Automatic Protocol Verification with Parametric Physical Layers

by

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

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

AVACS Technical Report No. 86

2012

ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Bernd Finkbeiner, Martin Fränzle,
Ernst-Rüdiger Olderog, Andreas Podelski
ATRs (AVACS Technical Reports) are freely downloadable from www.avacs.org

Author(s) contact: Michael Gerke(gerke@cs.uni-saarland.de).

Automatic Protocol Verification with Parametric Physical Layers*

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

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

The FlexRay standard describes a robust communication protocol for distributed components. Originally developed for the automotive industry, FlexRay recently attracted a lot of attention in the aeronautics industry. In this much more sensitive domain, however, some standard assumptions (concerning, e.g., the maximal harness length) easily exceed the limits specified in the original standard.

In this paper, we advocate to use model checking as the key technique to automatically establish the correctness of an asynchronous communication protocol that is parametric in its environment assumptions. For this purpose, we present a general methodology for an efficient modeling of communication protocols with parametric physical layers. As a case study, we apply these guidelines to obtain a model of FlexRay's physical layer protocol based on timed automata. Using the model checker Uppaal, we are able to establish new results that significantly extend the rather conservative assumptions in the standard.

I. Introduction

In this paper, we analyze the FlexRay bus protocol, which stems from the automotive context. 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 non automotive contexts like aeronautics. 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.

We review various recent approaches to verify the FlexRay protocol. Initial efforts were completely manual.[4] However, a manual analysis is both error-prone[5] and expensive. Errors in the proof can be avoided by using (semi-)automated proof checkers such as interactive theorem proving tools.[6, 7] However, the use of such tools still requires a substantial manual effort. Furthermore, the analysis is only valid for contexts that correspond to the assumptions made, so the verification would have to be repeated for every relevant change in the context.

In our own work,[8] we have used the real-time model checker UPPAAL 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,[9] FlexRay tolerates one glitch every four samples. In fact, this tolerance is robust under various variations of the hardware parameters, e.g., clockdrift. While fault tolerance holds for a wide range of hardware configurations, it strongly depends on design parameters like the size of the voting window: for example, the voting window of five samples, specified in the standard, allows for up to one glitch every four samples, while an alternative voting window of three samples would allow for one glitch every three samples. In turn, a voting window of size three would not allow for two glitches next two each other, while a voting window of size five allows for up to two glitches at arbitrary positions, including next to each other, in every sequence of 88 consecutive samples. Different patterns of glitches and different ways to model them are considered, and they influence the robustness of the protocol to variations of hardware parameters.

*This work has been supported by the DFG as part of SFB/TR 14 AVACS

Our verification effort is based on a general hardware model that is parameterized in low-level timing details such as hold times and propagation delays. The key challenge has been to develop a model that is sufficiently small to allow for automatic verification and, at the same time, sufficiently precise to describe the intricate interplay of the bit-clock alignment mechanism with the timing behavior of the underlying asynchronous hardware.

Based on the lessons learned in this verification project, the report outlines a general modeling and analysis methodology for communication protocols that need to function robustly in a range of application contexts and physical environments.

II. Overview

We present guidelines for modeling physical layer protocols using the FlexRay physical layer protocol as an instructive example.

We present a model of the physical layer protocol of the FlexRay coding/decoding unit (CODEC). As illustrated in Figure 1, our model is mainly structured into a model of the protocol and a model of the underlying hardware. The *protocol model*, which is given in Section IV, consists of a sender and a receiver. The hardware model consists of a model of the bus and the jitter and a glitch model, which connects the bus with the receiver.

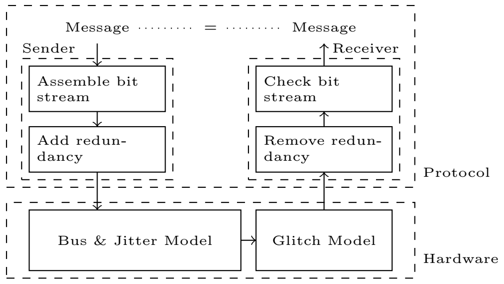


Figure 1. The structure of the model.

We regard the message *frames*, which are obtained from the next-higher FlexRay layer and contain data to be transferred as well as protocol related information, as simple byte strings independent of their format and content and call these *messages* in the following. 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 so-called *samples*), removes the redundancy and transmits the message received to the next-higher layer of the FlexRay protocol. If the received message is not the same as the message sent, the receiver goes into a designated error state.

The *hardware model*, which will be described in Section V, describes the underlying hardware, including the communication bus and the effects of glitches and jitter.

The scenario considered in our model is the reception of a message from a sending CODEC of a FlexRay controller that is directly connected to the receiving CODEC. It is sufficient to consider the scenario of one sending and one receiving *controller*, as the number of receiving controllers does not influence the message transfer process. According to the FlexRay standard [10, Chap. 5], FlexRay uses a time division multiple access (TDMA) scheme, which excludes collisions.[11] The correctness of higher protocol levels and the ability of FlexRay to deal with errors outside the error model are beyond the scope of this work.

A. The Error Model

In our model, we consider two types of erroneous behavior: *glitches* induced by influences from the environment, and *jitter* induced by the asynchronous nature of physical layer protocols. Loss of information due to an error is generally modeled by nondeterministically choosing a value for the affected samples.

Glitches. Environmental interferences can always disturb electronic communication, but smaller disturbances should be compensated in a fault-tolerant physical layer protocol. A sample taken from the bus might have been replaced by an arbitrary value. Simply said, it is possible that something different from the bit that has been sent is received. Such a loss of information on the bus is called a *glitch* [10, Section 3.2.2]. If too many glitches occur, the message might be compromised. However, the FlexRay physical layer protocol compensates for infrequent glitches. We explore two glitch models:

(1) *real-time glitches*, parameterized in the duration of a glitch and the minimum glitch-free period between any pair of glitches;

(2) *sample glitches*, parameterized in the number of glitch-affected samples that may occur in any sequence of a parameterized number of consecutive samples.

Jitter. In addition to glitches, the communication protocol 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. Additionally, the transition between voltage levels takes varying amounts of time. All undesired behavior caused by these effects is called *jitter*, and is described in the model of the bus.

B. Related work

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 that depending on the concrete application topology, replacing FlexRay's default physical layer by an alternative physical layer (and physical layer protocol) such as RS485 should be considered. Since FlexRay's physical layer protocol, that we verify here, is not bound to a particular physical layer implementation, the impact of our results remains untouched when changing the physical layer implementation.

There are several previous formalizations of the FlexRay physical layer protocol. Beyer et al.[4] gave the first manual deductive correctness proof. Schmaltz[5, 12] 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.[11, 13] An effort to unify these results into one comprehensive correctness proof of the FlexRay protocol is presented by Müller and Paul.[6] 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.

Vaandrager et al.[14] use Uppaal[15] to derive invariants of the Biphase Mark physical layer protocol, which are used for semi-automatically proving the formal correctness with the proof assistant PVS. Brown and Pike[16] follow an alternative approach, where they use the verification tool SAL to increase the degree of automation in the correctness proofs of the Biphase Mark and the 8N1 protocols. Unlike the FlexRay physical layer protocol, these protocols are not designed for an unreliable physical environment.

In contrast to all the semi-automatic approaches mentioned above, we presented a *fully automatic* correctness proof of the FlexRay physical layer protocol only using Uppaal in an earlier work.[8] We also considered an *unreliable* (and therefore more realistic) physical environment to study the fault tolerance of the protocol.

C. Contribution of the Paper

Based on our previous work,[8] the present work presents the following extensions.

- **Optimized model of the protocol and the hardware.** In Sections IV and V, we present an optimized version of the model (in terms of complexity) of the protocol and the hardware. The new model allows an easier understanding of the verification effort, especially compared to manual or semi-automatic proofs.

- **Extended error model.** While the old error model only allowed the specification of sample glitches, the new model also allows real-time glitches. Separating the glitch model into its own component allows for easier switching between glitch models. In Section VI, we report on new analysis results providing a much deeper insight into the FlexRay physical layer protocol.

- **General modeling guidelines.** Beyond FlexRay, in Section III.B, we present general guidelines for modeling physical layer protocols with model checking in mind.

III. Modeling the FlexRay physical layer protocol

We present a model of the physical layer protocol of the FlexRay coding/decoding unit (CODEC). The scenario considered in our model is the reception of a message from a sending CODEC of a FlexRay controller that is directly connected to the receiving CODEC. It is sufficient to consider the scenario of one sending and one receiving *controller*, as the number of receiving controllers does not influence the message transfer process. Section V describes a model of the hardware connecting the two CODECs.

This model can easily be adjusted to different hardware implementations. In Section IV we describe a model of the physical layer protocol of FlexRay's CODEC, which aims at providing a certain resilience against error introduced by the hardware connecting the two CODECs. The FlexRay standard[10] specifies two different types of communication: `symbols`, which are used for protocol relevant communication, and message `frames`, which contain a message to be transferred as well as protocol related information. We regard the message *frames*, which contain a message to be transferred as well as protocol related information, as simple bit strings independent of their format and content and call these strings *messages* in the following. As the transfer of `symbols` is less complex, the interesting scenario is the transfer of a message. According to [10, Chapter 5], 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.[11] W.l.o.g., the model considers the transfer of a message in the static segment. The correctness of the higher protocol levels and the ability of FlexRay to deal with errors outside the error model is beyond the scope of this work.

The task of the FlexRay physical layer protocol is to send a message (in this case a so called FlexRay message frame) from a sending FlexRay controller to a receiving one using a bus, i.e., the physical layer that is connecting them. The sender transmits a stream of single bits by putting a stream of bit-cells on the bus, and the receiver samples the value of the bus to reconstruct that stream. To verify properties of the protocol operating on a given physical layer, it is necessary to create a model of the physical layer and the two controllers which can then be model-checked. The model of the physical layer will include an error model describing the effects of interference on the physical layer, i.e., when and how long information on the physical layer can be corrupted in the error scenario under consideration. Such errors are called glitches. The behavior of the physical layer can only be accurately described if a reference to real time is possible. Jitter caused by the asynchronousness of the receiver and the sender also is best described using references to real time. We thus choose to model the system as a network of timed automata. This allows us to introduce clocks, i.e., stop watches that run at a uniform speed and can be reset to start again from 0 time units, providing the possibility to reference real time.

A. Timed Automata

We assume familiarity with extended timed automata and refer the reader to the UPPAAL tutorial[15] for more background. Extended timed automata are derived from the classical timed automata model by Alur and Dill,[17] but introduce features like integer variables and synchronization that allows modeling real-time systems in a concise manner. Figure 2 gives a brief overview of the syntax. Each automaton consists of a set of locations, representing discrete control points, which can be labeled with *invariants* over clock variables indicating the condition under which the system can stay in that location. Transitions can be labeled with *synchronization channels* over which a sender (identified by "!") can synchronize with a receiver (identified by "?") to take a joint transition. Also, each transition can have an *update expression* to set clock or integer variables, and a *guard* determining its enabledness. To improve the readability of complex models, we split them into segments using the auxiliary syntax shown in Figure 3.

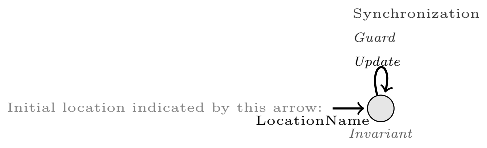
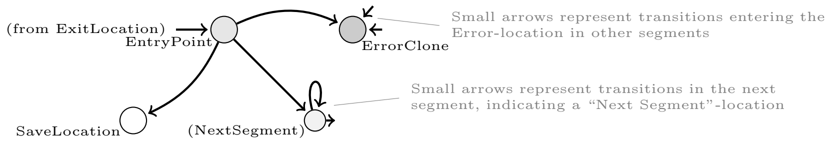


Figure 2. Syntax of locations and transitions



(from ExitLocation)
EntryPoint
ErrorClone
Small arrows represent transitions entering the Error-location in other segments
Small arrows represent transitions in the next segment, indicating a "Next Segment"-location
SaveLocation
(NextSegment)

Figure 3. Syntax used to describe splitted automata

- **NextSegment** is a location that represents the parts of the automaton reachable by the incoming transitions that are described in the next segment. The location is marked by gray color, smaller size and the name of the **EntryPoint** of the next segment in brackets.

- **EntryPoint** is the location entered in this segment by entering the **NextSegment** location of the previous segment.
- **ErrorClone** is a location indicating an error state. It is reachable from various segments of the automaton. Incoming transitions from other segments are indicated by small arrows.
- **SaveLocation** is a location from which no error state can be reached. The location is left white.

In the example shown in Figure 3, the location **EntryPoint** is also the "ExitLocation" of this segment, as there is a transition from **EntryPoint** to **NextSegment**.

B. Design Principles

Several principles guided the design of the model. We first give an overview list and then discuss the principles in more detail.

- Separate the protocol into *components*.
- Reduce the number of clocks by *partitioning the behavior* with respect to the oscillator triggering it:
  - The bus behavior depends on the sender's clock.
  - The sampler behavior depends on the receiver's clock.
- *Ignore constant delays* as communication is one-way only.
- Reduce interval intersection checks to checking membership in an interval: Choose an arbitrary delay on the wire and *extend unstable intervals* by the sampling interval.
- Abstract from time where possible by *replacing time with order* when only the order of events is relevant.
  - Make local computations instantaneous in one clock cycle.
  - Trigger sequential components using a chain of clock tick signals.
- Abstract from message content and concrete timing: The sender produces a *nondeterministic message*, formats the stream as specified by the protocol, and exhibits any possible timing behavior.
- Just two dedicated end locations: The receiver checks the message by testing (1) if it has the specified format, and (2) if the message content is preserved. If either of the tests does not succeed, the receiver enters a *dedicated error location.* Otherwise, a safe location is entered.
- Using *nondeterminism keeps the memory consumption of the model checking process small* by avoiding counting.
  - Send a nondeterministic number of bytes instead of counting bytes.
  - Forget the message content except for nondeterministically chosen bits. For these bits, memorize at most one bit in a byte, and forget it as soon as possible.
- Use a *parameterized hardware* model for easy adoption to different hardware.

Components. It is reasonable to model the overall scenario as a network of component automata, as it allows changing parts of the model without having to change the complete model. For our FlexRay physical layer protocol model, we have three main parts: the sender, the bus, and the receiver. Each part in turn consists of one or more timed automata. In the following, the name of the corresponding component is given in brackets. The sender generates a message stream and puts it on the bus (sender control). The receiver has several components: it samples the bus (bus-sampler), flattens the sample stream (voting), selects bits from the flattened stream (strobing), and checks whether the stream has the correct format (receiver control). In order to verify the protocol, the receiver control automaton also checks whether the message was correctly transmitted. As the receiver and the sender are implemented as hardware, all their steps are triggered by their respective oscillators (sender clock, receiver clock). The bus is a physical medium that needs time to change its value (bus). The jitter model is co-located with the bus. Glitches (glitch) are introduced between the bus and the sampler.

Partitioning Behavior. To keep the model checking process of systems with timed behavior efficient, it is desirable to minimize the number of clocks in the model. The time-related behavior in the scenario to be modeled can be partitioned into behavior triggered by the sender and behavior triggered by the receiver. Both controllers have an oscillator whose respective edges trigger the steps in the protocol. These oscillators are not perfect and the time between two consecutive edges can deviate from the standard up to a fixed percentage, causing drift between the two oscillators. In case of the sample-glitch model, only two clocks are needed, one for each oscillator. All other timing-related behavior can only refer to one of these clocks, because the behavior is triggered by an edge of one of the oscillators. In case of the real-time-glitch model, a third clock is needed as the glitches can occur independent of the protocol's operation and thus need an independent source of time. The behavior of the bus can be described in terms of the sender's clock, as the sender triggers the changes in the value of the bus. Of course, using the sender's clock for the bus limits the description of the behavior of the bus to behavior faster than one clock cycle, as the senders clock will be reset at the beginning of the next clock cycle. More precisely, we are limited to describing a bus were the sum of its delay variance, its maximal duration of an undefined bus state of neither 0 nor 1 during a change of the bus state, and the setup time of the bus-samplers register is smaller than the minimal duration of a clock cycle. As FlexRay operates at 80 MHz, this requirement is easily fulfilled by standard hardware. The behavior of the bus-sampler can be described in terms of the receiver's clock, as the receiver's clock edge determines the position of the sampling interval. Of course, this limits the description of the bus-sampler's behavior to behavior faster than one clock cycle, which is easily achieved, as a sampling interval of a register does have to fit into a clock cycle in order to obtain standard hardware behavior anyway.

Ignore Constant Delays. As the information flows from the sender via the bus to the receiver, and there is no feedback from the receiver to the sender of the bus, all constant delays to the flow of the information can be ignored, as they will be invisible to the receiver anyway. Differences between the constant delays relating two different controllers in a sending or receiving role are handled by the upper layers of the FlexRay protocol, using a time division multiple access scheme. Thus the scenario for the analysis of the FlexRay physical layer protocol can be simplified to one sender and one receiver, which allows to drop all constant delays. Constant delays inside the receiver model can also be dropped if the information flow is strictly in one direction. The only feedback inside the controller is the signal enabling the synchronization of the strobing mechanism, which is sent by the receiver control to the strobing component. However, this signal is sent after receiving the next-to-last bit before it is used, so a correct implementation of the receiving controller can easily make sure that the signal arrives in time, allowing us to ignore all constant delays inside the receiver as well.

Extend unstable interval. Information on the bus is usually not really digital, but analogous. Thus, for certain intervals during a non-instantaneous change of the bus' value, an unstable value will be on the bus. As sampling from the bus usually is also not instantaneous, a value on the bus is required to be stable in certain intervals to ensure that a stable value is sampled. One of those intervals can be extended by the size of the other interval in the model, removing the need to actually check an interval intersection. Furthermore, as the constant delay of the bus is irrelevant, it can be chosen such that checking the intersection of the two intervals boils down to simply testing if a clock tick of the receiver occurs in a certain period of time after a tick of the senders clock.

Replacing Time with Order. To reduce the number of clocks in our model, and thus keep model checking efficient, we replace *time with order* where possible. In particular, the local behavior of the protocol in the sender and in the receiver can be described in discrete terms. We assume each controller to work as specified and focus solely on the interaction between the controllers and the physical layer. This narrows the focus of the modeling effort to the interaction of the controllers with the bus and to the drift between the two controller's oscillators.[6] In turn, this allows us to model the preparation of the message stream and the handling of the received sample stream as stepwise processes, where the steps are triggered by the local oscillators and are instantaneous. So all the computations of one logical step in the protocol are modeled as happening in one clock cycle, at the exact time of the clock edge, with no time passing during these computations.

The order in which the data is processed by the different components is important and is fixed using a chain of components. The oscillator components generate a chain of synchronization signals for each clock edge, so each component gets its clock edge signal on its dedicated handshake synchronization channel, triggering the components in the correct order. In a setting in which the events are ordered, we have two different forms of communications between components at our disposal. For asynchronous communication, we can use shared variables, as the order of accesses to the variables is fixed by the chaining of clock edge signals. For synchronous communication, we can on the other hand directly use the handshake synchronization channels of timed automata. To avoid blocking in synchronous computation, receiving components are designed to have an enabled transition labeled with the respective communication channel.

Nondeterministic message. For the scope of verification, we have to check that FlexRay's physical layer protocol ensures that all possible messages can be received correctly with any timing behavior conforming to the protocol. To keep the process efficient, we abstract from the possible messages produced by the higher layers of the FlexRay protocol, and check that *any* message is delivered correctly. This way, we keep the sender process model small, as we are not concerned with the message composition this way. We can do so by letting the sender non-deterministically guess the message. A core component of FlexRay's physical layer protocol, namely the assembly of the message stream, still needs to be modeled, as it is crucial for error correction on the receiver side.

Dedicated error location. To check for possible errors, the receiver control component of the model contains an error location, which cannot be left again. The receiver control checks whether the format of the message fits [10, Sect. 3.2.1.1], if the format is not as expected, the error location is entered. In order to analyze the physical layer, we let receiver control also check whether the received message is the one that was sent, and enter the error location if a message bit is flipped. The absence of transmission errors can thus be checked by determining the reachability of the error location. For model checking, this alleviates us from the problem of encoding the complicated correctness notion of a FlexRay physical layer message transfer explicitly into some form of temporal logic property.

After the reception of a FlexRay message stream, the receiver controller enters a safe location, which cannot be left again. To speed up the model-checking process, both the error location and the safe location cause a deadlock,i.e., time stops after one of the locations has been entered, limiting the exploration of states that cannot reach the error location when the safe location has been entered, or cannot reach some other receiver controller location when the error location was entered. Furthermore, this allows to check for absence of transmission errors and the absence of other deadlocks by checking for the reachability of deadlocks outside of the safe location.

Nondeterminism Keeps the Memory Consumption of the Model Checking Process Small. There are more than 10^{600} different possible flexray messages, as each message can contain up to 262 bytes payload, so checking all possible messages cannot be achieved with brute force. It is a huge simplification to abstract from the number of bytes. Our model of flexray will use a message with a nondeterministic number of bytes. This allows us to stop counting the bytes: if the bus and the error model and the drift are the same, the model can only distinguish between being at a certain position within a message byte, or being at a certain position before or after the reception of message bytes. Thus, when all possible timing related combinations are explored, there is no additional discrete complexity due to the number of bytes, and there are only 256 different combinations of bits in a byte. Moreover, results achieved using this abstraction are valid for messages with any number of bytes.

The discrete complexity is further reduced by nondeterministically memorizing just one bit from a byte to check if this bit was received correctly. If there was a reachable transmission error, one of the affected bits could be memorized and thus this error could be seen, making the error location reachable. This abstraction allows to 'forget' at least 7 bits of a byte after they has been sent, further reducing the discrete complexity.

Parameterized Hardware. As the behavior of the hardware is described in terms of real time, the model of the hardware is kept parameterized. This allows to change the assumptions on the performance of the hardware by just changing a handful of parameters, making the re-verification of properties of the FlexRay physical layer protocol on a changed hardware platform a 'push button' procedure. Furthermore, certain hardware model parameters can be explored and their effect on the protocol's properties be analyzed, yielding reliable requirements for the hardware.

C. Modeling Communication Protocols

It is important to identify the relevance of real time behavior to the protocol. Usually this will be restricted to describing continuous behavior related to the physical characteristics of the communication medium and the drift between the oscillators. All the other aspects can be reduced to discrete behavior. As complex discrete behavior is best analyzed with tools that usually do not handle real-time well, it is wise to identify the smallest unit that can be verified independently of the rest of the protocol and contains all or most of the real time related behavior. The physical layer protocol or the clock synchronization and TDMA scheme are such units. The rest of the protocol can be analyzed using results obtained by analyzing these smaller units independently.

For FlexRay, the clock synchronization and TDMA scheme were verified manually in.[18] The FlexRay physical layer protocol defied precise manual analysis[4] and is thus verified using model-checking in[8] and this work, using the correctness of the TDMA scheme as an assumption.

For modeling physical layer protocols, several guidelines have been laid out above. The essence is a focus on the real-time related behavior that reduces the complexity of the discrete behavior through the use of nondeterminism, in the most general 'one sender – one receiver' scenario.

IV. The Protocol Model

We model a scenario in with one sender and one receiver.

A. The Sender

The Bit Stream Format. A message is transmitted as a structured stream [10, Sect. 3.2.1.1] of bit cells as shown in Fig. 4. In every bit cell, the bit value is held for eight clock cycles (not shown in the figure).

The start of the stream is the so-called *transmission start sequence* (TSS), which consists of a sequence of low bits. The length of the sequence is fixed for the cluster and may vary from 3 to 15 bits see [10, Sections B.2.1]. It precedes every transmission.

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. The high to low transition in the middle of the BSS is used as a trigger for the bit clock alignment.

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.

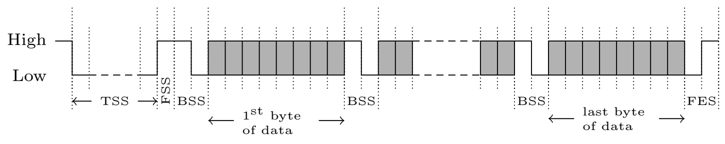


Figure 4. Format of a message bit stream.

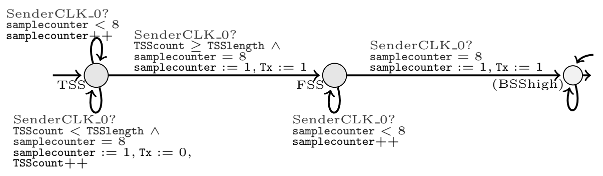


Figure 5. Model of the start of the transmission.

Sending the Bit Stream. The sending of the bit stream is modeled by the automaton shown in Fig. 5. The message is generated nondeterministically as shown in Fig. 6. Also, the sender nondeterministically determines whether a particular bit should be verified by the receiver. In this case, the value of the chosen bit is stored in `savedTx` and its offset within the current byte is stored in `savedindex`[a]. In our model, the variable `End` is used to signal to the receiver that that the bit stream is about to end (shown in Fig. 7), as we do not have a predefined number of message bytes.

B. The Receiver

[a]The initial value `savedindex` = 8 means "no bit to test".

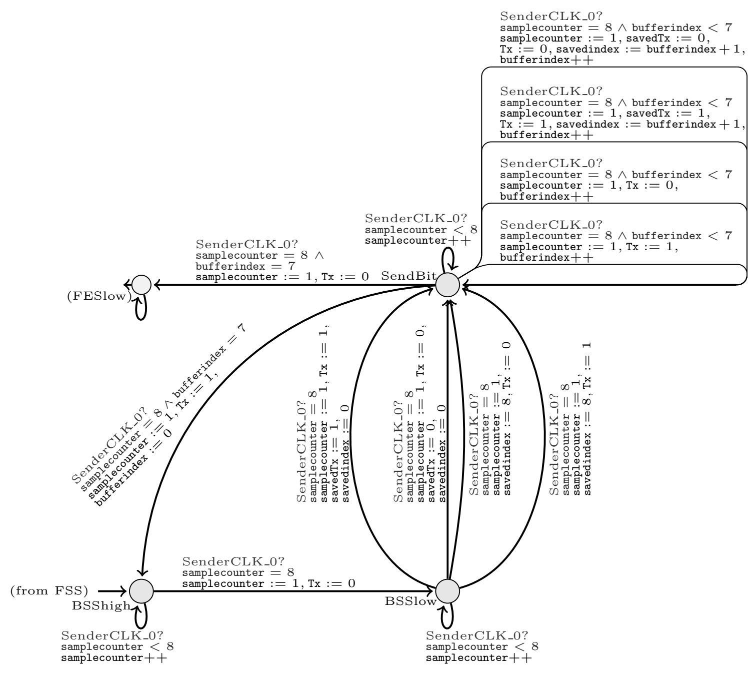


Figure 6. Model of the transmission of the message bytes.

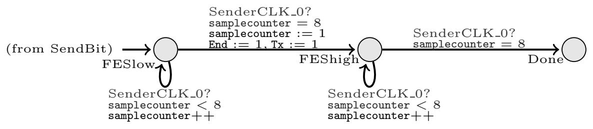


Figure 7. Model of the end of the transmission.

Voting. In order to reconstruct the bit stream sent by the sender, the receiver takes several samples from each bit cell. The five most recent samples always form the so-called *voting window*.[b] In each clock cycle, a *voted value*, i.e., the value of the majority of the five samples in the voting window, is computed from these. As the size of the voting window is odd, there will always be a clear majority.

Our receiver model always maintains the respective previous four samples and the sample obtained in the current clock cycle. The variable `window0` always holds the newest value. As the third step in every receiver clock cycle, the values of the `window` variables are shifted accordingly, as shown in Fig. 8. If the majority of the `window` variables contains a 1, `VV` is set to 1, and to 0 otherwise. The respective previous value of `VV` is stored in `OldVV`, as the first step of each receiver clock cycle.

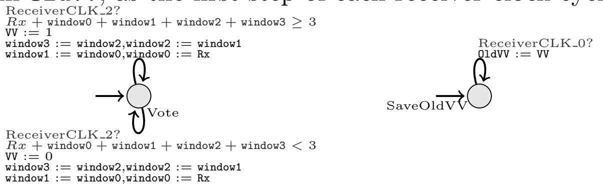

ReceiverCLK_2?
Rx + window0 + window1 + window2 + window3 ≥ 3
VV := 1
window3 := window2,window2 := window1
window1 := window0,window0 := Rx
Vote
ReceiverCLK_2?
Rx + window0 + window1 + window2 + window3 < 3
VV := 0
window3 := window2,window2 := window1
window1 := window0,window0 := Rx
ReceiverCLK_0?
OldVV := VV
SaveOldVV

Figure 8. Model of the voting process.

Strobing. From each bit cell, only one voted value is used to reassemble the bit stream. To avoid choosing values that are affected by glitches, the fifth voted value (computed from samples from the middle of the bit cell) is taken as the so-called *strobed value.*

Bit Clock Alignment. In order to identify the (approximate) boundaries of the bit cells and thus the strobed values, the receiver keeps the variable `strobecounter` synchronized to the stream of received voted values.

[b]According to the FlexRay standard [10, Sect. 3.2.6], one sample is taken in one *sample clock period*, which is derived "from the oscillator clock period directly or by means of division or multiplication". Here, a *sample clock period* of one clock cycle is assumed in accordance with.[4, 5, 11–13]

The bit clock alignment mechanism makes use of the bit stream format. At the beginning of the transmission and during the *byte start sequences*, the first transition of the voted value from high to low is detected and `strobecounter` is reset to 2 for the next voted value. Thus, the second recognized voted value of the bit cell is considered the second voted value of the cell.

As shown in Fig. 9, `strobecounter` has no default value, but is initialized nondeterministically. When the new voted value, `VV`, is 0 and the voted value from the cycle before, `OldVV`, is 1, and `EnableSyncEdgeDetect` enables the bit clock alignment mechanism, `strobecounter` is reset to 2, as the received 0 is the first bit of the new bit cell, and the bit clock alignment mechanism is deactivated using `EnableSyncEdgeDetect`. This condition is checked in the 4th step of every receiver clock cycle.

When `strobecounter` has a value of 5 and `strobecounter` is not reset, `VV` is chosen as the strobed bit in the 4th step of the respective receiver clock cycle. The channels `Strobed_1` and `Strobed_0` communicate the value of the new strobed bit.

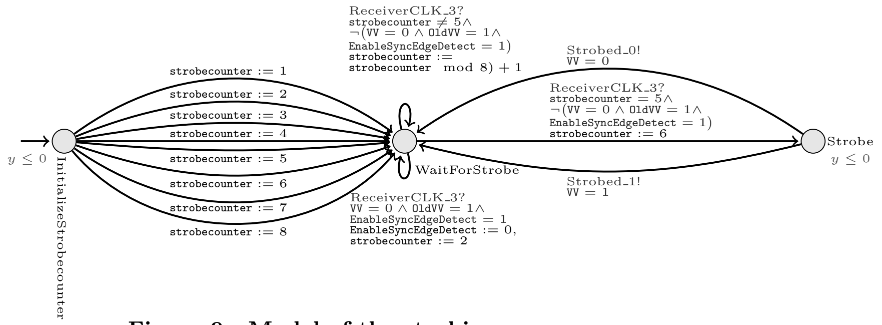


Figure 9. Model of the strobing process.

Receiving the Bit Stream. When channels `Strobed_1` and `Strobed_0` signal that a new value has been strobed, the receiver checks if it is consistent with the expected format of the bit stream, in the last step of the respective receiver clock cycle, as shown in Fig. 10. As soon as a received value is not the expected one, the error state `DECerr` is entered.

The received TSS is accepted if it contains at least `TSSmin` bits. A further bit of the TSS is accepted if not more than `TSSmax` bits have been received before.

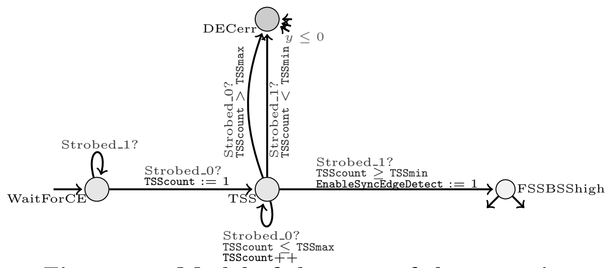


Figure 10. Model of the start of the reception.

During the reception of the TSS or after the reception of a message byte, the variable `EnableSyncEdgeDetect` is used, as shown in Fig. 11, to enable the bit clock alignment mechanism. During the reception of a message byte, the number of bits received so far within this byte is counted using variable `bufferindex`. When `savedindex` indicates that the current message bit is to be verified, the strobed bit is compared to `savedTx`. The variable `End` is checked to prohibit entering the location `Done` too early.

Note that no time will pass after the end of the scenario, i.e., when either the location `Done` has been safely entered, or an error moved the automaton into `DECerr`.

V. The Hardware Model

In FlexRay networks, each controller has a local *oscillator* that clocks all local circuits. The individual controllers run asynchronously and communicate via a shared *bus*. In our model, we use *registers* (standard circuits used to persist values) to simulate the low-level timing behavior of transmitting bit values from sender to receiver.

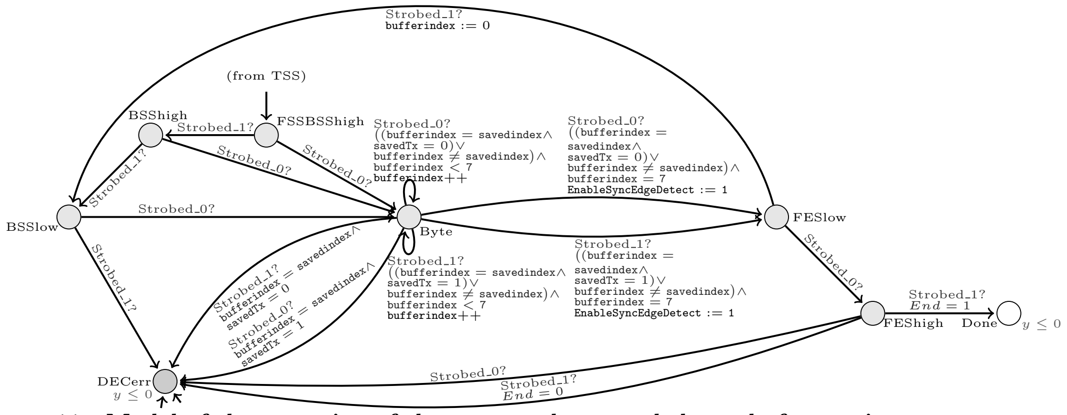


Figure 11. Model of the reception of the message bytes and the end of reception.

Figure 12 gives an overview. The sender begins a transmission of a bit by storing its value in a register `Tx`. The bus content is represented as the output of register `Tx`, which is connected to a register `Rx` on the receiver's side. Following,[4,5,8,11–13] as proposed by,[19] we forward the output of register `Rx` through a consecutive register `Rxx` to suppress metastability problems. This adds a delay of one clock cycle and resolves an unstable value in `Rx` to either 1 or 0 in `Rxx`. However, as we can ignore constant delays, we do not model `Rxx` but instead immediately resolve an unstable register `Rx` nondeterministically to 1 or 0.

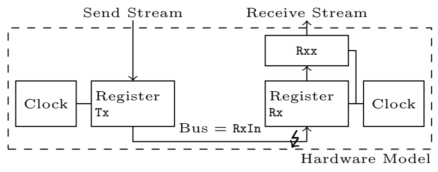


Figure 12. Overview of the hardware sub-architecture.

A. Oscillators

We model the local oscillators of the sender and the receiver as automata that emit *tick*-events (SenderCLK and ReceiverCLK) which, in turn, are received by other automata modeling connected circuits. To allow us to trigger the automata in a specific order, the SenderCLK event is signaled first on channel `SenderCLK_0` and then on channel `SenderCLK_1`, with no time allowed to pass between the activation of the two channels. The ReceiverCLK event is signaled on 4 channels in a similar manner to trigger automata in a chained order, allowing to partition the model of the computations in a receivers clock cycle into 4 consecutive steps.

According to the specification, distributed oscillators may deviate from the standard rate up to a certain bound [10, Appendix A.1]. Furthermore, as these oscillators are not started at the same time, their periods can be shifted arbitrarily. This is modeled by not specifying a minimum length for the first cycle of the receiver's oscillator in Fig. 13. Here, x and y are continuous-valued clock variables.

In our model, we parametrize the length of an ideal clock cycle (which is the same for each controller) by `CYCLE`. To model the deviation, we use a parameter `DEVIATION`. This gives us a lower and an upper bound for tick-events:

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

B. Registers

Following the setting of,[4,5,8,11–13] we assume a *register semantics* to model the timing behavior of the bus which connects the sender and the receiver. This model is easily adaptable to other implementations, as properties like transmission delay and sampling interval will can describe most implementations, just the parameters that describe them have to be changed accordingly. Before we explain the actual transmission of bit values via the bus, we first give a general description of the low-level timing behavior of registers.

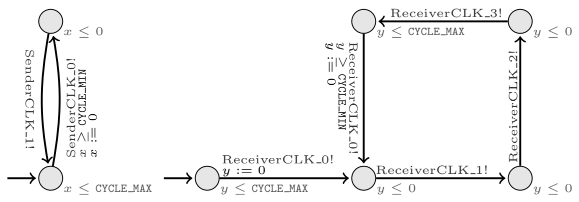


Figure 13. Oscillators for sender and receiver.

Register Semantics. The behavior of a particular register hardware is described in terms of the following parameters:

- SETUP (HOLD) is the *setup (hold) time*, i.e., the time that the value on the input of a register is required to be stable before (after) the occurrence of a tick-event;
- PMIN (PMAX, where PMIN $\leq$ PMAX), is the *minimal (maximal) propagation delay*, i.e., the minimal (maximal) time after which a register changes its output to an undefined value (to the new value) after the occurrence of a tick-event.

The register content represents a particular Boolean value using voltage levels: A value below a certain voltage level is considered as 0 and a voltage above a certain level is considered as 1. However, there is a certain range of voltage levels between the two thresholds that cannot be interpreted as any Boolean value.

Fig. 14 illustrates a scenario in which first a register's input I and, after a tick-event, also its output R changes from X to Y. Here, τ refers to the time between two consecutive tick events and Ω indicates an undefined state of the register's output.

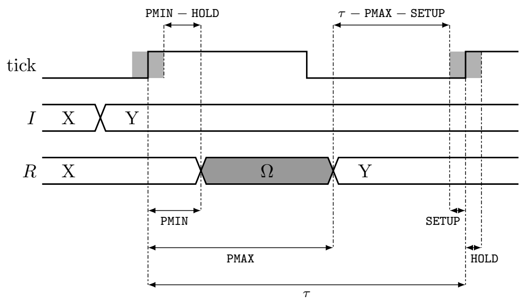


Figure 14. Value change scenario of a register R.

We assume that the unknown value is stable before $\tau -$ SETUP, i.e., before it could violate the setup times of connected registers in the next cycle. In the FlexRay context, for a particular controller, all inputs of registers are connected to circuits that use the same oscillator as the registers. Hence, according to [20, Sect. 5.2], we assume that all local inputs are stable.

More generally, let $R(t)$ and $I(t)$ be a register's output and input at a point of time t, respectively, and let T be the point of time of a tick event, $t_{old} = T - \tau + \texttt{PMAX}$, and $t_{next} = T + \tau + \texttt{PMIN}$. Furthermore, let there be a point of time t' where the register's input changes, i.e., $T - \texttt{SETUP} \leq t' \leq T + \texttt{HOLD}$ such that $I(t') \neq R(t_{old})$. Then, the output of a register at time t, $t_{old} \leq t \leq t_{next}$, is formally defined as

$$R(t) = \begin{cases} R(t_{old}) & t_{old} \leq t \leq T + \texttt{PMIN}, \\ \Omega & T + \texttt{PMIN} < t < T + \texttt{PMAX}, \\ X & T + \texttt{PMAX} \leq t \leq t_{next}, \end{cases}$$

$$\text{where} \qquad X = \begin{cases} I(T) & \text{if } \forall t'.(T - \texttt{SETUP} \leq t' \leq T + \texttt{HOLD}) \Rightarrow (I(t') = I(T)), \\ \Omega & \text{otherwise.} \end{cases}$$

Model of the Bus. Figure 15 shows the automaton modeling the transmission of a bit value according to the register semantics defined in the beginning of this section. Recall the structure of the hardware sub-architecture shown in Fig. 12. In our model, we represent register Tx's content by a variable Tx, and the value of the bus by a variable volt. As the bus value is high whenever it is idle [10, Sect. 3.2.4], volt is initialized with 1.

At every tick of the sender's clock, the variable Tx is checked in the second step: if the sender is still writing the same value to the bus, nothing changes, but if the sender tries to write a different value to the bus, volt changes its value. Here, we represent an undefined bus content by a value of 2 for RxIn, and use the parameters HLMIN, HLMAX, LHMIN, and LHMAX to model the delays induced by the hardware: As a conservative approximation, we assume

$$\mathtt{HLMIN} = \mathtt{LHMIN} = \mathtt{PMIN} \quad \text{and} \quad \mathtt{HLMAX} = \mathtt{LHMAX} = \mathtt{PMAX}.$$

The constant delay added by the wire can be ignored as it is constant and there is no feedback to the sender, but it can also be set to any arbitrary value. We chose $\mathtt{WIREDELAY} = \mathtt{HOLD} - \mathtt{PMIN}$. Remember that the register requires a stable input in the sampling interval $[T_r - \mathtt{SETUP}, T_r + \mathtt{HOLD}]$ around T_r, where T_r is the time of the receiver's clock's tick event. If a new value is put on the bus with a tick of the sender's clock at time T_s, the bus will be unstable at the receivers side during the interval $[T_s + \mathtt{WIREDELAY} + \mathtt{PMIN}, T_s + \mathtt{WIREDELAY} + \mathtt{PMAX}]$. Thus, we want:

$$T_s + \mathtt{PMAX} + \mathtt{WIREDELAY} < T_r - \mathtt{SETUP} \Leftrightarrow T_s + \mathtt{PMAX} + \mathtt{WIREDELAY} + \mathtt{SETUP} < T_r$$

and we also want:

$$T_s + \mathtt{PMIN} + \mathtt{WIREDELAY} > T_r + \mathtt{HOLD} \Leftrightarrow T_s + \mathtt{PMIN} + \mathtt{HOLD} - \mathtt{PMIN} > T_r + \mathtt{HOLD} \Leftrightarrow T_s > T_r$$

So, an unstable value will be sampled when a receiver's clock's tick event occurs in the interval $[T_s, T_s + \mathtt{PMAX} + \mathtt{WIREDELAY} + \mathtt{SETUP}]$.

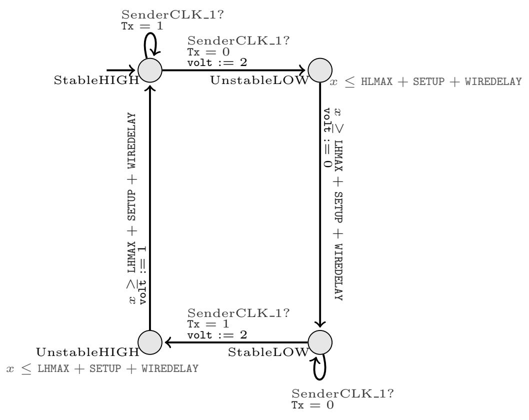


Figure 15. Simple model of the bus.

Model of the Receiving Register. Figure 16 shows the automata modeling the sampling process on the receiver's side. The receiver samples a value from the bus using the register Rx. Assuming a perfect physical layer, the register Rx's input, represented by a variable RxIn, would be equal to the value of the variable volt. However, as we assume an unreliable physical layer, volt is used to set the value of RxIn by the glitch model.

If the value of RxIn is stable, Rx is updated to RxIn. If the value of RxIn is unstable, RxIn = 2, Rx is nondeterministically assigned 1 or 0, as metastability is suppressed as described above. This has the additional advantage of subsuming all cases with shorter unstable periods, as all cases where the correct value is recognized inside the period here modeled as being unstable are subsumed by the case where the unstable value is nondeterministically resolved to the correct value. Note that the the stability requirement during the sampling interval of the register is moved to the bus model by lengthening the bus instability period as described above.

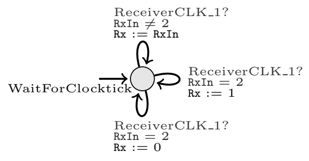


Figure 16. Simple model of the sampling process.

C. Glitch Model

As the bus is not assumed to be a perfect medium, information on the bus may be destroyed by glitches. As these glitches are caused by external interference, we will only analyze them independently of the FlexRay protocol. The position of the glitches in the message stream on the bus is thus not measured in terms of what part of the stream is affected, but only relative to the other glitches. The analysis focuses on patterns of glitches. To describe these patterns, we have two approaches: We can either (1) give the maximal duration of a glitch and the minimum glitch free time after a glitch, or we can (2) consider the number of received samples that can be compromised by a glitch in a certain sequence of consecutive samples. Approach 1, *real-time-glitches*, introduces yet another source of real time behavior, and is well suited to describe glitches. Approach 2, *sample-glitches*, adds only discrete behavior, and is well suited to describe the effects of glitches on the protocol. We will explore both approaches separately, but the choice only affects the glitch model.

Real-time-glitches. We introduce a third clock, a, to measure the duration of a glitch. If a glitch occurs, the bus will be unstable for $\texttt{ERRDUR}$ time units. Sampling a value from the bus will be unaffected by the glitch if it occurs at least $\texttt{ERRDUR} + \texttt{SETUP}$ time units after the glitch, due to the requirement of a stable bus value during the sampling interval. Similarly, sampling a value from the bus before the glitch will be unaffected by the glitch if it occurs at least `HOLD` time units before the glitch. Thus, the automaton shown in Figure 17 moves to the location `Glitch` `HOLD` time units before the occurrence of the glitch, and leaves it $\texttt{ERRDUR} + \texttt{SETUP}$ time units after the occurrence of the glitch.

Clock a is reused after the glitch to measure the period after the glitch in which no other glitch can occur, defined by $\texttt{ERRDIST}_t$. As the clock is reset $\texttt{ERRDUR} + \texttt{SETUP}$ time units after the occurrence of the last glitch, there will be no glitch for the next $\texttt{ERRDIST}_t - \texttt{SETUP}$ time units, and thus no move to the location `Glitch` for the next $\texttt{ERRDIST}_t - \texttt{SETUP} - \texttt{HOLD}$ time units. The boolean variable `A`, initialized to *true*, is used to allow the occurrence of the first glitch at an arbitrary position.

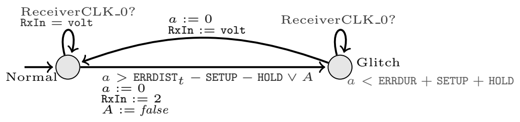


Figure 17. Model of real-time glitches: At most one glitch in $\texttt{ERRDIST}_t$.

This model allows a very natural description of the glitch pattern as there are just two parameters: how long a glitch can last, and how long it takes after a glitch until another glitch may happen. Note that this allows to analyze glitches that affect two consecutive samples, but the affected samples will only be next to each other.

However, analyzing the effects of two short glitches not next to each other during an otherwise glitch-free period *independently of the samples* needs the introduction of a fourth clock, b, and one more boolean variable, `B`, which is initialized to *true*. The automaton shown in Figure 17 is then replaced by the automaton shown in Figure 18. This automaton works similar to the one it replaces, the difference being that it has two glitch locations which are independent of each other, but work the same way with b replacing a and `B` replacing `A` in one of the glitch locations.

If more than two samples in a voting window can be affected by a glitch, any voted value can be changed (if the voting window has size 5), because the glitch can cause a new majority in the voting window, rendering further analysis unnecessary.

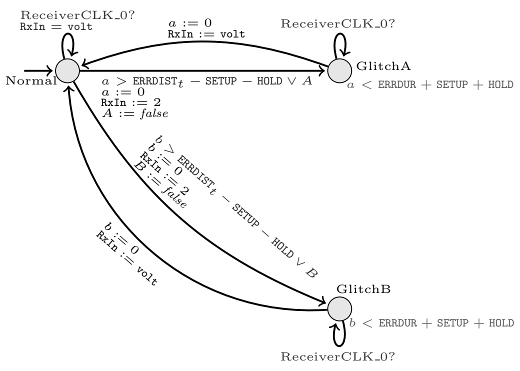


Figure 18. Model of real-time glitches: At most 2 glitches in $\mathtt{ERRDIST}_t$.

Sample glitches. As the resilience of the protocol to glitches does depend on the samples that are affected by the glitch, we can also abstract from the duration of a glitch and just model the glitch in terms of affected samples, as done in.[8] As more than 2 glitch-affected samples in a voting window of size 5 are not interesting (they positively can change the voted value), there are only two interesting scenarios: one glitch-affected sample, or two glitch-affected samples, both in some sequence of consecutive samples, respectively. The second case can be divided into two sub-scenarios for a more detailed analysis: two glitch-affected samples next to each other, or two independent glitch-affected samples.

If just one sample can be glitch-affected in a sequence of consecutive samples, this can be modeled as shown in Figure 19 by nondeterministically choosing some sample to be glitch-affected and then let the next $\mathtt{ERRDIST}_s$ samples not be affected by a glitch. The not glitch-affected samples are counted using the variable `Lerr`.

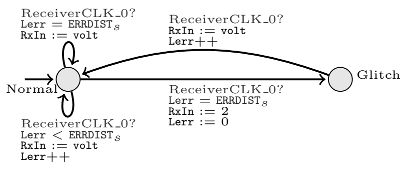


Figure 19. Model of sample glitches: at most 1 glitch-affected sample in $\mathtt{ERRDIST}_s + 1$ consecutive samples.

The automaton from Figure 19 can be extended as shown in Figure 20 to model two glitch-affected samples next to each other where the next $\mathtt{ERRDIST}_s$ samples will not be affected by a glitch.

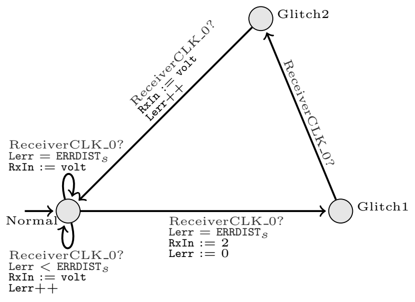


Figure 20. Model of sample glitches: at most 2 glitch-affected samples, next to each other, in every sequence of $\mathtt{ERRDIST}_s + 2$ consecutive samples.

If two arbitrary samples can be glitch-affected in a sequence of consecutive samples, this is modeled by nondeterministically choosing an affected sample, then nondeterministically choosing another one, making sure that at least $\texttt{ERRDIST}_s$ samples have been received since the next-to-last glitch-affected sample before a third sample may be nondeterministically chosen to be glitch-affected. The number of samples since the last glitch-affected one are counted using the variable `Lerr`, and the number of samples since the next-to-last glitch-affected sample are counted using the variable `NLerr`.

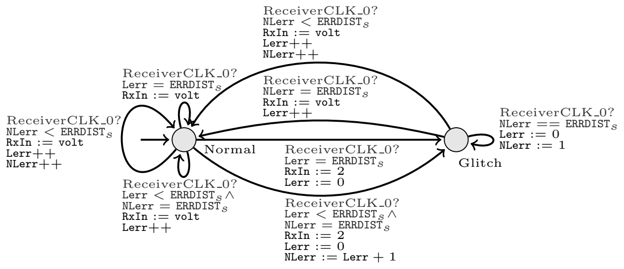


Figure 21. Model of sample glitches: at most 2 glitch-affected sample in $\texttt{ERRDIST}_s + 1$ consecutive samples.

Replacing the automaton from Figure 19 with the automaton shown in Figure 21 allows to check for two independent glitches in a consecutive sequence of samples.

VI. Analysis of the FlexRay Physical Layer Protocol

With Uppaal version 4.1.4-64bit, we verified that a voting window of size five allows for up two glitch-affected samples at arbitrary positions 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 needed 210 minutes using 83GiB of memory to verify this property. If the glitch-affected samples were next to each other, it took only 262 seconds and 2.1GiB of memory to verify this property. The property that up to one glitch-affected sample in every sequence of four consecutive samples is allowed for, which was already shown in,[8] took 21 seconds to verify in the sample-glitch model, using 207MiB of memory. While tools specializing on handling discrete complexity should be able to handle this verification task more space-efficiently, we created a variant of the model better suited to Uppaal: we introduced a third (and forth) clock and created the real-time-glitch model.

The third clock was used to measure the duration of a glitch and the glitch free period after the glitch, yielding a more intuitive error model. Note that we thus decoupled the occurrence of a glitch from the number of samples affected. This allowed us to analyze the relationship between maximal glitch duration and the minimal glitch-free time between two glitches. This model could be analyzed with Uppaal version 4.1.4-64bit in just 30 seconds using only 204MiB for the short glitch (affects at most one sample in the sequence) version, and in just 43 seconds using only 311MiB for the long glitch (affects at most consecutive 2 samples in the sequence) version.

The results from the real-time-glitch model confirmed our findings from the sample glitch model: a glitch of less than $2 * \texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$ and a glitch-free period of more than $86 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$ in between two glitches are allowed for. So, 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 two each other in a sequence of 88 consecutive samples are allowed for. If the glitch is shorter than $1*\texttt{CYCLE\_MIN}-\texttt{SETUP}-\texttt{HOLD}$, the glitch-free period in between two glitches can be shortened to longer than $3 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$. So, 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 is allowed for.

Using both the third and the fourth clock, a and b, allowed to model two independent glitches, represented by the automata locations `GlitchA` and `GlitchB`. Clock a was associated to `GlitchA` and clock b to `GlitchB`. The clocks were used to measure the duration of a glitch modeled by their associated location, and the period after the glitch that was free of glitches modeled by this location. For example, if a glitch was modeled by `GlitchA`, another glitch modeled by `GlitchB` could occur during the "glitch modeled by `GlitchA`"-free period after the former glitch, but no further glitch could occur after the latter one until the period measured by a was over, as the period free of "glitches modeled by `GlitchB`" after the latter glitch leaves no location that could be used to model this additional glitch. This model (the glitches affect at most two arbitrary samples in the sequence) was analyzed with Uppaal version 4.1.4-64bit in 335 seconds using 1.3GiB of memory.

Again, the results confirmed our findings from the sample glitch model: if you have two arbitrary glitches, and each glitch is shorter than $1 * \texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$, the "glitch modeled by the same location"-free period in between two glitches modeled by the same location should be longer than $87 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$ to allow the glitches to be tolerated. So, two arbitrary glitch-affected samples in each sequence of 88 consecutive samples can be tolerated.

When comparing UPPAAL's time and memory consumption of the sample glitch model with the real-time glitch model, it can be seen that the introduction of large discrete counters, as needed to count the glitch free samples in the 2 in 88 case, can be more expensive than introducing additional clocks, depending on the tool used.

A. Analysis of the Glitch Patterns

In hindsight, the resilience of the protocol against these specific glitch patterns can be partially explained using insight into the protocol: The FlexRay physical layer protocol uses the strobing process to pick the 5th voted value out of the 8 voted values corresponding to the 8 samples a bit cell is composed of in the ideal case. However, this strobing mechanism uses the counter variable `strobecounter` which is synchronized to the sample stream at sync edges, which occur during the Byte Start Sequence (BSS). The sync edges are thus separated by 10 bit cells, which consist of 8 samples each in the ideal case.

1 in 4. One glitch-affected sample in a sequence of 4 consecutive samples can lengthen or shorten a received bit cell by at most one sample. If there is just one glitch in a voting window, this is easily seen.

However, in a voting window of size five, there can be 2 glitch-affected samples, but only if the last and the first sample of the voting window are glitch-affected. As such a voting window contains 3 not glitch-affected samples, this means that the majority of the voting window can only be changed if the three other samples in the voting window do not agree on a value. In turn, this means that the voting window without the glitches would be filled with samples of value $j \in \{0, 1\}$ up to a position $1 < i < 5$, and would be filled with samples of value $(1 - j)$ in the positions after i. As the position 1 and 5 are glitch affected, they could both flip their value, which would not affect the majority, or just one of them flips its value, which can change one sample of the five from j to $(j - 1)$ or vice versa, resulting in a changed voted value one receiver clock cycle to late or to early.

Even the sync edges can be delayed or arrive early by up to 1 sample. In this case, the 4th or 6th voted value may be strobed instead of the 5th, while only the 1st or 8th can be affected by a glitch. Clockdrift can lead to one FlexRay controller overtaking the other once every $\texttt{CYCLE\_MAX}/(\texttt{CYCLE\_MAX} - \texttt{CYCLE\_MIN})$ samples, so the 3rd or 7th voted value may also be strobed, but these cannot be affected by glitches.

Thus, 1 glitch-affected sample in every sequence of 4 consecutive samples does not cause a transmission error.

2 in 88. Two glitch-affected samples can lengthen or shorten a received bit cell by up to two samples. The sync edges can be delayed or arrive early by up to 2 samples. Thus, `strobecounter` can be up to 2 samples late or early due to glitches. This can cause the 3rd or the 7th voted value to be strobed. Accounting for drift, the 2nd or 8th voted value could be strobed. However, the 1st, 2nd, 7th or 8th voted value of a bit cell could be affected by a glitch. Thus, another glitch-affected sample before resynchronization of the counter `strobecounter` could lead to a glitch-affected voted value being strobed, which in turn could violate the stream format or flip a message bit, so we have to exclude this by increasing the glitch free period.

Resynchronization of `strobecounter` will happen at the next sync edge, every 80 samples in the ideal case. To justify the number of 88 samples in the case of two glitch-affected samples occurring next to each other, lets consider an example: Assume that 2 samples are flipped 2 samples before a sync edge, i.e., the i^{th} sample and the $(i + 1)^{\text{th}}$ sample are flipped, and the sync edge occurs after the $(i + 3)^{\text{th}}$ sample. The counter `strobecounter` would be 2 samples early, strobing the 3rd voted value of each bit cell. Drift moves this to strobing the 2th voted value, assuming that the receiver's oscillator is faster than the sender's oscillator. The next sync edge will occur after the $(i + 3 + 80 + 1)^{\text{th}}$ sample, the drift having added one extra sample. If the $(i+3+80+1+3)^{\text{th}}$ and $(i+3+80+1+4)^{\text{th}}$ samples are flipped, the strobecounter would again be 2 samples early. This scenario assumes 4 glitch-affected samples in a sequence of $1 + 3 + 80 + 1 + 4 = 89$ samples. This can be excluded if each sequence of $3 + 80 + 1 + 4 = 88$ samples may contain at most 2 glitch-affected samples.

However, this example is not sufficient to explain the result obtained from the real-time-glitch model that a glitch of less than $2 * \texttt{CYCLE\_MIN}$ and a glitch-free period of more than $86 * \texttt{CYCLE\_MAX}$ in between two glitches are allowed for, as it assumes a receiver's oscillator being faster than the sender's one, thus adding one extra sample. The sample added cannot be glitch-affected, as three glitch-affected samples

Table 1. Standard parameter values based on conservative approximations of the parameters taken from the FlexRay standard[10] and the Nangate Open Cell Library.[9]

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 (adj.) out of x" stands for "at most y (adjacent) glitch-affected samples in x consecutive samples".

Glitch pattern	Parameter	Value	Corresponds to
1 out of 4	$\texttt{ERRDIST}_s$	3	3 samples
2 adj. out of 88	$\texttt{ERRDIST}_s$	86	86 samples
2 out of 88	$\texttt{ERRDIST}_s$	87	87 samples
Short real-time glitch	`ERRDUR`	$\texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$	$10.57125\,ns$
	$\texttt{ERRDIST}_t$	$3 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$	$39.46625\,ns$
Long real-time glitch	`ERRDUR`	$2 * \texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$	$23.0525\,ns$
	$\texttt{ERRDIST}_t$	$86 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$	$1078.5225\,ns$
2 independent real-time glitches	`ERRDUR`	$\texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$	$10.57125\,ns$
	$\texttt{ERRDIST}_t$	$87 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$	$1091.04125\,ns$

can cause a changed voted value to be strobed. Thus it would have to be added in the glitch-free period of more than $86 * \texttt{CYCLE\_MAX}$, yielding 87 not glitch-affected samples in that period. However, only 86 not glitch-affected samples are required for the example, so for the 87th sample, the requirement that it is not affected by a glitch could be dropped. This contradicts the result that the glitch free period has to be longer than $86 * \texttt{CYCLE\_MAX}$, as even a glitch free period of exactly $86 * \texttt{CYCLE\_MAX}$ can lead to a transmission error. The discrepancy between the requirements on the glitch patterns in our scenario constructed using hindsight with the results in mind and the stricter requirements on the glitch pattern induced by model-checking the real-time-glitch model reiterate the need for automatic analysis of such intricate scenarios, as manual analysis fails to achieve the same precision, even if the desired results of the analysis are known beforehand.

B. Analysis of Parameters

We use Uppaal version 4.1.4-64bit to check the non-reachability of location `DECerr`, i.e., we check that no transmission error occurs. Additional properties were checked[8] in the basic model, but, being the essential property, only absence of transmission errors was checked in the models with varied parameters.

From,[8] we know the relationship between acceptable error patterns and the size of the voting window: For voting windows of an uneven size from 3 to 9, one glitch-affected sample in a consecutive sequence of $\lceil \mathit{windowsize}/2 \rceil + 1$ samples is tolerable.

In a first analysis, we use conservative approximations based on,[9, 10] which are listed in Table 1. We globally assume a CPU frequency of $80\,MHz$

The impact of changing *either* the hardware parameters `PMIN` and `PMAX`, or `DEVIATION` on the amount of tolerable glitches is shown in Table 3.

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

Note the subtle difference in the maximal tolerable delay variance $\texttt{PMAX} - \texttt{PMIN}$ and clock drift `DEVIATION` between short real-time glitches and long ones. This shows that there is an intricate relationship between the glitch patterns and the tolerable parameter changes. Nevertheless, the analysis provides stringent hardware requirements which, if met, will guarantee robustness against the respective glitch patterns.

The results of Müller and Paul,[6] demonstrate that a controller with a `DEVIATION` of 76 is safe for a FlexRay like bus if a reliable physical layer is assumed. Together with our result that the FlexRay physical layer protocol running on hardware with a `DEVIATION` of 80 can tolerate an unreliable physical layer with glitch patterns described in Table 2, demonstrates the resilience of the FlexRay physical layer

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". For the DEVIATION measurements, the values ERRDUR and ERRDIST$_t$ correspond to are the changed results yielded by the formulas for their value.

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 adj. out of 88
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6056$[c]	2 out of 88
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6086$	short real-time glitch
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6056$	long real-time glitch
$\mathtt{PMAX} - \mathtt{PMIN} \leq 6056$	2 ind. real-time glitches
$\mathtt{DEVIATION} \leq 92$	1 out of 4
$\mathtt{DEVIATION} \leq 80$[c]	2 out of 88
$\mathtt{DEVIATION} \leq 90$	2 adj. out of 88
$\mathtt{DEVIATION} \leq 92$	at most 2
$\mathtt{DEVIATION} \leq 218$	at most 1
$\mathtt{DEVIATION} \leq 348$	none
$\mathtt{DEVIATION} \leq 92$	short real-time glitch
$\mathtt{DEVIATION} \leq 90$	long real-time glitch
$\mathtt{DEVIATION} \leq 80$	2 ind. real-time glitches

[c]largest value which cannot be shown to be unsafe when UPPAAL is limited to 127GiB of memory, the maximum available to the authors

protocol as well as it's clock synchronization and time-division-multiple-access scheme against less precise oscillators.

Furthermore, our results demonstrate a high resilience of the FlexRay physical layer protocol against changes in the variance of the propagation delay. From[9] we initially assumed a variance in the propagation delay of $1435\,ps$, but we could increase this to $7570\,ps$ without changing the tolerable error patterns. If the length of the harness is increased, this will be vital, as the delay variance of a longer harness will be greater than that of a short one.[1]

If the model is configured with the properties of a specific hardware environment, it is easy to reevaluate which error patterns are tolerable, and thus provide hard guarantees on error resilience, which is very desirable in an application area like aerospace.

VII. Conclusion

We provided a model of FlexRay's physical layer protocol and its hardware environment. Several lessons learned during the modeling were presented and can be applied when modeling physical layer protocols.

Using UPPAAL to analyze our model, we confirmed the high error-resilience of FlexRay's physical layer protocol. Providing different glitch patterns, we demonstrated how they can either be modeled as sample glitches from the protocol-centric point of view, or as real-time glitches from the glitch-centric point of view. Our observations confirm that the way of modeling a glitch pattern does not influence the protocol's error resilience against this pattern. We analyzed the effect of changing hardware parameters on the error resilience of the protocol against these patterns. During our analysis, we noted that increasing the delay variance decreases the maximal tolerable clock drift, and vice versa. For 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]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.

[7]Müller, C., *Complete Formal Hardware Verification of Interfaces for a FlexRay-like Bus*, PhD-thesis, Universität des Saarlandes, 2011.

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

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

[11]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.

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

[13]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.

[14]Vaandrager, F. and Groot, A. d., "Analysis of a Biphase Mark Protocol with Uppaal and PVS," *Formal Aspects of Computing Journal*, Vol. 18, No. 4, December 2006, pp. 433–458.

[15]Behrmann, G., David, A., and Larsen, K. G., "A Tutorial on UPPAAL," *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, edited by M. Bernardo and F. Corradini, No. 3185 in LNCS, Springer–Verlag, September 2004, pp. 200–236.

[16]Brown, G. M. and Pike, L., "Easy Parameterized Verification of Biphase Mark and 8N1 Protocols," *TACAS*, Vol. 3920 of *LNCS*, Springer, 2006, pp. 58–72.

[17]Alur, R. and Dill, D. L., "A Theory of Timed Automata," *Theo. Comp. Sci.*, Vol. 126, No. 2, 1994.

[18]Böhm, P., "*Implementation of the High-Level Components of a Bus Controller for a Time Triggered Serial Bus*," Bachelorthesis, Universität des Saarlandes, 2006.

[19]Männer, R., "Metastable states in asynchronous digital systems: Avoidable or unavoidable?" *Microelectronics Reliability*, Vol. 28, No. 2, 1998, pp. 295–307.

[20]Keller, J. and Paul, W. J., *Hardware design: Formaler Entwurf digitaler Schaltungen*, Vol. 15 of *Teubner-Texte zur Informatik*, 1995.