -> Nat;
    symbol: Signal -> Symbol;
    signal: Symbol -> Signal;
    noise_id: Nat;
var id: Sender;
eqn NODES = 3;
    noise_id = NODES + 1;
    CYCLE_length = NODES * length(FRAME(1)) + NIT_length;
    FRM_START(id) = Int2Nat(id - 1) * length(FRAME(1));
    symbol(CAS_BIT) = CAS;
    symbol(FIRST_HEADER_BIT(id)) = FRAME_HEADER(id);
    symbol(FIRST_BODY_BIT(id)) = FRAME(id);
    symbol(NOISE) = NOTHING;
    symbol(NONE) = NOTHING;
    signal(CAS) = CAS_BIT;
    signal(FRAME_HEADER(id)) = FIRST_HEADER_BIT(id);
    signal(FRAME(id)) = FIRST_BODY_BIT(id);

act % Bus <--> Node
    get, get', put, put', Get, Put: Sender # Signal;
    % Barrier
    bus: Signal;
    bit, wait;
    Encode: Symbol;
    % MAC <--> CODEC
    encode: Symbol;
    macCAS, macStart, macStop;
    % CODEC <--> POC
    decode, Decode: Sender # Symbol;
    is_idle, Is_idle: Bool;
    % Events (for debugging and visualisation purposes)
    enter_operation, init_sched, startup_failed, attempt_startup: Sender;
    abort: Nat;
    reset, Reset: Sender;

% The noise model is parameterised with three values: the maximum length of a
% noise burst, during which the noise process generates noise. The minimum
% backoff and maximum backoff bound the period in between bursts.
proc
```

```
  Noise = NoiseP(0, 0);
  NoiseP(burst, backoff: Nat) =
      ((backoff >= noise_min_backoff || burst > 0) && burst < noise_max_burst)
      -> put(noise_id, NOISE) .
         NoiseP(burst=burst + 1, backoff=0)
  + (backoff <= noise_max_backoff)
      -> put(noise_id, NONE) .
         NoiseP(burst=0, backoff=backoff + 1)
  + (burst == noise_max_burst)
      -> put(noise_id, NONE) .
         NoiseP(burst=0, backoff=1)
  + sum s: Signal . get(noise_id, s) . NoiseP();

% The bus process is modelled as a discrete-time process that reads from NODES
% processes and writes the combined signal (two senders results in noise) back
% to those processes.
map combine: Signal # Signal -> Signal;
var a, b: Signal;
eqn combine(NONE, b) = b;
    combine(a, NONE) = a;
    !(isNone(a) || isNone(b)) -> combine(a, b) = NOISE;
% Two variants are given: one that reads and writes in arbitrary order every bit
% period...
proc
  UnorderedBus(r, w: Set(Sender), nr, nw: Nat, s: Signal) =
  (nr <= NODES) -> (
    sum i: Sender, s': Signal . (!(i in r)) -> put'(i, s') .
      UnorderedBus(r + {i}, w, nr + 1, nw,
                   if(i == mute_node, s, combine(s, s')))
  ) <> (
    (nw <= NODES) -> (
      sum i: Sender . (!(i in w)) -> get'(i, if(i == deaf_node, NONE, s)) .
        UnorderedBus(r, w + {i}, nr, nw + 1, s)
    ) <> (
      bus(s) . UnorderedBus({}, {}, 0, 0, NONE)
    )
  );
% ... and one that does it in a fixed order.
  OrderedBus(r, w: Sender, s: Signal) =
  (r <= NODES + 1) -> (
    sum s': Signal . put'(r, s') .
      OrderedBus(r + 1, w, if(r == mute_node, s, combine(s, s')))
  ) <> (
    (w <= NODES + 1) -> (
      get'(w, if(w == deaf_node, NONE, s)) . OrderedBus(r, w + 1, s)
    ) <> (
      bus(s) . OrderedBus(1, 1, NONE)
    )
  );
% It should not make a difference which is used, so the UnorderedBus is only
% there fore debugging purposes.
  Bus = OrderedBus(1, 1, NONE);

% The CODEC process models the SDL process of the same name. We abstract away
% from the process of bitstrobing. What remains is the mutually exclusive
% sending and receiving of bit patterns.
proc CODEC = nested(id: Sender)
{
  initial state Init() = Receive(0, 0, 0, NONE);

  state Receive(lastsender: Sender, idle, data: Nat, sig: Signal) =
    is_idle(idle >= CHIRP_length) . Receive()
  + bit . Receive()
  + put(id, NONE) . Receive()
  + sum S: Symbol . encode(S) . Send(signal(S), 0, length(S))
  + sum S: Signal . get(id, S) .
    (
      isNone(S) -> Receive(0, if(isNone(sig),
                          min(CHIRP_length, idle + 1), 1), 0, S)
    + isNoise(S) -> Receive(0, 0, 0, S)
    + isData(S) -> Announce(s(S), 0, if(lastsender == s(S), data+1, 0), sig)
    + isCAS(S) -> Announce(s(S), 0, if(isCAS(sig), data + 1, 1), S)
    + isHeader(S) || isBody(S) -> Announce(s(S), 0, 1, S)
    );

  state Announce(lastsender: Sender, idle, data: Nat, sig: Signal) =
    is_idle(false) . Announce()
  + (data > 0 && data == length(symbol(sig)))
      -> decode(id, symbol(sig)) .
         Receive(sig=if(isHeader(sig), FIRST_BODY_BIT(s(sig)), sig))
      <> Receive();

  state Send(sig: Signal, sent, len: Nat) =
    is_idle(false) . Send()
  + bit . Send()
  + (sent == 0) -> put(id, sig) . Send(sent = 1)
  + (sent > 0 && sent < len)
      -> put(id, if(isCAS(sig), sig, DATA_BIT(id))) .
         Send(sent = sent + 1)
  + (sent == len && isHeader(sig))
      -> put(id, DATA_BIT(id)) .
         Send(DATA_BIT(id), sent + 1, length(FRAME(id)))
  + (sent == len && !isHeader(sig))
      -> sum S: Signal . get(id, S) . Receive(0, 0, 0, NONE)
      <> sum S: Signal . get(id, S) . Send();
};

% The MAC process models the SDL process of the same name, and decides when the
% CODEC process should send data. When it is active, it periodically sends a
% (startup) frame. If activation is done by a macCAS action, a CAS symbol is
% sent before activation. The MAC process is deactivated by a macStop action.
proc MAC(id: Sender, togo: Int, active: Bool) =
  macCAS . encode(CAS) . MAC(active=true, togo=FRM_START(id) + length(CAS) - 1)
+ macStart . MAC(active=true, togo=FRM_START(id))
+ macStop . MAC(active=false)
+ active -> (
    (togo > 0) -> (
      wait . MAC(togo=togo - 1)
    ) <> (
      encode(FRAME_HEADER(id)) . MAC(togo=CYCLE_length - 1)
    )
  ) <> wait . MAC();

% The DelayedController models a normal POC process that starts after MIN_DELAY
% and before MAX_DELAY clock ticks.
proc DelayedController = nested(id: Sender)
{
  initial state Off = Wait(0);

  state Wait(counter: Nat) =
    decode(id, CAS) . Wait()
  + sum id': Sender .
      decode(id, FRAME_HEADER(id')) .
      Wait()
  + sum id': Sender .
      decode(id, FRAME(id')) .
      Wait()

  + (counter < MAX_DELAY) -> bit . Wait(counter=counter + 1)
  + (counter >= MIN_DELAY) -> Controller(id, 0);
};

% The Controller process models the startup behaviour of the POC SDL process.
% Its states correspond directly to the SDL states defined in the FlexRay
% specification.
proc Controller = nested(id: Sender, attempts: Nat)
{
  % StartupPrepare is called every time a startup attempt is initiated. If the
  % number of startup attempts exceeds some maximum (in our case, we choose this
  % maximum to be 3), then startup is aborted altogether.
  initial state StartupPrepare =
    (attempts < 3) -> attempt_startup(id) . ColdstartListen(0, 0)
                   <> startup_failed(id) . Failed(id);

  % In the ColdStartListen state, the node waits for communication on the bus.
  % If no communication is detected, a coldstart is attempted by sending a CAS
  % symbol (by activating the MAC through a macCAS action).
  state ColdstartListen(tStartup, tStartupNoise: Nat) =
  (
    decode(id, CAS) . ColdstartListen(tStartupNoise=0)
```

```
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    ColdstartListen(tStartupNoise=0)
+ sum id': Sender .
    decode(id, FRAME(id')) .
    bit .
    init_sched(id) .
    InitialiseSchedule(0, id')
+ reset(id) . AbortStartup(attempts=0)

+ bit .
  (
    (tStartup >= 2 * CYCLE_length - 1 ||
     tStartupNoise >= 4 * CYCLE_length - 1)
    -> ( is_idle(true) .
         macCAS .
         abort(id) .
         ColdstartCollisionResolution(attempts=attempts+1, timer=-length(CAS))
       + is_idle(false) .
         ColdstartListen(tStartup=0,
                         tStartupNoise=tStartupNoise + 1)
       )
  + (tStartup < 2 * CYCLE_length - 1 &&
     tStartupNoise < 4 * CYCLE_length - 1)
    -> ( is_idle(true) .
         ColdstartListen(tStartup=tStartup + 1,
                         tStartupNoise=tStartupNoise + 1)
       + is_idle(false) .
         ColdstartListen(tStartup=0,
                         tStartupNoise=tStartupNoise + 1)
       )
  )
);

% In InitialiseSchedule, the node waits for the clock synchronisation process
% to finish its initialisation. In this model, we abstracted away from clock
% synchronisation, and therefore this state can be left as soon as the second
% frame of the sender that this node is synchronising on (syncon) is received.
state InitialiseSchedule(timer: Nat, syncon: Sender) =
(
  decode(id, CAS) . InitialiseSchedule()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    InitialiseSchedule()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    (
      (id' == syncon)
      -> ((timer == CYCLE_length - 1)
          -> bit .
```

```
              IntegrationColdstartCheck(
                Int2Nat(FRM_START(syncon) + length(FRAME(id'))), syncon,
                false, false, false, false)
           <> AbortStartup()
          )
       <> InitialiseSchedule()
    )
+ reset(id) . AbortStartup(attempts=0)

+ bit .
  (timer > CYCLE_length)
  -> AbortStartup()
  <> InitialiseSchedule(timer=timer + 1)
);

% Once a CAS symbol was sent, the ColdStartCollisionResolution state is
% entered to see if any other node is taking on the role of leading coldstart
% node. If not, then this node is the leading coldstart node, and the
% ColdstartConsistencyCheck state is entered. Otherwise, startup is aborted.
state ColdstartCollisionResolution(timer: Int) =
(
  decode(id, CAS) . AbortStartup()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    AbortStartup()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    ColdstartCollisionResolution()
+ reset(id) . AbortStartup(attempts=0)

+ bit .
  (timer < CYCLE_length * 4)
  -> ColdstartCollisionResolution(timer=timer + 1)
  <> ColdstartConsistencyCheck(0, false, false)
);

% During the ColdstartConsistencyCheck, the node checks that during two
% consecutive cycles a frame is decoded. If no-one followed, then it is
% assumed that the other nodes were not actve yet, and the ColdStartGap state
% is entered. If in the first cycle a frame is decoded, but not in the second,
% then it is assumed that something went wrong, and the startup attempt is
% simply aborted. If all went well, normal operation is entered.
state ColdstartConsistencyCheck(timer: Nat, ok1, ok2: Bool) =
(
  decode(id, CAS) .  ColdstartConsistencyCheck()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    ColdstartConsistencyCheck()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    ColdstartConsistencyCheck(ok1=true, ok2=timer >= CYCLE_length)
+ reset(id) . AbortStartup(attempts=0)

+ (timer < CYCLE_length * 1 - 1)
  -> bit . ColdstartConsistencyCheck(timer=timer + 1)
+ (timer < CYCLE_length * 2 - 1 && timer >= CYCLE_length * 1 - 1)
  -> (ok1 -> bit . ColdstartConsistencyCheck(timer=timer + 1)
          <> ((attempts < 3) -> macStop .
                                ColdstartGap(attempts=attempts + 1, timer=0)
                             <> bit . AbortStartup()))
+ (timer >= CYCLE_length * 2 - 1)
  -> (ok1 -> bit . enter_operation(id) . NormalOperation()
          <> AbortStartup())
);

% During the ColdStartGap, no frames are sent for one cycle. After this cycle,
% frame sending is recommenced, so that other nodes may follow. Entering the
% ColdStartGap state counts as a new coldstart attempt, and can therefore only
% be done a limited number of times.
state ColdstartGap(timer: Nat) =
(
  decode(id, CAS) .  AbortStartup()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    AbortStartup()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    ColdstartGap()
+ reset(id) . AbortStartup(attempts=0)

+ (timer < CYCLE_length * 1)
  -> bit . ColdstartGap(timer + 1)
  <> macStart . bit .
     attempt_startup(id) . ColdstartCollisionResolution(1)
);

% The IntegrationColdstartCheck is performed when a node is not the leading
% coldstart node. It checks that either the leading coldstart node is still
% sending frames as expected, or otherwise at least two other nodes are
% sending frames. If this is not the case, then the coldstart attempt is
% aborted.
state IntegrationColdstartCheck(timer: Nat, syncon: Sender,
                                seen1, seen2, ok1, ok2: Bool) =
(
  decode(id, CAS) .  IntegrationColdstartCheck()
+ reset(id) . AbortStartup(attempts=0)
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    IntegrationColdstartCheck()
+ sum id': Sender .
```

```
    decode(id, FRAME(id')) .
    IntegrationColdstartCheck(
      seen1=timer >= 1 * CYCLE_length,
      ok1=seen1 || (id'==syncon && timer >= 1 * CYCLE_length),
      seen2=timer >= 2 * CYCLE_length,
      ok2=seen2 || (id'==syncon && timer >= 2 * CYCLE_length))

+ (timer < 1 * CYCLE_length - 1)
  -> bit . IntegrationColdstartCheck(timer=timer + 1)
+ (timer < 2 * CYCLE_length - 1 && timer >= 1 * CYCLE_length - 1)
  -> bit . IntegrationColdstartCheck(timer=timer + 1)
+ (timer < 3 * CYCLE_length - 1 && timer >= 2 * CYCLE_length - 1)
  -> (ok1 -> bit . IntegrationColdstartCheck(timer=timer + 1)
          <> bit . AbortStartup())
+ (timer >= 3 * CYCLE_length - 1)
  -> (ok2 -> macStart . ColdstartJoin(0)
           <> AbortStartup())
);

% During the ColdstartJoin phase, a coldstart node that is not the leading
% coldstart node starts sending frames.
state ColdstartJoin(timer: Nat) =
(
  decode(id, CAS) .  ColdstartJoin()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    ColdstartJoin()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    ColdstartJoin()
+ reset(id) . AbortStartup(attempts=0)

+ bit . (timer < 3 * CYCLE_length)
  -> ColdstartJoin(timer + 1)
  <> enter_operation(id) . NormalOperation()
);

% When a startup attempt is aborted, the MAC is stopped and the StartupPrepare
% state is entered again.
state AbortStartup() =
(
  decode(id, CAS) . AbortStartup()
+ sum id': Sender .
    decode(id, FRAME_HEADER(id')) .
    AbortStartup()
+ sum id': Sender .
    decode(id, FRAME(id')) .
    AbortStartup()
+ reset(id) . AbortStartup(attempts=0)
```

```
  + macStop . abort(id) . Controller()
  );

  % During normal operation, there is no more controlling behaviour of the POC.
  % We must however actively ignore bit and decode actions to avoid deadlock.
  state NormalOperation() =
    bit . NormalOperation()
  + sum s: Symbol . decode(id, s) . NormalOperation()
  + reset(id) . AbortStartup(attempts=0);

  % When startup failed (i.e., the node ran out of startup attempts), this is
  % signalled by a startup_failed action. The other actions are there to avoid
  % deadlocks in the model.
  state Failed() =
    bit . Failed()
  + sum s: Symbol . decode(id, s) . Failed()
  + reset(id) . AbortStartup(attempts=0)
  + startup_failed(id) . Failed();

};

% An absent node is modelled explicitly to be able to reuse this specification
% for two-node scenarios without having to change the number of parallel
% components.
proc AbsentNode(id: Sender) =
  wait|bit|bit . AbsentNode()
+ sum s: Signal . get(id, s) . AbsentNode()
+ put(id, NONE) . AbsentNode();

% A normal node consists of a CODEC, a MAC and a POC that starts immediately.
proc NormalNode(id: Sender) =
  allow({wait|bit|bit, Encode|bit, get, put, Decode, Is_idle, macCAS|macCAS,
         macStart|macStart, macStop|macStop, enter_operation, startup_failed,
         abort, init_sched, reset, attempt_startup},
    comm({encode|encode->Encode, decode|decode->Decode,
          is_idle|is_idle->Is_idle},
      CODEC(id) || Controller(id, 0) || MAC(id, 0, false)
  ));

% A delayed node consists of a CODEC, a MAC and a POC that starts operation
% after a bounded initial delay.
proc DelayedNode(id: Sender) =
  allow({wait|bit|bit, Encode|bit, get, put, Decode, Is_idle, macCAS|macCAS,
         macStart|macStart, macStop|macStop, enter_operation, startup_failed,
         abort, init_sched, reset, attempt_startup},
    comm({encode|encode->Encode, decode|decode->Decode,
          is_idle|is_idle->Is_idle},
      CODEC(id) || DelayedController(id) || MAC(id, 0, false)
  ));
```

```
% If a resetting node is present, then this process controls when it may be
% reset. For larger statespaces, this process can be changed to restrict the
% frequency with which a node can be reset.
proc Resetter =
  reset(reset_node) . Resetter()
+ bit . Resetter();

% The system consists of three nodes. Each node is of type Absent, Normal or
% Delayed. The bus|... multi-actions form the clock tick barrier, all other
% actions are either a result of communication between subcomponents, or actions
% that signal relevant events such as failure to start up.
init
  allow({Get, Put,
         bus|Encode|Encode|Encode|bit|bit|bit ,
         bus|Encode|Encode|wait|bit|bit|bit |bit,
         bus|Encode|wait|wait|bit|bit|bit |bit|bit,
         bus|wait|wait|wait|bit|bit|bit |bit|bit|bit,
         Decode, Is_idle, init_sched,
         macCAS|macCAS, macStart|macStart, macStop|macStop,
         enter_operation, startup_failed, abort, Reset, attempt_startup},
    comm({get|get'->Get, put|put'->Put, reset|reset->Reset},
```

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2009):

09/01	Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf	Compositional Service Trees
09/02	P.J.l. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin	Queue merge: a Binary Operator for Modeling Queueing Behavior
09/03	Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemse	Breadth-Bounded Model Checking
09/04	Muhammad Atif and MohammadReza Mousavi	Formal Specification and Analysis of Accelerated Heartbeat Protocols
09/05	Michael Franssen	Placeholder Calculus for First-Order logic
09/06	Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle	POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy
09/07	Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Liegl and Hannes Werthner	Pattern-based Analysis of Windows Workflow
09/08	Mike Holenderski, Reinder J. Bril and Johan J. Lukkien	Swift mode changes in memory constrained real-time systems
09/09	Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko	Behavioural analysis of an I^2C Linux Driver
09/10	Ugur Keskin	In-Vehicle Communication Networks: A Literature Survey
09/11	Bas Ploeger	Analysis of ACS using mCRL2
09/12	Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote	Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic
09/13	Luca Aceto, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	A Rule Format for Unit Elements
09/14	Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst	Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance
09/15	MohammadReza Mousavi and Emil Sekerinski, Editors	Proceedings of Formal Methods 2009 Doctoral Symposium
09/16	Muhammad Atif	Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems
09/17	Jeroen Keiren and Tim A.C. Willemse	Bisimulation Minimisations for Boolean Equation Systems
09/18	Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran	On-the-fly Auditing of Business Processes
10/01	Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi	Analytical Software Design: Introduction and Industrial Experience Report
10/02	F.E.J. Kruseman Aretz	Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study

10/03	Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	On Rule Formats for Zero and Unit Elements
10/04	Hamid Reza Asaadi, Ramtin Khosravi, MohammadReza Mousavi, Neda Noroozi	Towards Model-Based Testing of Electronic Funds Transfer Systems
10/05	Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nolte	Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited
10/06	Zvezdan Protić	Locally unique labeling of model elements for state-based model differences
10/07	C.G.U. Okwudire and R.J. Bril	Converting existing analysis to the EDP resource model
10/08	Muhammed Atif, Sjoerd Cranen, MohammadReza Mousavi	Reconstruction and verification of group membership protocols
10/09	Sjoerd Cranen, Jan Friso Groote, Michel Reniers	A linear translation from LTL to the first-order modal μ-calculus
10/10	Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien	Extending an Open-source Real-time Operating System with Hierarchical Scheduling
10/11	Eric van Wyk and Steffen Zschaler	1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)
10/12	Pre-Proceedings	3rd International Software Language Engineering Conference
10/13	Faisal Kamiran, Toon Calders and Mykola Pechenizkiy	Discrimination Aware Decision Tree Learning
10/14	J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran	Specification Guidelines to avoid the State Space Explosion Problem
10/15	Daniel Trivellato, Nicola Zannone and Sandro Etalle	GEM: a Distributed Goal Evaluation Algorithm for Trust Management
10/16	L. Aceto, M. Cimini, A.Ingolfsdottir, M.R. Mousavi and M. A. Reniers	Rule Formats for Distributivity
10/17	L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi	Decompositional Reasoning about the History of Parallel Processes
10/18	P.D. Mosses, M.R. Mousavi and M.A. Reniers	Robustness os Behavioral Equivalence on Open Terms
10/19	Harsh Beohar and Pieter Cuijpers	Desynchronisability of (partial) closed loop systems
11/01	Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf	Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!
11/02	M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen	Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification
11/03	H.R. Mahrooghi and M.R. Mousavi	Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols
11/04	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Benefits of Applying Formal Methods to Industrial Control Software
11/05	Jan Friso Groote and Jan Lanik	Semantics, bisimulation and congruence results for a general stochastic process operator
11/06	P.J.L. Cuijpers	Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence
11/07	F.P.M. Stappers, M.A. Reniers and S. Weber	Transforming SOS Specifications to Linear Processes
11/08	Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf	A Component Framework where Port Compatibility Implies Weak Termination
11/09	Tseesuren Batsuuri, Reinder J. Bril and Johan Lukkien	Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol

11/10	Neda Noroozi, Ramtin Khosravi, MohammadReza Mousavi and Tim A.C. Willemse	Synchronizing Asynchronous Conformance Testing
11/11	Jeroen J.A. Keiren and Michel A. Reniers	Type checking mCRL2
11/12	Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran	Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems
11/13	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Experience report on developing the Front-end Client unit under the control of formal methods
11/14	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Ananlyzing a Controller of a Power Distribution Unit Using Formal Methods
11/15	John Businge, Alexander Serebrenik and Mark van den Brand	Eclipse API Usage: The Good and The Bad
11/16	J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius	Investigating the Effects of Designing Control Software using Push and Poll Strategies
11/17	M.F. van Amstel, A. Serebrenik And M.G.J. van den Brand	Visualizing Traceability in Model Transformation Compositions
11/18	F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber	Dogfooding the Structural Operational Semantics of mCRL2
12/01	S. Cranen	Model checking the FlexRay startup phase