FlexRay for Avionics: Automatic Verification with Parametric Physical Layers

Michael Gerke, Rüdiger Ehlers, Bernd Finkbeiner, and Hans-Jörg Peter

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

The automotive FlexRay standard is increasingly attracting attention in the aeronautics industry. Upgrading existing physical layers, such as CAN-based systems, with FlexRay is attractive, especially given that inexpensive FlexRay hardware is available on the market. However, such a change of the application context requires a careful examination of the assumptions the protocol is based on. For example, the FlexRay standard assumes that the harness length is at most 24 meters, a requirement that is typically met by ground vehicles but not by planes. In this paper, we present a methodology for the formal analysis of the impact of changes to the physical setting on the fault tolerance of the protocol. We use the real-time model checker Uppaal to automatically verify the behavior of the FlexRay physical layer protocol when executed on a range of hardware configurations. We report on design considerations and lessons learned in building a model that is sufficiently small to allow for automatic verification and, at the same time, sufficiently precise to describe the intricate real-time interplay of protocol and hardware.

I. Introduction

The application of mass-market bus protocols in the aeronautics context is an attractive alternative to using proprietary protocols whose hardware is typically expensive. However, as x-by-wire applications have strict requirements on the quality of service of the bus protocol, a careful examination of how changes to the physical-layer properties affect the protocol behavior is imperative for the safe application of such protocols in new application contexts.

In this paper, we analyze the FlexRay bus protocol, which stems from the automotive context and has recently generated a lot of interest in the aeronautics industry. FlexRay provides automatic low-level error correction and predictable timing. Upgrading existing physical layers, such as CAN-based systems, with FlexRay is very attractive, especially given that inexpensive FlexRay hardware is available on the market.[1,2] However, the FlexRay specification is based on environment assumptions that do not necessarily hold in the aeronautics context. For example, the FlexRay standard assumes that the harness length is at most 24 meters, a requirement that is typically met by ground vehicles but not by planes.[1,3] This problem can be eliminated by formally verifying that the desired properties of the protocol still hold under the changed assumptions.

A first attempt at formalizing and verifying FlexRay was undertaken in the *Verisoft* project, where Beyer et al. gave a deductive correctness proof.[4] This "paper-and-pencil" verification effort was later extended to a more comprehensive proof that was machine-checked using the interactive theorem prover Isabelle/HOL.[5–7] Such manual correctness proofs can be helpful to increase the confidence in a protocol for a given application context, but they are very expensive and it is doubtful if the manual effort can be scaled to a systematic exploration of the impact of the application context on the behavior of the protocol.

In our own work,[8] we have used the real-time model checker UPPAAL[9] to verify real-time properties of FlexRay's physical layer protocol. The advantage of model checking is that it is completely automatic once the protocol including its physical layer has been formalized using an automata-based framework. Our analysis provides a detailed picture of the robustness of the protocol under changes of the physical layer properties. We showed that, for typical hardware parameters, such as those of a realistic design from the Nangate Open Cell Library,[10] FlexRay tolerates one glitch-affected sample in every sequence of four samples or two glitch-affected samples in every sequence of 88 samples. In fact, this tolerance is robust under variations of the hardware. For example, the variance in the delay of the propagation of values on the bus can be increased to up to 7570 ps without negative impact on the tolerable glitch patterns.

Although the model checking process itself is automatic, developing a model that is sufficiently small to make automatic verification feasible and, at the same time, sufficiently precise to describe the intricate real-time interplay of protocol and hardware, is a formidable challenge. Models of real-time communication protocols suffer from two sources of state explosion. Firstly, the combination of multiple *clocks*, which are needed to describe the behavior of oscillators and low-level timing details such as hold times and propagation delays, leads to a combinatorial explosion of the state space. Secondly, the *data* domain is huge. In theory, a model of the FlexRay physical layer protocol must account for more than 10^{600} different possible messages.

In this paper, we describe the modeling principles that have allowed us to analyze the FlexRay physical layer with a high degree of precision using an off-the-shelf model checker. Our approach aims for efficiently checkable models by using both clocks and data as economically as possible. The introduction of clocks can often be avoided if the timing of two events depend on each other. In FlexRay, for example, certain events on the bus are triggered by the sender, whose actions are in turn triggered by the ticks of its oscillator. By reusing the clock that measures the time between the ticks in the description of the resulting actions and events, we can specify the entire causal chain with a single clock. The explosion in the data domain can be reduced significantly through the careful introduction of nondeterminism into the model. While we cannot restrict the space of potential messages without affecting the soundness of our analysis, we eliminate the *storage* of data in the model whenever possible by generating data nondeterministically at data-dependent branchings in the protocol.

The paper is structured as follows. We begin with a brief summary of the FlexRay physical layer protocol in Section II. In Section III, we describe our verification approach based on automatic parameter exploration. In Section IV, we discuss the principles that have guided the development of our model. The results of our analysis are presented in Section V. In Section VI, we discuss connections to related work.

II. The FlexRay Physical Layer Protocol

In this section, we briefly review the FlexRay[11] physical layer protocol.

Topology and timing. The FlexRay protocol is used to establish the communication in a network of *electronic control units* (ECUs), which can be arbitrary embedded devices. Each ECU is connected via a *controller* to a shared *communication channel* (which we call *bus* in the following).

The *timing hierarchy* [11, Chapter 5] of FlexRay is shown in Figure 1. In one cycle of the communication, one static slot is reserved for usage by each controller. Thus, every controller connected to the bus gets the chance to send a message. This *static segment* of the cycle is followed by a *dynamic segment* where every controller is allowed to try to send a message outside the normal schedule. FlexRay uses a *time division multiple access* (TDMA) scheme during the static segment, and a flexible FTDMA scheme during the dynamic segment, both of which exclude collisions.[12, 13] The end of the cycle consists of a small *symbol window* for protocol-related communication, and finally a *network idle time*.

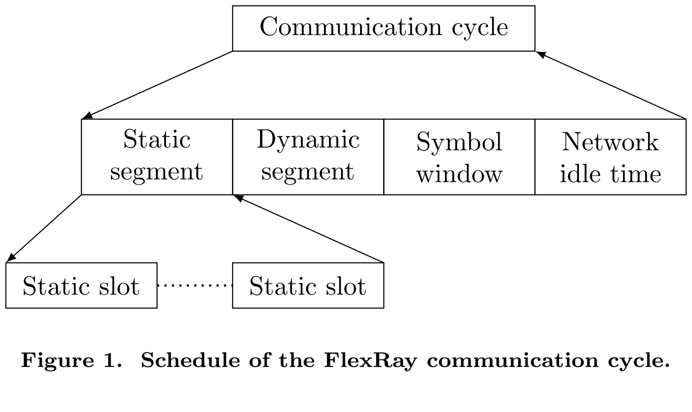
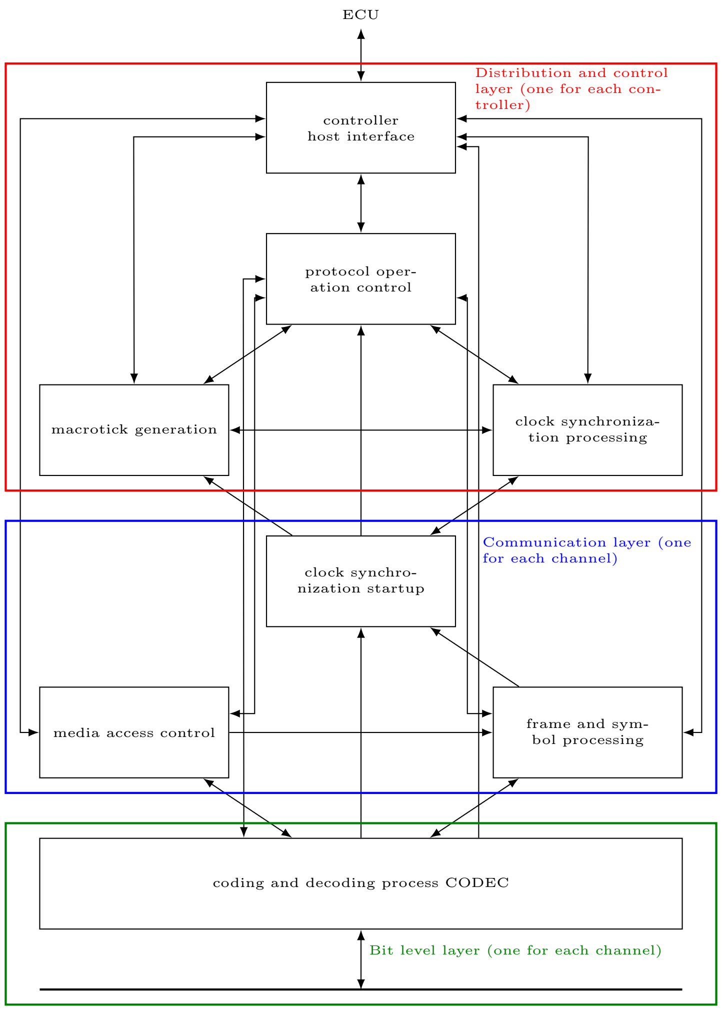


Figure 1. Schedule of the FlexRay communication cycle.

Communication. A controller is based on a layered architecture comprising communicating processes. Each process is assigned either to the *distribution-and-control layer*, the *communication layer*, or the *bit-*



ECU
Distribution and control layer (one for each controller)
controller host interface
protocol operation control
macrotick generation
clock synchronization processing
Communication layer (one for each channel)
clock synchronization startup
media access control
frame and symbol processing
coding and decoding process CODEC
Bit level layer (one for each channel)

Figure 2. Overview on the architecture of a FlexRay controller.

level layer. Figure 2 shows an overview.

The communication between two controllers is handled by the coding and decoding processes (CODEC). The FlexRay standard[11] specifies a format for messages, the *message frames* (or just *frames*), which contain the actual payload that is to be transferred, as well as protocol-related information. The format of a frame is shown in Figure 3.

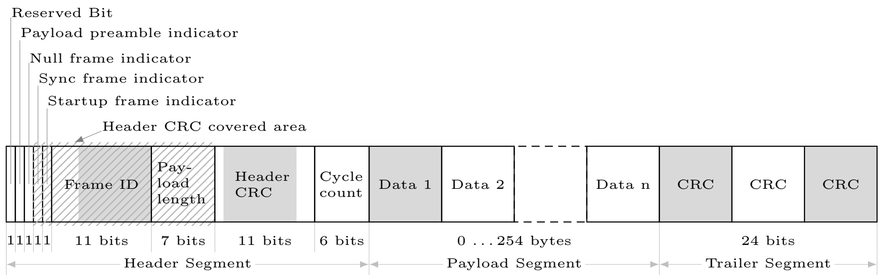


Figure 3. Format of a frame.

A message frame is transmitted as a structured stream [11, Section 3.2.1.1] of bits as shown in Figure 4. The start of the stream is the *transmission start sequence* (TSS), which consists of a sequence of zeros and precedes every transmission. The length of this sequence depends on the structure of the overall network and may vary between 3 to 15 bits [11, Sections B.2.1]. After the TSS, the *frame start sequence* (FSS) signals the start of a message transmission. The FSS consists of a single high bit. The receiving controller accepts a transmission even if the FSS is received zero or two times. The bit string of the message is partitioned into bytes. Each message byte is prefixed with a *byte start sequence* (BSS). The BSS consists of one high bit followed by one low bit. At the end of the message, a *frame end sequence* (FES) is appended. The FES consists of one low bit followed by one high bit. Every bit of the stream is transmitted as a bit cell, i.e., the bit value is held for eight clock cycles.

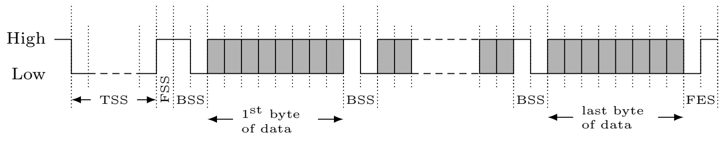


Figure 4. Format of a message bit stream.

III. Parameter Exploration based on Timed Automata

In this section, we recall the timed automaton model by Alur and Dill, introduce the parametric extension, and briefly discuss our FlexRay model.

A. Timed Automata

We use *networks of timed automata*[14,15] as our modeling language. Timed automata extend finite state machines by *clock variables* (or just *clocks*) whose values range over the nonnegative reals. For a comprehensive overview and a more formal treatment of the subject, we refer to the textbook by Clarke et al. [16, Chapter 17].

Syntax. A timed automaton has a finite set of *locations* with a dedicated *initial location.* Locations are connected via *edges*. Clocks are referenced in *clock constraints*, which can be used as enabledness conditions, called *guards*, on edges, and as *location invariants* (or just *invariants*) on locations. A clock constraint is a conjunction of clock comparisons of the form

$$x_1 \prec_1 c_1 \ \wedge \ \ldots \ \wedge \ x_n \prec_n c_n,$$

where $x_1, \ldots, x_n$ are clocks, $\prec_1, \ldots, \prec_n \in \{<, \leq, =, \geq, >\}$, and $c_1, \ldots, c_n$ are nonnegative integer constants. Each edge is additionally labeled with an *action* and with a set of clocks that should be reset to zero when the edge is taken.

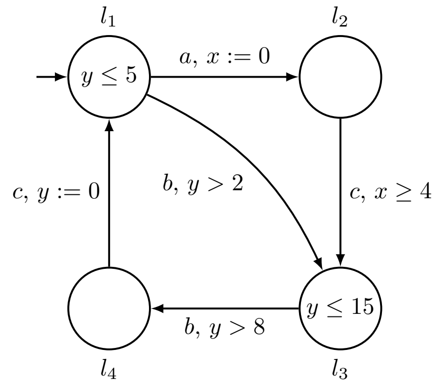


Figure 5. Example timed automaton with the locations l_1, l_2, l_3 and l_4, the clocks x and y, as well as the actions a, b and c.

Figure 5 shows an example timed automaton with locations l_1, l_2, l_3 and l_4, clocks x and y, and actions a, b and c. The initial location is l_1. The locations l_1 and l_3 have invariants $y \leq 5$ and $y \leq 15$, the other locations l_2 and l_4 have no invariants (i.e., they have the invariant **true**). The edge between l_1 and l_2 is labeled with action a, has a reset for x, and no guard (i.e., its guard is **true**). The edge between l_2 and l_3 is labeled with action c, has no resets, and a guard $x \geq 4$.

Semantics. The *configuration* (or *state*) of a timed automaton comprises the location the automaton is currently in and the current clock values. Due to the continuous value domain of the clocks, there are infinitely many states. The set of all states is called the *state space.* The states are connected via two types of transitions:

- *action transitions* that represent an instantaneous change to another location by taking an edge;
- *delay transitions* that represent a uniform elapsing of time (i.e., a synchronous increase on all clock values).

Two states s and s', whose locations are l and l', respectively, are connected via an action transition iff there is an edge e going from l to l' such that the clock values in s satisfy the guard of e and the clock values after executing the resets of e satisfy the location invariant of l'. The states s and s' are connected via a delay transition iff $l = l'$ and the clock values after the delay still satisfy the invariant of l.

The execution of a timed automaton starts in the *initial state*, comprising the initial location and where all clock values are 0, and corresponds to a path of transitions connecting states. Such a path is also called a *timed trace* (or just a *trace*) that is represented by a (possibly infinite) sequence of actions and delays.

For example, the execution of the timed automaton in Figure 5 starts in l_1 with $x = y = 0$. Due to l_1's invariant, the execution has to leave l_1 within 5 time units. During that time, an action transition leading to l_2 can be executed, resetting x. Then, at l_2, we have $x \leq y$ and time can elapse arbitrarily. After at least 4 time units have passed, another action transition to l_3 can be executed unless $y > 15$, in which case the target invariant would not be satisfied.

Networks. Timed automata can be syntactically composed into *networks*, in which the automata run in parallel and synchronize on shared actions. We also call the timed automata in a network *components.*

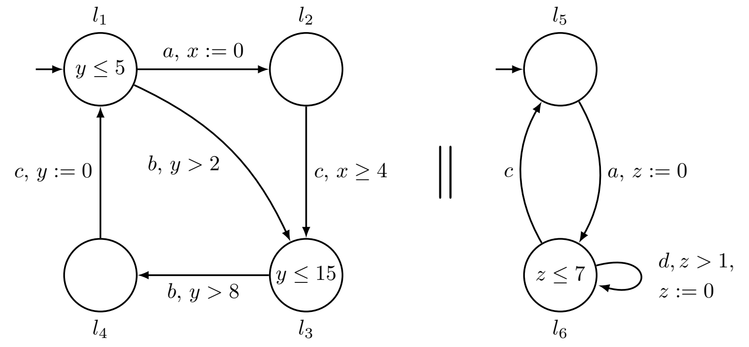


Figure 6. Example network comprising two timed automata that synchronize on actions a and c.

Figure 6 shows an example network of two timed automata, which synchronize on actions a and c. Provided that the clock guards are satisfied, an automaton can only execute an edge labeled with action a or c if the other automaton executes an edge with the same action concurrently. Otherwise, the two automata run asynchronously in the sense that they can independently execute edges labeled with action b or d.

B. Parametric Models

As a syntactic extension, we also allow *integer variables* and *arrays of integers* (or just *arrays*) with a bounded domain in the description of our timed automata. Integer variables and arrays are referenced in *integer expressions*, which can be arbitrary arithmetic expressions yielding integer values. Sometimes, we use pseudocode such as foo(bar()) in integer expressions, where foo and bar are functions returning integer values. We also allow expressions of the form $v_1 \prec v_2$, where v_1 and v_2 are integer variables and $\prec \in \{<, \leq, =, \geq, >\}$, yielding the values 0 or 1. We call such expressions *integer constraints.*

Guards and location invariants are extended to conjunctions of integer and clock constraints. Additionally, integer variables can also be used in place of constants in clock constraints. For manipulating integer variables, edges are extended by *update expressions* of the form

$$v_1 := e_1, \ \dots \ , v_n := e_n,$$

where $v_1, \dots, v_n$ are integer variables or arrays and $e_1, \dots, e_n$ are integer expressions.

We identify a subset of the integer variables as *parameters.* Parameters are not assigned an initial value and do not appear on the left hand side of update expressions. All other integer variables are initialized with a specific value. If an initial value is not explicitly given, we assume the initial value 0. Figure 7 shows a timed automaton with parameters P, Q and R, and additional integer variables i and j.

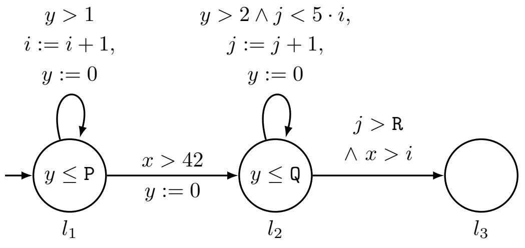


Figure 7. Example timed automaton with parameters P, Q and R, and additional integer variables i and j.

C. Modeling FlexRay with Parametric Timed Automata

We model the FlexRay physical layer protocol as a network of parametric timed automata. As illustrated in Figure 8, we partition the components of the network into two parts: one that represents the *protocol layer* and one that represents the *underlying hardware* including an *error model.* The correctness of the higher protocol levels as well as FlexRay's ability of to deal with errors outside the error model is beyond the scope of this work.

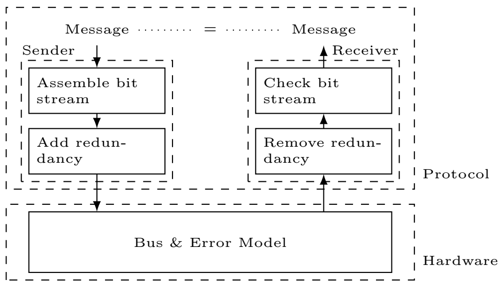


Figure 8. The structure of the model. The arrows indicate the flow of information.

In our model, the *sender* embeds the message in a structured *bit stream.* To introduce redundancy, every bit of this stream is sent as a *bit cell* in which the bit value is held for eight clock cycles. The *receiver* in turn reads one value in every clock cycle from the bus (the *samples*), removes the redundancy and checks whether the received message deviates from the sent one. The error model, which is described in more detail in Section V–A, is incorporated into the components of the hardware layer. A detailed description of the complete model is available in a technical report.[17]

IV. Modeling Principles

In this section, we discuss the modeling principles that have allowed us to analyze the FlexRay physical layer with a high degree of precision using an off-the-shelf model checker. Model checking is a complete verification technique: unlike, for example, testing, model checking accounts for every possible behavior of the system. The price for the complete coverage is that model checking is computationally expensive: model checking a network of timed automata takes exponential time in the size of the network. Our modeling approach aims for efficiently checkable models by using clocks and data as economically as possible. In the following subsections we discuss these two principles in more detail.

A. Modeling with Fewer Clocks

The number of clocks has a dramatic impact on both the running time and the memory consumption of the model checker. The model checker tracks the relationships between the different clocks using data structures such as difference bound matrices[18] (DBMs), which record, for every pair of clocks, an upper and a lower bound on the current difference between the clock values. The number of different values of this data structure that are encountered during a run of the model checker grows exponentially with the number of clocks.

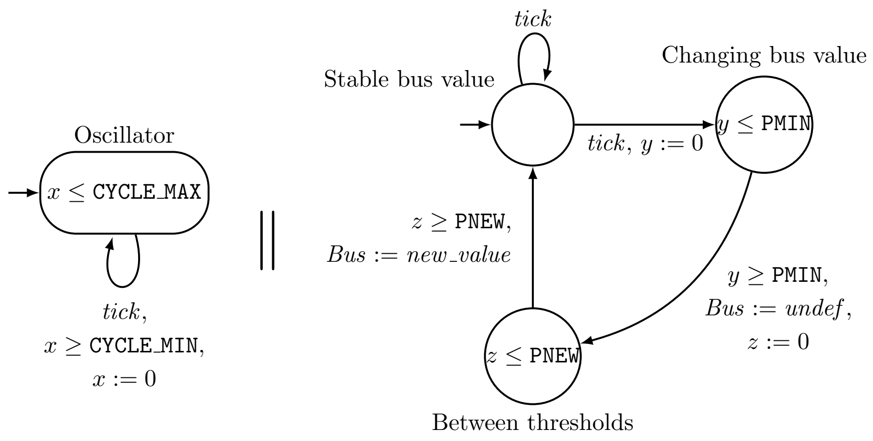


Figure 9. Model of the sender's oscillator and the bus, synchronizing on action *tick*. We use one clock for the oscillator, one clock for the time to reach the first threshold, and another one for the time to reach the second threshold.

Our modeling approach reduces the number of clocks by exploiting dependencies between the clocks. Typically, if some event triggers another event, then the clock measuring the timing of the first event can also be used to measure the timing of the second event. In FlexRay, such dependencies start with the two oscillators in the sender and receiver, which each cause various dependent events. As a result, we can model the *entire* physical layer protocol and the hardware with just *two* clocks.[a]

We illustrate the approach with an example, where we model some propagation of signals along the bus. A timed automaton network for this scenario is shown in Figure 9. We start in a state where the bus value is stable. When a clock tick occurs (driven by the local oscillator of the sender), the sender can begin to drive a new value on the bus. The receiver does not immediately see the new value. Only after PMIN time units, the sender has driven the voltage on the bus beyond the threshold for recognizing the old bus value. After additional PNEW = PMAX − PMIN time units, the value of the bus at the receiver's side has been driven beyond the threshold for the new bus value, as shown in Figure 18. In the model, we implement this by measuring three time spans: the length of the clock cycle, the time to drive the bus beyond the first threshold, and the time to drive it beyond the second threshold. A clock cycle does not always have the same length (there are no perfect oscillators in practice), but its length is always in between CYCLE_MIN and CYCLE_MAX. We have one clock for measuring each of these time spans.

Analyzing our model, we find dependencies between the clocks: The events on the bus are triggered by the sender. The sender's actions are triggered by a tick from its oscillator. Thus, the events on the bus are triggered by a tick from the sender's oscillator. This allows us to describe events on the bus using the clock for generating the ticks of the sender's oscillator. We observe that the we can deduce from the clock value of x the values of y and z. In the model of Figure 9, x and y always have the same value, and z is equal to $x +$ PMIN in the "Between thresholds" location, which is the only location in which the value of z is used.

[a]This does not include the error model, which needs a separate clock.

Thus, we can remove the clocks y and z, replace any reference to y by x, and replace any reference to z by $x + \texttt{PMIN}$. Exploiting the equality $\texttt{PNEW} = \texttt{PMAX} - \texttt{PMIN}$, we obtain the simplified model shown in Figure 10.

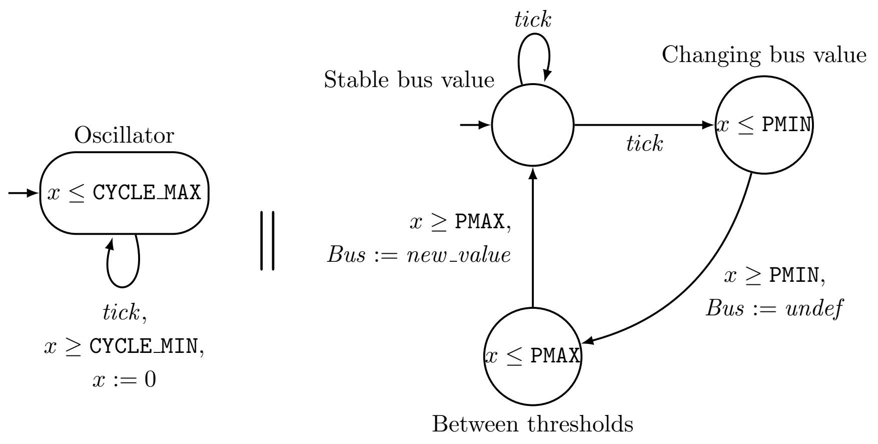

Oscillator
$x \leq \texttt{CYCLE\_MAX}$
$tick$, $x \geq \texttt{CYCLE\_MIN}$, $x := 0$
$tick$
Stable bus value
Changing bus value
$x \leq \texttt{PMIN}$
$tick$
$x \geq \texttt{PMAX}$, $Bus := new\_value$
$x \geq \texttt{PMIN}$, $Bus := undef$
$x \leq \texttt{PMAX}$
Between thresholds

Figure 10. Optimized model of the sender's oscillator and the bus, synchronizing on action *tick*. Here, we use just the clock needed to generate the *tick* action of the sender's oscillator, under the assumption that $\texttt{PMIN} \leq \texttt{PMAX} \leq \texttt{CYCLE\_MIN}$ holds.

B. Modeling with Less Data

The number of possible messages in a transmission protocol is astronomic: in the case of FlexRay, there are more than 10^{600} possible different messages. Since a natural specification of the correctness of the protocol would require that *every* message is transmitted correctly, the large number of messages immediately translates to an equally large state space: for example, our model might first fix and store the data to be sent, then transmit the message via sender and receiver, and finally compare the stored data against the delivered message.

To avoid this effect, we must avoid storing the actual data in the protocol. We let the model *forget* the data to be sent and instead generate the data on-the-fly at data-dependent branchings of the protocol: whenever the choice of a transition in the model depends on the actual message, we just nondeterministically choose a transition. In addition to the data to be sent, one can often also ignore the format of the message. Protocols typically have a fixed format for messages, such as the format of FlexRay message frames shown in Figure 3. The state components needed to enforce this format can be eliminated if the modeled protocol layer does not depend on the format.

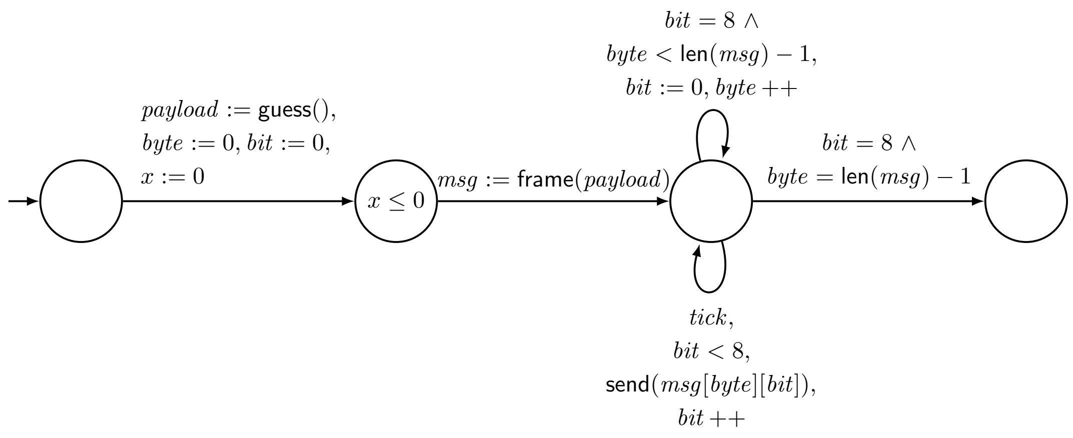

$payload := \mathsf{guess}()$, $byte := 0, bit := 0$, $x := 0$
$x \leq 0$
$msg := \mathsf{frame}(payload)$
$bit = 8 \wedge$ $byte < \mathsf{len}(msg) - 1$, $bit := 0, byte{++}$
$bit = 8 \wedge$ $byte = \mathsf{len}(msg) - 1$
$tick$, $bit < 8$, $\mathsf{send}(msg[byte][bit])$, $bit{++}$

Figure 11. Generating a message payload, encasing it in the frame format, and sending it.

As an example, consider the automaton shown in Figure 11. We initialize the data that should be transmitted in some array *payload*, whose values are chosen nondeterministically using the (pseudocode) function guess(). Then, we use pseudocode to encase *payload* in the frame format shown in Figure 3 and obtain a message frame that is stored in an array *msg*. The bits of *msg* are then sent to a message stream generator that guarantees the format seen in Figure 4. If we drop the idea that only legal message frames are to be transmitted in the model, but rather all messages of a possible length, then we can remove the necessity to store the initial message altogether: we can just guess the message bits on-the-fly as we go. We can additionally guess the length of the message on-the-fly. If we allow them to have an arbitrary length (in bytes), we do not even have to store how many bytes we have already sent. The resulting automaton is shown in Figure 12. Its state space is reduced to only 11 states: The automaton could be in the first location, it could be in the second location and *bit* could have any integer value from 0 to 8, or it could be in the last state, where only 8 is a possible value for *bit*.

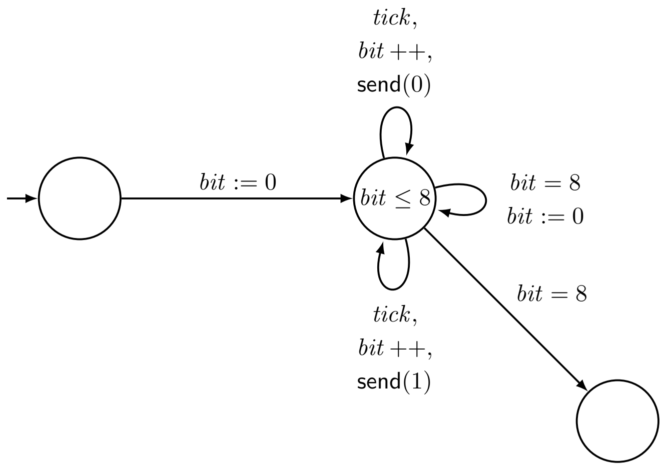


Figure 12. Generating and sending of a message while abstracting from its actual length, contents and format.

The drawback of ignoring the message is, of course, that we can no longer specify that the messages are received correctly. It is sufficient, however, to store a *single bit* of the message, as long as the bit is chosen nondeterministically. If a bit is not correctly transmitted, the nondeterminism guarantees that there is at least one trace of the model where this bit is stored. In order to verify that the one stored bit is correctly received, we need to identify the received bit that corresponds to the stored bit. For this purpose, the model of the sender shown in Figure 13 stores the position pos_{stored} of the bit in the message, and the model of the receiver shown in Figure 14 counts the received message bits until it identifies the bit corresponding to the stored bit bit_{stored}.

The timed automata in Figures 15 and 16 show a version of the model that has been optimized based on an additional observation: we can reduce the counting effort for identifying the position of the stored bit by keeping track of the number of bits that are in transit. We introduce a counter *offset* that stores the number of bits in transit when the sender model stores the bit. While the bit is not yet stored, the sender model shown in Figure 15 increments *offset* whenever it sends a bit to the message stream generator. While the receiver model shown in Figure 16 has not yet verified the correct reception of the stored bit, it decrements *offset* whenever it has received a message bit. When a bit has been stored, the receiver model checks the value of *offset* to identify the received bit that corresponds to the stored bit.

V. Analysis of the FlexRay Physical Layer Protocol

We have used the real-time model checker UPPAAL[9] to analyze the impact of changes in the parameters of the model, reflecting changes in the application context, on the fault tolerance of the model. We start this section by describing the parametric error model, which we use to characterize fault tolerance. In the following subsections, we present the results of the analysis with UPPAAL.

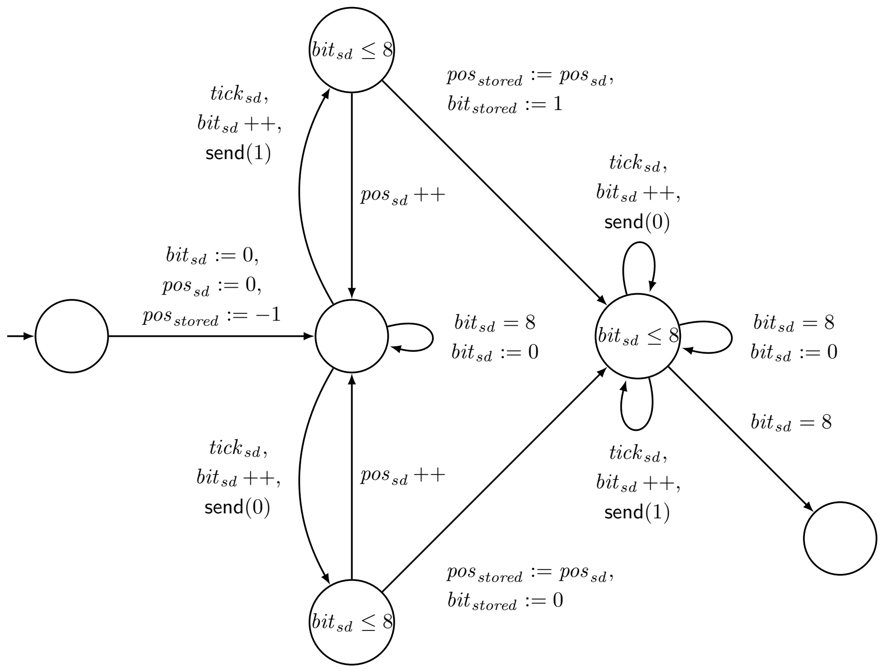


Figure 13. Generating and sending of a message while abstracting from its actual length, contents and format. The sender model nondeterministically chooses a bit to store and stores its position as well.

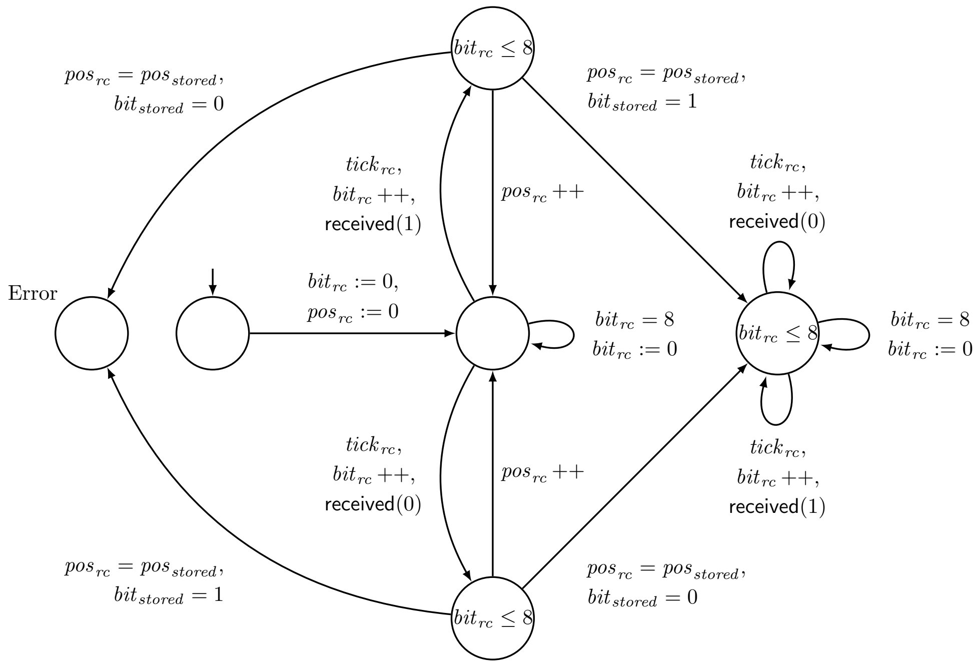
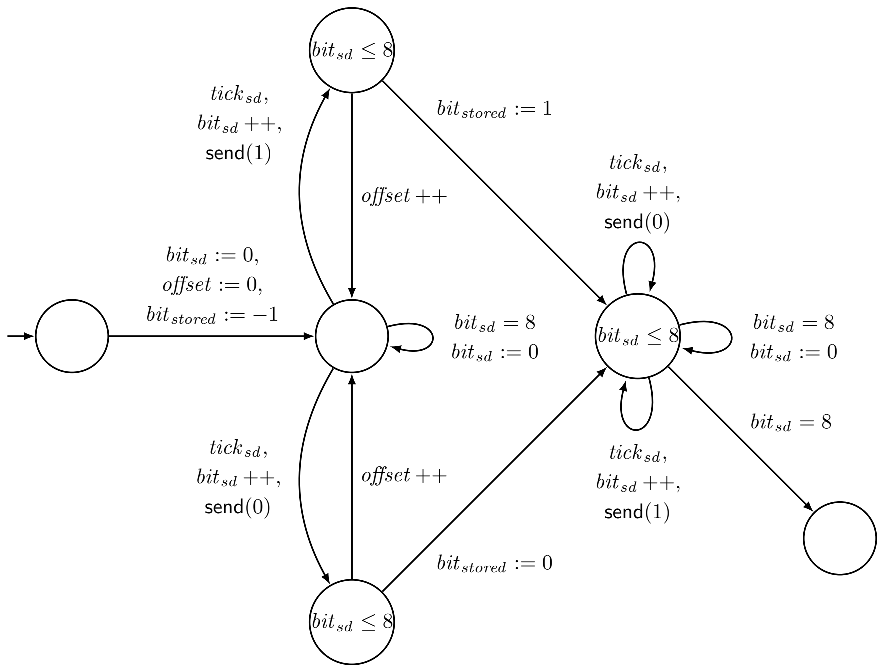
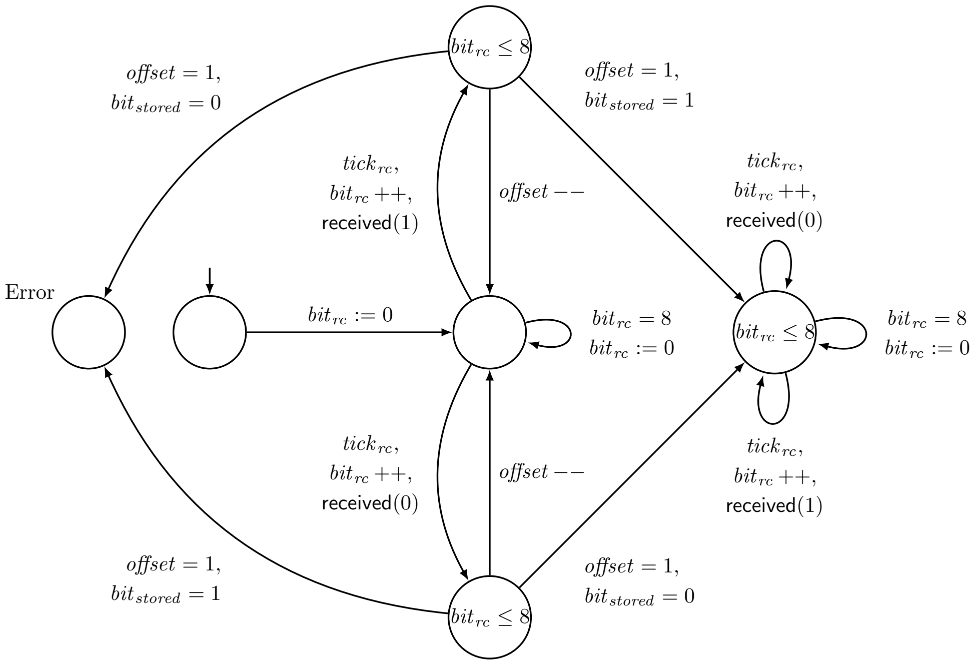


Figure 14. Receiving a message while abstracting from its actual length, contents and format. The receiver model checks that the bit received at the position of the stored bit has the same value as the stored bit.



$bit_{sd} \leq 8$
$tick_{sd}$, bit_{sd} ++, send(1)
$bit_{stored} := 1$
$offset$ ++
$tick_{sd}$, bit_{sd} ++, send(0)
$bit_{sd} := 0$, $offset := 0$, $bit_{stored} := -1$
$bit_{sd} = 8$ $bit_{sd} := 0$
$bit_{sd} \leq 8$
$bit_{sd} = 8$ $bit_{sd} := 0$
$bit_{sd} = 8$
$tick_{sd}$, bit_{sd} ++, send(0)
$offset$ ++
$tick_{sd}$, bit_{sd} ++, send(1)
$bit_{stored} := 0$
$bit_{sd} \leq 8$

Figure 15. Generating and sending of a message while abstracting from its actual length, contents and format. The sender model nondeterministically chooses a bit to store and stores the number of bits in transit in *offset*.



$bit_{rc} \leq 8$
$offset = 1$, $bit_{stored} = 0$
$offset = 1$, $bit_{stored} = 1$
$tick_{rc}$, bit_{rc} ++, received(1)
$offset$ −−
$tick_{rc}$, bit_{rc} ++, received(0)
Error
$bit_{rc} := 0$
$bit_{rc} = 8$ $bit_{rc} := 0$
$bit_{rc} \leq 8$
$bit_{rc} = 8$ $bit_{rc} := 0$
$tick_{rc}$, bit_{rc} ++, received(0)
$offset$ −−
$tick_{rc}$, bit_{rc} ++, received(1)
$offset = 1$, $bit_{stored} = 1$
$offset = 1$, $bit_{stored} = 0$
$bit_{rc} \leq 8$

Figure 16. Receiving a message while abstracting from its actual length, contents and format. If a bit is stored, the receiver model checks that a bit received with $offset = 1$ has the same value as the stored bit.

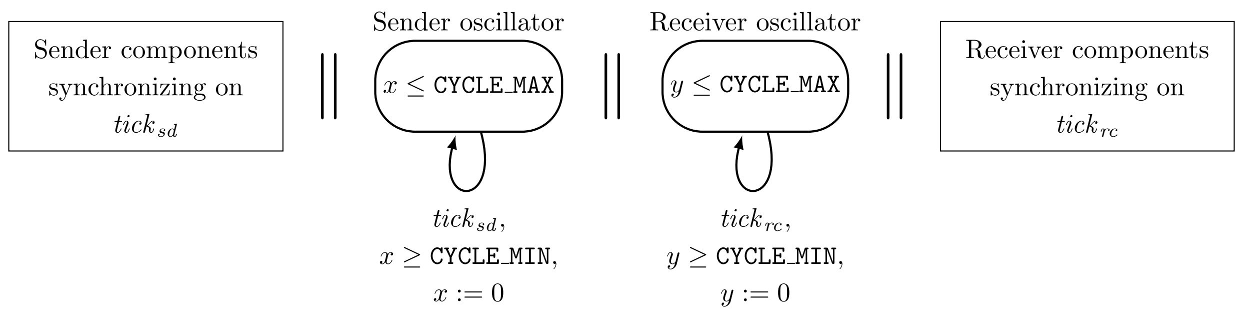

Sender components synchronizing on $tick_{sd}$
Sender oscillator
$x \leq$ CYCLE_MAX
$tick_{sd}$, $x \geq$ CYCLE_MIN, $x := 0$
Receiver oscillator
$y \leq$ CYCLE_MAX
$tick_{rc}$, $y \geq$ CYCLE_MIN, $y := 0$
Receiver components synchronizing on $tick_{rc}$

Figure 17. Network of parametric timed automata showing the oscillators for sender and receiver.

A. The Error Model

We consider two types of erroneous behavior: *jitter* induced by the asynchronous nature of physical layer protocols, and *glitches* induced by influences from the environment.

Jitter. Asynchronous communication protocols must deal with several undesired effects due to the displacement of pulses in the signal. Since sender and receiver do not share a common oscillator, there may be a drift between the local oscillators. We model this drift as parametric timed automata as shown in Figure 17. `CYCLE_MIN` and `CYCLE_MAX` are parameters that encode the potential deviation in the pulsing resulting in a *clock drift* between sender and receiver. We have that

$$\texttt{CYCLE\_MIN} = \texttt{CYCLE} - \frac{\texttt{DEVIATION}}{2} \qquad \text{and} \qquad \texttt{CYCLE\_MAX} = \texttt{CYCLE} + \frac{\texttt{DEVIATION}}{2}.$$

Each other component in the network either synchronizes on $tick_{sd}$ or $tick_{rc}$, depending whether it represents a part of the sender or the receiver, respectively.

Also subsumed under the term *jitter* is when the transition between voltage levels takes varying amounts of time, as depicted in Figure 18. This kind of erroneous behavior is reflected in the modeling of the hardware layer, as shown in Figure 10 on Page 8.

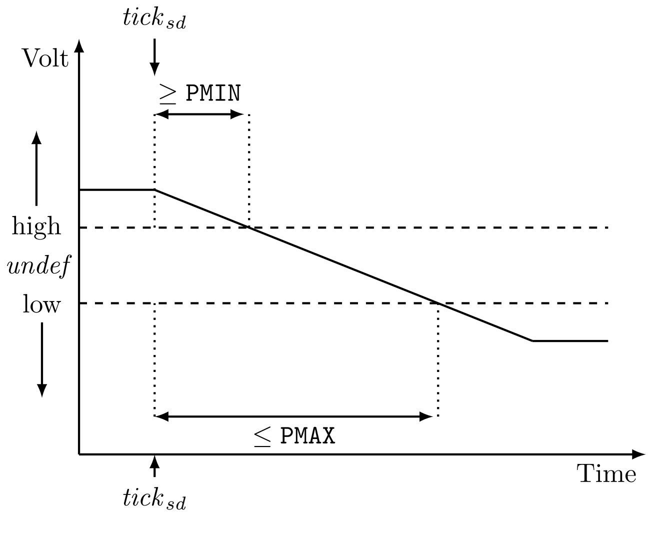
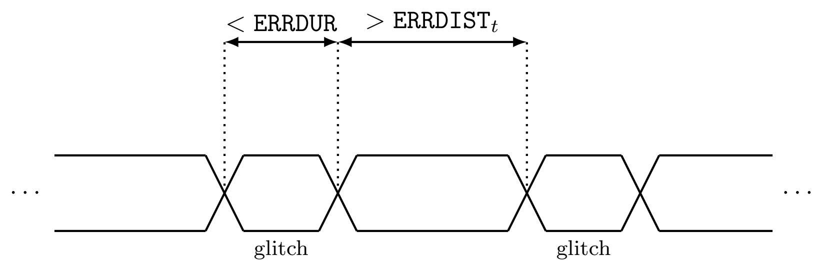
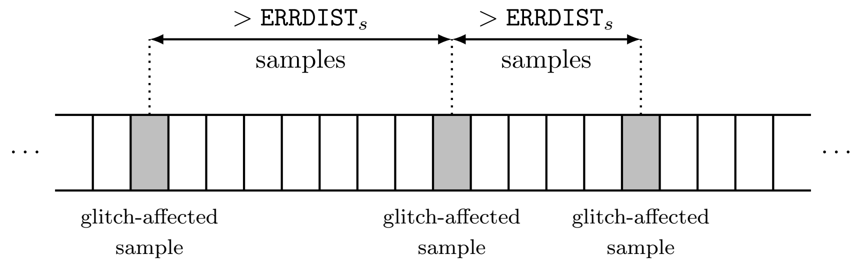


Figure 18. Characteristic timing diagram of a transition between voltage levels.

Glitches. Environmental interferences can always disturb electronic communication. The purpose of a fault-tolerant physical layer protocol is to compensate for such disturbances up to a certain extend. A



(a) Real-time glitch pattern, parametrized in the duration ERRDUR and glitch-free period ERRDIST$_t$.



(b) Sample glitch pattern: 1 out of ERRDIST$_s + 1$.

Figure 19. Real-time vs. sample glitch patterns.

stream of bits transmitted between a sender and a receiver might get compromised at some positions. That is, some bit values in the received stream might differ from the corresponding values in the sent stream. Such a deviation for a particular bit is called a *glitch*.

In order to explore the limits of a particular compensation mechanism, we incorporate the nondeterministic occurrence of glitches in our model. The way how glitches might occur is formalized as *glitch patterns*, where we distinguish between *real-time glitches* and *sample glitches*. A *real-time glitch pattern* is parametrized in the duration of a glitch and the minimum glitch-free period between any pair of glitches. A *sample glitch pattern* is parametrized in the number n of glitch-affected samples that may occur in any sequence of ERRDIST$_s + n$ consecutive samples. Figure 19 shows the two types of glitch patterns.

B. Model Checking the FlexRay Physical Layer Protocol

Our first analysis is based on an instantiation of the protocol model using standard values for all parameters, as shown in Table 1. The parameter values are taken from the FlexRay standard[11] and the Nangate Open Cell Library.[10] Using Uppaal version 4.1.4, we verified that the model with this configuration tolerates at most two glitch-affected samples at arbitrary positions, including next to each other, in every sequence of 88 consecutive samples. This was achieved using the sample glitch model. However, the counting associated with the sample glitch model introduced a considerable amount of discrete complexity: Uppaal needs 250 minutes and 89 GiB of memory to verify this property.

We also used the sample glitch model to verify that our previous result[8] that a model with the configuration shown in Table 1 tolerates one glitch-affected sample in every sequence of 4 consecutive samples is still valid for our new model. To model real-time glitches, we introduced a third clock to measure the duration of a glitch ERRDUR and the glitch-free period ERRDIST$_t$. The results from the real-time glitch model confirmed our findings from the sample glitch model: a glitch of less than $2 *$ CYCLE_MIN and a glitch-free period of more than $86 *$ CYCLE_MAX in between two glitches are tolerable. Hence, if a glitch can affect at most two adjacent samples, the next 86 samples have to be unaffected by glitches, confirming that two glitch-affected samples next to each other in a sequence of 88 consecutive samples are tolerable. This model was analyzed in just 23 seconds using only 111 MiB.

If the glitch is shorter than $1 *$ CYCLE_MIN, the glitch-free period in between two glitches can be shortened to longer than $3 *$ CYCLE_MAX. Hence, if a glitch can affect at most one sample, the next 3 samples have to be unaffected by glitches, confirming that one glitch-affected sample in a sequence of four consecutive samples

Table 1. Standard parameter values based on conservative approximations of the parameters taken from the FlexRay standard[11] and the Nangate Open Cell Library.[10]

Parameter	Value	Corresponds to
`CYCLE`	$10,000$	$\frac{1}{80\,MHz} = 12.5\,ns$
`DEVIATION`	30	$\pm 0.15\,\%$
`SETUP`	368	$460\,ps$
`HOLD`	1160	$1450\,ps$
`PMIN`	12	$15\,ps$
`PMAX`	1160	$1450\,ps$

Table 2. Tolerable glitch patterns with standard parameter values. The glitch pattern "y out of x" stands for "at most y glitch-affected samples in x consecutive samples" and thus an error distance of $x - y$.

Glitch pattern	Parameter	Value	Corresponds to
1 out of 4	$\texttt{ERRDIST}_s$	3	3 samples
2 out of 88	$\texttt{ERRDIST}_s$	86	86 samples
Short real-time glitch	`ERRDUR`	`CYCLE_MIN`	$12.48125\,ns$
	$\texttt{ERRDIST}_t$	$3 * \texttt{CYCLE\_MAX}$	$37.55625\,ns$
Long real-time glitch	`ERRDUR`	$2 * \texttt{CYCLE\_MIN}$	$24.9625\,ns$
	$\texttt{ERRDIST}_t$	$86 * \texttt{CYCLE\_MAX}$	$1033.548\,ns$

is tolerable. This model was analyzed in just 45 seconds using only 172 MiB.

Our results based on the standard parameter values from Table 1 are summarized in Table 2.

C. Parameter Exploration

In our second analysis, we no longer fix the standard parameter values from Table 1, but rather investigate the impact of changes to the parameter values on the resulting fault tolerance. Table 3 shows the results with respect to changes to *either* the hardware parameters `PMIN` and `PMAX`, *or* to `DEVIATION`. The analysis provides stringent hardware requirements which, if met, will guarantee robustness against the respective glitch patterns. Note the subtle difference in the maximal tolerable delay variance `PMAX` − `PMIN` and clock drift `DEVIATION` between short real-time glitches and long ones. This suggests that there is an intricate relationship between the glitch patterns and the tolerable parameter changes.

Our results demonstrate a substantial resilience of the FlexRay physical layer protocol against imprecise oscillators as well as against changes in the variance of the propagation delay. The latter result is of particular interest in the aeronautics context, as the longer harnesses used in planes[1] will likely increase the delay variance. Based on data from the Nangate Open Cell Library,[10] we initially assumed a variance in the propagation delay of $1435\,ps$, but this value can be increased up to $7570\,ps$ without a negative impact on the tolerable glitch patterns.

VI. Related work

The combination of FlexRay's rich feature set and the availability of cheap off-the-shelf bus components aroused interest in extending the protocol's use from the automotive into the avionics domain. Srinivasan and Lundqvist[19] compared FlexRay with two older bus protocol standards, namely MIL-STD-1553 and TTP/C. They conclude that FlexRay is the most versatile of these standards and should be the protocol of choice in the long run. Heller and Reichel[1] discuss the electrical effects of the high cable length that is typically present in the aeronautics context onto the FlexRay bus protocol. They propose a simulation-based analysis of the performance of alternatives to FlexRay's default physical layer, such as RS485. Our analysis can be used to additionally determine the constraints on the physical setting that allow the FlexRay physical layer protocol to be used without negative impact on its fault tolerance.

Paulitsch and Hall[2] discuss the problem of *fault containment* in FlexRay, i.e., ensuring that the bus

Table 3. Impact of changes to the parameter values on the tolerable glitch patterns. The glitch pattern "at most y" means "at most y glitch-affected samples in the overall stream at arbitrary positions".

Changed parameter	Tolerable glitch patterns
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6086$	1 out of 4
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6086$	at most 2
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6056$	2 out of 88
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6086$	$\mathtt{ERRDUR} = \mathtt{CYCLE\_MIN} \wedge \mathtt{ERRDIST}_t = 3 * \mathtt{CYCLE\_MAX}$
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6056$	$\mathtt{ERRDUR} = 2 * \mathtt{CYCLE\_MIN} \wedge \mathtt{ERRDIST}_t = 86 * \mathtt{CYCLE\_MAX}$
$\mathtt{DEVIATION} \leq 92$	1 out of 4
$\mathtt{DEVIATION} \leq 80$	2 out of 88
$\mathtt{DEVIATION} \leq 92$	at most 2
$\mathtt{DEVIATION} \leq 218$	at most 1
$\mathtt{DEVIATION} \leq 348$	none
$\mathtt{DEVIATION} \leq 92$	$\mathtt{ERRDUR} = \mathtt{CYCLE\_MIN} \wedge \mathtt{ERRDIST}_t = 3 * \mathtt{CYCLE\_MAX}$
$\mathtt{DEVIATION} \leq 90$	$\mathtt{ERRDUR} = 2 * \mathtt{CYCLE\_MIN} \wedge \mathtt{ERRDIST}_t = 86 * \mathtt{CYCLE\_MAX}$

remains operational upon the failure of some connected component. When integrating several distributed systems onto one bus, having fault containment in a bus system assures that a failure of one system does not affect the others, which is crucial in the aerospace context. The authors point out that FlexRay appears to be a strong field bus candidate for the aerospace domain, provided that a suitable bus guardian is used, and that a careful re-examination of FlexRay's dependability for the aerospace domain is equally necessary to be done.

There are several previous formalizations of the FlexRay physical layer protocol. Beyer et al.[4] gave the first manual deductive correctness proof. Schmaltz[5,6] presented a semi-automatic correctness proof in which the proof obligations are discharged using Isabelle/HOL and the NuSMV model checker. This proof has also been integrated into larger verified system architectures.[12,20] An effort to unify these results into one comprehensive correctness proof of the FlexRay protocol is presented by Müller and Paul.[7] They prove a deviation of oscillators from the ideal rate of up to 0.38% safe, assuming a reliable physical layer. However, Müller and Paul explicitly leave the verification of stronger fault tolerance properties, which includes resilience against an unreliable physical layer, to future work.

In previous work,[8] we presented the first *fully automatic* correctness proof of the FlexRay physical layer protocol. In contrast to the approaches mentioned above, we considered an *unreliable* (and therefore more realistic) physical environment, in which we studied the fault tolerance of the protocol. The model used in the current paper is a refinement of our earlier model. In particular, the new model includes real-time glitches in addition to sample glitches. Our new model was developed based on the modeling principles presented in this paper. Using the new model, we were able to verify additional properties of the FlexRay physical layer protocol, such as the fault tolerance with respect to two glitch-affected samples in a sequence of 88 samples.

VII. Conclusions

We have demonstrated that automatic model checking can be used to determine the impact of changes to the physical setting on the fault tolerance of the FlexRay physical layer protocol. The key challenge in this effort has been to build a model that is sufficiently small to allow for automatic verification and, at the same time, sufficiently precise to describe the intricate real-time interplay of protocol and hardware. In the paper, we have outlined modeling principles that lead to efficiently checkable models, using both clocks and data economically.

Our analysis can be used to determine the application settings in which the protocol will operate robustly. If the hardware parameters fall within the bounds found by our analysis, the protocol is guaranteed to be tolerant with respect to the specified glitch patterns. In addition to these general results, our analysis can easily be repeated for specific parameter values in order to determine the specific error patterns tolerated by the protocol in a given application context.

Of particular interest for aeronautics is our result that the FlexRay protocol is resilient with respect to changes in the variance of the propagation delay, as the longer harnesses used in planes[1] will likely increase the variance compared to ground vehicles. During our analysis, we noticed, however, that increasing the delay variance decreases the maximal tolerable clock drift, and vice versa. In future work, we plan to investigate the impact of varying both at the same time to precisely work out the trade-off.

Acknowledgment. This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

[1]Heller, C. and Reichel, R., "Enabling FlexRay for Avionic Data Buses," *IEEE/AIAA 28th Digital Avionics Systems Conference (DASC)*, 2009.

[2]Paulitsch, M. and Hall, B., "FlexRay in Aerospace and Safety-Sensitive Systems," *IEEE Aerospace and Electronic Systems Magazine*, Vol. 23, No. 9, 2008, pp. 4–13.

[3]FlexRay Consortium, *FlexRay Communications System Electrical Physical Layer Application Notes Version 2.1 Revision B*, 2006.

[4]Beyer, S., Böhm, P., Gerke, M., Hillebrand, M., Rieden, T. I. d., Knapp, S., Leinenbach, D., and Paul, W. J., "Towards the Formal Verification of Lower System Layers in Automotive Systems," *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, IEEE Computer Society, 2005, pp. 317–326.

[5]Schmaltz, J., "A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware," *7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, edited by J. Baumgartner and M. Sheeran, IEEE Press Society, November 11–14 2007, pp. 223–230.

[6]Schmaltz, J., "A Formal Model of Lower System Layers," *Formal Methods in Computer Aided Design (FMCAD'06)*, IEEE Computer Society, 2006, pp. 191–192.

[7]Müller, C. and Paul, W., "Complete Formal Hardware Verification of Interfaces for a FlexRay-Like Bus," *CAV*, edited by G. Gopalakrishnan and S. Qadeer, Vol. 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 633–648.

[8]Gerke, M., Ehlers, R., Finkbeiner, B., and Peter, H.-J., "Model Checking the FlexRay Physical Layer Protocol," *Formal Methods for Industrial Critical Systems (FMICS)*, Vol. 6371 of *Lecture Notes in Computer Science*, Springer-Verlag, 2010, pp. 132–147.

[9]Behrmann, G., David, A., and Larsen, K. G., "A Tutorial on Uppaal," *SFM*, edited by M. Bernardo and F. Corradini, Vol. 3185 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 200–236.

[10]Nangate Inc., *Nangate 45nm Open Cell Library Databook*, 2009.

[11]FlexRay Consortium, *FlexRay Communications System Protocol Specification Version 2.1 Revision A*, 2005.

[12]Alkassar, E., Böhm, P., and Knapp, S., "Formal Correctness of an Automotive Bus Controller Implementation at Gate-Level," *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008)*, edited by B. Kleinjohann, L. Kleinjohann, and W. Wolf, Vol. 271 of *International Federation for Information Processing*, Springer, 2008, pp. 57–67.

[13]Alkassar, E., Böhm, P., and Knapp, S., "Correctness of a Fault-Tolerant Real-Time Scheduler and its Hardware Implementation," *Sixth ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'08)*, IEEE Computer Society, 2008, pp. 175–186.

[14]Alur, R., Courcoubetis, C., and Dill, D. L., "Model-Checking for Real-Time Systems," *LICS*, IEEE Computer Society, 1990, pp. 414–425.

[15]Alur, R. and Dill, D. L., "A Theory of Timed Automata," *Theoretical Computer Science*, Vol. 126, No. 2, 1994, pp. 183–235.

[16]Clarke, E. M., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, 2001.

[17]Gerke, M., Ehlers, R., Finkbeiner, B., and Peter, H.-J., "Automatic Protocol Verification with Parametric Physical Layers," Tech. Rep. 86, SFB/TR 14 AVACS, 2012, ISSN: 1860-9821, www.avacs.org.

[18]Dill, D. L., "Timing Assumptions and Verification of Finite-State Concurrent Systems," *Automatic Verification Methods for Finite State Systems*, edited by J. Sifakis, Vol. 407 of *Lecture Notes in Computer Science*, Springer, 1989, pp. 197–212.

[19]"Real-time Architecture Analysis: A COTS Perspective," *Digital Avionics Systems Conference, 2002*, 2002, pp. 5D4–1 – 5D4–9.

[20]Knapp, S. and Paul, W., "Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification," *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, edited by T. Reps, M. Sagiv, and J. Bauer, Vol. 4444 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 53–81.