Model Checking the FlexRay Physical Layer Protocol

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

Reactive Systems Group
Saarland University
66123 Saarbrücken, Germany
{gerke | ehlers | finkbeiner | peter}@cs.uni-saarland.de

Abstract. The FlexRay standard, developed by a cooperation of leading companies in the automotive industry, is a robust communication protocol for distributed components in modern vehicles. In this paper, we present the first timed automata model of its physical layer protocol, and we use automatic verification to prove fault tolerance under several error models and hardware assumptions.

The key challenge in the analysis is that the correctness of the protocol relies on the interplay of the bit-clock alignment mechanism with the precise timing behavior of the underlying asynchronous hardware. We give a general hardware model that is parameterized in low-level timing details such as hold times and propagation delays. Instantiating this model for a realistic design from the Nangate Open Cell Library, and verifying the resulting model using the real-time model checker UPPAAL, we show that the communication system meets, and in fact exceeds, the fault-tolerance guarantees claimed in the FlexRay specification.

1   Introduction

The safety-critical functionality of modern cars is increasingly implemented in distributed embedded components that connect through a robust communication system. Since delays or communication errors in such X-by-wire applications can cause serious harm, fault tolerance is a key consideration in the design of the communication protocols.

In this paper, we study the physical layer of the FlexRay protocol [7]. Developed by the FlexRay Consortium, a cooperation of leading companies including BMW, Bosch, Daimler, Freescale, General Motors, NXP Semiconductors, and Volkswagen, FlexRay was first employed in 2006 in the pneumatic damping system of BMW's X5, and fully utilized in 2008 in the BMW 7 Series. The FlexRay specification was completed in 2009 and is widely expected to become the future standard for the automotive industry.

The role of the physical layer is to compensate for low-level communication errors such as *glitches*, i.e., incorrect transmissions due to electromagnetic interference and similar effects, and *jitter*, resulting from clock drift between asynchronous components. For this purpose, the protocol includes a complicated

voting and *bit-clock alignment* mechanism, which analyzes a stream of samples, identifies the boundaries of the individual bit transmissions, and computes the correct value of the bits.

How robust is the resulting protocol? The FlexRay standard states, somewhat vaguely, that "the decoding function attempts to enable tolerance of the physical layer against presence of one glitch in a bit cell when the length of the glitch is less than or equal to one channel sample clock period," adding in a footnote that "there are specific cases where a single glitch cannot be tolerated and others where two glitches can be tolerated" [7, Sect. 3.2.7]. Clearly, a more precise characterization of the fault tolerance is desirable. The challenge is, however, that the correctness of the protocol relies on the interplay of the bit-clock alignment mechanism with the timing behavior of the asynchronous hardware. A careful analysis of the fault tolerance must therefore include a detailed timing model of the underlying hardware.

Previous efforts [4,13,12,9,1] to analyze FlexRay have been based on manual or semi-automatic verification methods, which make it very difficult to determine the robustness of the protocol under different error models and hardware configurations. We present a new formalization of the FlexRay physical layer protocol, parameterized in several low-level timing details such as hold times and propagation delays, that is based on *timed automata*. Because timed automata can be analyzed fully automatically using model checkers such as UPPAAL [3], we can quickly analyze the model for different settings and track the dependence of the protocol on hardware and design parameters.

Our analysis provides a detailed picture of the robustness of the FlexRay physical layer protocol. We show that, for typical hardware parameters, such as those of a realistic design from the Nangate Open Cell Library [11], FlexRay tolerates one glitch every four samples. In fact, this tolerance is robust under variations of the hardware. For example, the protocol tolerates a clock drift of up to 0.46%, which significantly exceeds the limit of 0.15% described in the FlexRay standard. While fault tolerance thus 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 the following sections, we give a detailed presentation of the model and the results of our analysis.

2 Overview

We present a model of the physical layer protocol of the FlexRay coding/decoding unit (CODEC). As illustrated in Fig. 1, our model is structured into a model of the protocol and a model of the underlying hardware. The *protocol model*, which is given in Sect. 3, consists of a sender and a receiver.

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
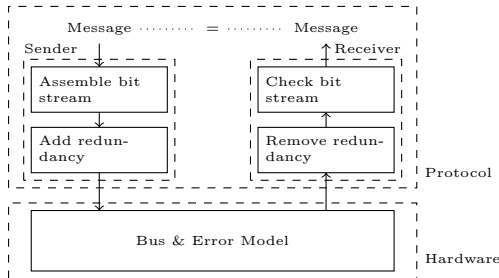
Fig. 1. The structure of the model.

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 eigth clock cycles.

The *receiver* in turn reads one value in every clock cycle from the bus (the so-called *samples*), removes the redudancy 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 Sect. 4, describes the underlying hardware, including the communication bus and the error model describing 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 [7, Chap. 5], FlexRay uses a time division multiple access (TDMA) scheme, which excludes collisions [1]. 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.

2.1 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.

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. We model this by nondeterministically flipping samples in the receiver process. Such a flip is called a *glitch* [7, Sect. 3.2.2]. If too many glitches occur, the message might be compromised. However,

the FlexRay physical layer protocol compensates for infrequent glitches. Our error model is parameterized in the *error distance*, which gives a lower bound on the number of correct samples between any pair of samples affected by glitches.

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*.

2.2 Timed Automata

We describe our model as a *network of timed automata* [2]. We assume familiarity with timed automata and refer the reader to a Uppaal tutorial [3] for more background. 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 at that location. Transitions can be labeled with *broadcast synchronization channels* over which a sender (identified by "!") can force receivers (identified by "?") to take a transition. Also, each transition can have an *update expression* to set clock or integer variables, and a *guard* determining its enabledness. Furthermore, a location can be marked as *committed* to force the system to immediately leave the location before time can pass. To improve the readability of complex models, we cut large automata into smaller ones.

2.3 Related work

There are several previous formalizations of the FlexRay physical layer protocol. Beyer et al. [4] gave the first manual deductive correctness proof. In [12,13], Schmaltz 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 [9,1].

Vaandrager et al. [14] use Uppaal 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 [6] 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, this paper presents a *fully automatic* correctness proof of the FlexRay physical layer protocol only using the real-time model checker Uppaal [3]. Furthermore, we consider a more realistic *unreliable* physical environment to study the fault tolerance of the protocol.

In addition to protocol verification, there are several related works in the more general setting of hardware verification. Bozga et al. [5] verify asynchronous

circuits with the real-time model checker KRONOS, where the low-level timing behavior of the individual gates is modeled by timed automata. A hierarchical approach to the verification of asynchronous circuits is described in [15]. By translating the system model together with a scheduler restricting the temporal evolution of the system into a communicating sequential processes (CSP) model, the possible timing behavior of the system is over-approximated to allow the efficient verification using a CSP model checker. The focus of this line of research is the analysis of asynchronous circuits on a chip, not the communication protocols considered in this paper.

3 The Protocol Model

We model a scenario in which the sender transmits a formatted bit stream, and the receiver checks if the format of the stream complies with the standard described in [7, Sect. 3.2.1.1] and if all message bits are received correctly. To avoid unnecessary counter variables that keep track of the current position within the message, we abstract from the concrete message length: after each transmitted byte, we let the sender nondeterministically determine whether a further message byte should follow, thus allowing an arbitrary message length.

3.1 The Sender

The Bit Stream Format. A message is transmitted as a structured stream [7, Sect. 3.2.1.1] of bit cells as shown in Fig. 2. As stated in Sect. 2.1, 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. 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.

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.
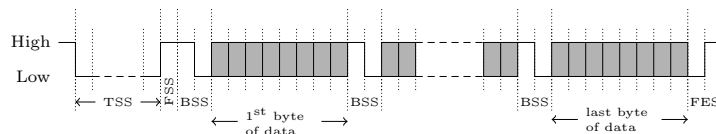

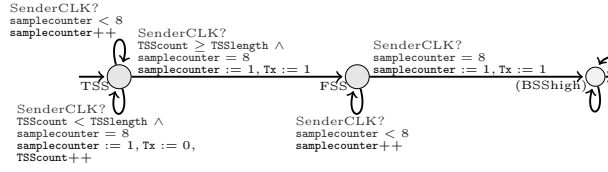**Fig. 2.** Format of a message bit stream.

5

Fig. 3. 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. 3. The message is generated nondeterministically as shown in Fig. 4. 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`[1]. 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. 5).
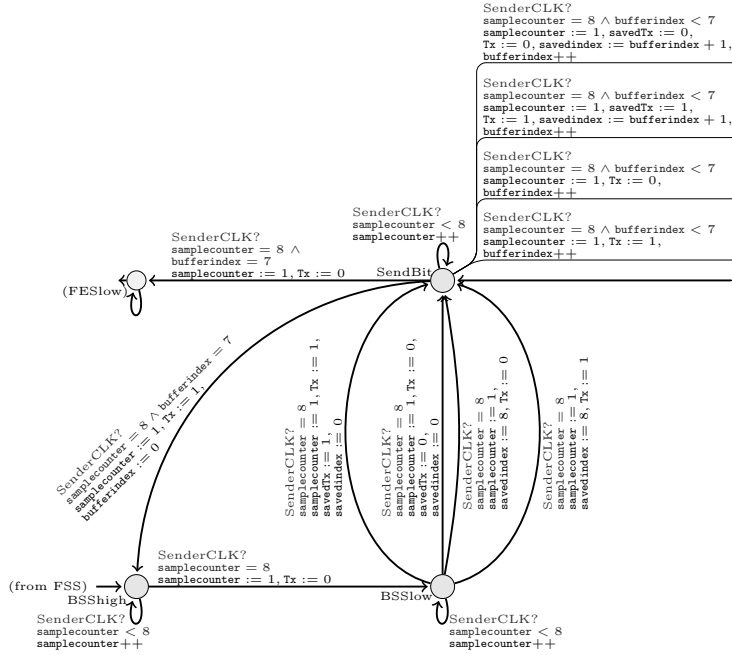

Fig. 4. Model of the transmission of the message bytes.

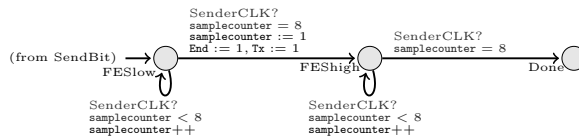[1] The inital value `savedindex` $= 8$ means "no bit to test".

Fig. 5. Model of the end of the transmission.

3.2 The Receiver

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*.[2] 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.
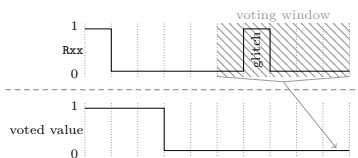

Fig. 6. Correction of a glitch through majority voting.

As depicted in Fig. 6, infrequently occurring glitches are mostly filtered out directly. However, if a glitch occurs close to a change in the sample sequence, it leads to a premature or delayed change of the voted value. More precisely, if the glitch inverts one of the samples of the new value, it takes one more cycle until the new value becomes the majority in the voting window. On the other hand, if the glitch inverts one sample of the old value, the value will change one cycle too early. Such untimely changes of the voting value may also be the result of jitter, as described in Sect. 4.2. The errors can also occur in combination, as shown in Fig. 7.

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. In every 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`.

---

[2] According to the FlexRay standard [7, 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 [4,13,12,1,9].
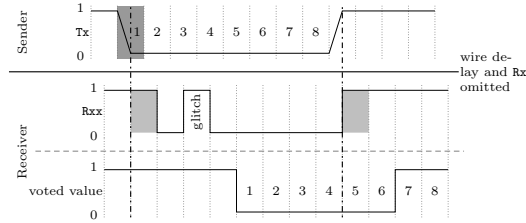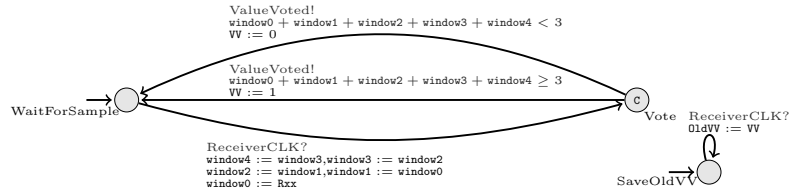
Fig. 7. Combination of jitter and glitch.

Fig. 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.

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.

If a combination of clock drift and a glitch interferes with the bit clock alignment mechanism by delaying the recognition of the high to low transition, `strobecounter` will be off by more than 1, thus parts of the next bit cell are also taken into account when computing the strobed value. This situation is shown in Fig. 7; recall the delay of two cycles introduced by the voting process. The bit clock alignment can analogously also happen too early.

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`.

8

When `strobecounter` has a value of 5 and channel `ValueVoted` signals that the voted value for this cycle of the receiver's clock is reached, `VV` is chosen as the value for `bstr`. Channel `Strobed` allows other automata to synchronize on this event in order to use the new `bstr` value.

Fig. 9. Model of the strobing process.

Receiving the Bit Stream. When channel `Strobed` signals that a new value has been strobed, the receiver checks if it is consistent with the expected format of the bit stream, 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.

Fig. 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 received value (stored in `bstr`) is compared to `savedTx`. The variable `End` is checked to prohibit entering the location `Done` too early, as shown in Fig. 12.
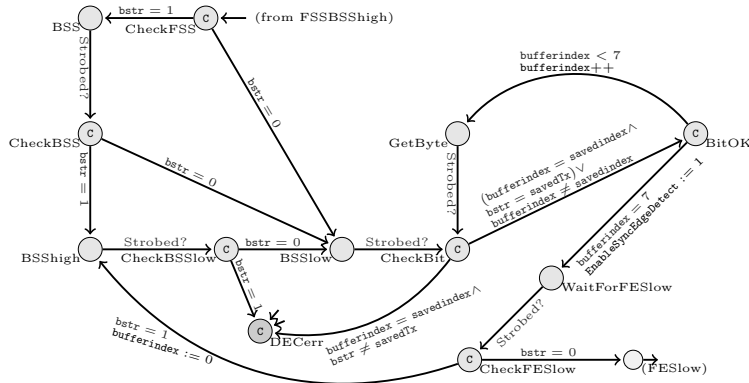

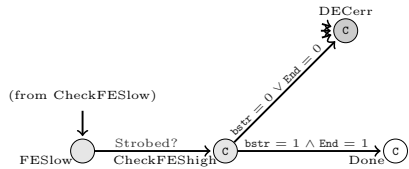**Fig. 11.** Model of the reception of the message bytes.

Fig. 12. Model of the end of the reception.

4 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 13 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 [4,13,12,9,1], as proposed by [10], we forward the output of register `Rx` through a consecutive register `Rxx` to suppress metastability problems.
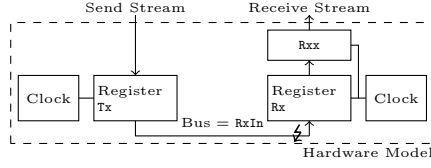
Fig. 13. Overview of the hardware sub-architecture.

4.1 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. According to the specification, distributed oscillators may deviate from the standard rate up to a certain bound [7, 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. 14. 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:

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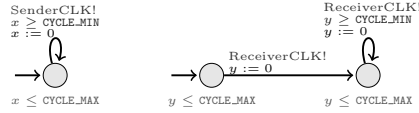

Fig. 14. Oscillators for sender and receiver.

4.2 Registers

Following the setting of [4,13,12,9,1], we assume a *register semantics* to model the timing behavior of the bus which connects the sender and the receiver. Before we come to the actual transmission of bit values via the bus, we first give a general description of the low-level timing behavior of registers.

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 ≤ 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. 15 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.
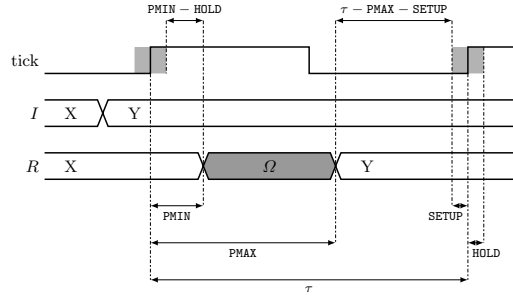

Fig. 15. Value change scenario of a register R.

We assume that the unknown value is stable before $\tau - \texttt{SETUP}$, 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 [8, 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

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

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

Model of the Bus. Figure 16 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. 13. In our model, we represent register `Tx`'s content by a variable `Tx`, and register `Rx`'s input (which also represents the bus' content) by a variable `RxIn`. As the bus value is high whenever it is idle [7, Sect. 3.2.4], `RxIn` is initialized with 1.

At every tick of the sender's clock, the variable `Tx` is checked: 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, `RxIn` 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

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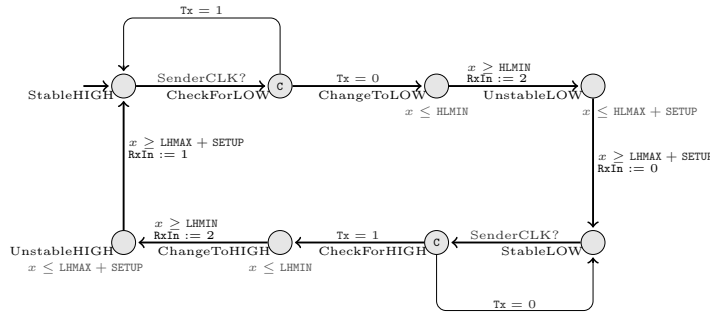

Fig. 16. Model of the bus.

Model of the Receiving Register. Figure 17 shows the automata modeling the sampling process on the receiver's side. The receiver samples a value from the bus using the register `Rx`. After exactly `HOLD` time units following a tick-event, we update `Rx` either (1) nondeterministically with 1 or 0 if `Rx`'s input `RxIn` changes or is undefined, or (2) with `RxIn` otherwise.

Furthermore, for modeling glitches, we introduce a variable **lasterror** that counts the number of samples without a glitch. Whenever **lasterror** $\geq$ `ERRDIST`, the sampling process nondeterministically decides whether the current sample is affected by a glitch.

5 Model Checking the FlexRay Physical Layer Protocol

In our analysis, we fix values for the model parameters and check several correctness properties (shown in Table 1) using the real-time model checker UPPAAL [3].
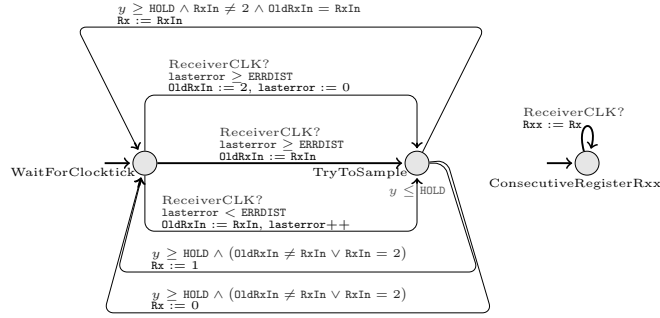
Fig. 17. Model of the sampling process.

In a first analysis, we use conservative approximations based on [7,11], which are listed in Table 2(a). We globally assume a CPU frequency of 80 MHz[3].

Table 1. Satisfied correctness properties and corresponding running times of UPPAAL on a computer with an AMD Opteron 2.6 GHz and 4 GB RAM.

Property	MC Time
`A<> Receiver_Control.TSS` It is always the case that the reception of the bit stream eventually starts.	0.65 sec
`A<> Receiver_Control.CheckFESlow` It is always the case that the first byte of a message is eventually correctly received.	7624.90 sec
`A[] !Receiver_Control.DECerr` Invariantly, the received bit stream is in the correct format and the received message is correct.	73.08 sec
`A[] (!Deadlock || Receiver_Control.Done)` Invariantly, there is no deadlock before the message is completely received.	136.47 sec

We initially assume an error distance of four which corresponds to one glitch in a voting window. This intuitive choice is overly pessimistic: in fact, the experiments show that for the standard parameters, we can tolerate an error distance of three without violating any correctness property.

The impact of changing the hardware parameters `PMIN`, `PMAX`, or `DEVIATION` on the amount of tolerable glitches (such that the properties from Table 1 are still preserved) is shown in Table 2(b). Interestingly, this analysis demonstrates the robustness of the FlexRay physical layer protocol even for more pessimistic hardware assumptions: beyond our conservative choice of the parameters, there is still a comfortable safety margin for reasonable error models.

---

[3] Note that 80 MHz corresponds to an *ideal* clock cycle. Recall that every actual clock cycle of a CPU may deviate up to a certain rate, defined by `DEVIATION`.

Table 2. Standard values based on conservative approximations of the parameters taken from the FlexRay standard [7] and the Nangate Open Cell Library [11], as well as the impact of changed parameters on the tolerable glitches. Here, "1 out of x" stands for "at most 1 glitch in x consecutive samples" and thus an error distance of $x - 1$, and "at most y" means "at most y glitches in the overall stream at arbitrary positions".

(a) Standard parameter values.

Parameter	Value	Corresponds to
CYCLE	10000	$\frac{1}{80\,MHz} = 12.5\,ns$
DEVIATION	30	$\pm 0.15\,\%$
SETUP	368	$460\,ps$
HOLD	1160	$1450\,ps$
PMIN	12	$15\,ps$
PMAX	1160	$1450\,ps$
ERRDIST	4	1 out of 5

(b) Changed parameter values.

Changed parameter	Tolerable glitches
PMAX − PMIN ≤ 6086	1 out of 4
PMAX − PMIN ≤ 6086	at most 2
PMAX − PMIN ≤ 9616	at most 1
DEVIATION ≤ 92	1 out of 4
DEVIATION ≤ 92	at most 2
DEVIATION ≤ 218	at most 1
DEVIATION ≤ 348	none
Voting window size = 3	1 out of 3
Voting window size = 5	1 out of 4
Voting window size = 7	1 out of 5
Voting window size = 9	1 out of 6

With slightly more elaborate adjustments to the automaton from Fig. 17, we also investigate an error model with two arbitrary glitches within every sequence of samples of a certain length. For instance, assuming the standard parameters from Table 2(a), it turns out that two glitches in a sequence of up to 82 samples lead to a violation of the correctness properties. The impact of changing the size of the voting window is shown in the last four rows of Table 2(b). Here, the error distance increases linearly in the size of the window.

6 Conclusion

In this paper, we have demonstrated the use of automatic verification to analyze the fault tolerance of a complex real-time protocol under variations of the design parameters, the error model, and the hardware parameters. Beyond proving that the physical layer protocol *meets* the fault tolerance requirements claimed in the FlexRay specification, our analysis gives a detailed picture of the *impact* the different parameters have on the robustness of the protocol.

An *a posteriori* analysis, as carried out in this paper, is helpful to understand the importance of individual design choices and hardware requirements in an established protocol, and to identify requirements that are too conservative and can therefore be relaxed. An interesting direction for future research might be to carry out the analysis *a priori*, exploring the design space of an as yet unfinished protocol: model checking variations of the protocol on a parameterized hardware model, like the one presented in this paper, can help the designer make safe and robust choices.

References

1. Alkassar, E., Böhm, P., Knapp, S.: Formal correctness of an automotive bus controller implementation at gate-level. In Kleinjohann, B., Kleinjohann, L., Wolf, W., eds.: 6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008). Volume 271 of International Federation for Information Processing., Springer (2008) 57–67
2. Alur, R., Dill, D.L.: A theory of timed automata. Theo. Comp. Sci. **126**(2) (1994)
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In Bernardo, M., Corradini, F., eds.: 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. Number 3185 in LNCS, Springer–Verlag (September 2004) 200–236
4. Beyer, S., Böhm, P., Gerke, M., Hillebrand, M., Rieden, T.I.d., Knapp, S., Leinenbach, D., Paul, W.J.: Towards the formal verification of lower system layers in automotive systems. In: ICCD '05: Proceedings of the 2005 International Conference on Computer Design, IEEE Computer Society (2005) 317–326
5. Bozga, M., Jianmin, H., Maler, O., Yovine, S.: Verification of asynchronous circuits using timed automata. Electr. Notes Theor. Comput. Sci. **65**(6) (2002)
6. Brown, G.M., Pike, L.: Easy parameterized verification of biphase mark and 8N1 protocols. In: TACAS. Volume 3920 of LNCS., Springer (2006) 58–72
7. FlexRay Consortium: FlexRay Communications System Protocol Specification Version 2.1 Revision A. (2005)
8. Keller, J., Paul, W.J.: Hardware design: Formaler Entwurf digitaler Schaltungen. Volume 15 of Teubner-Texte zur Informatik. (1995)
9. Knapp, S., Paul, W.: Realistic worst case execution time analysis in the context of pervasive system verification. In Reps, T., Sagiv, M., Bauer, J., eds.: Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday. Volume 4444 of Lecture Notes in Computer Science. Springer (2007) 53–81
10. Männer, R.: Metastable states in asynchronous digital systems: Avoidable or unavoidable? Microelectronics Reliability **28**(2) (1998) 295–307
11. Nangate Inc.: Nangate 45nm Open Cell Library Databook. (2009)
12. Schmaltz, J.: A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In Baumgartner, J., Sheeran, M., eds.: 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07), IEEE Press Society (November 11–14 2007) 223–230
13. Schmaltz, J.: A formal model of lower system layers. In: Formal Methods in Computer Aided Design (FMCAD'06), IEEE Computer Society (2006) 191–192
14. Vaandrager, F., Groot, A.d.: Analysis of a biphase mark protocol with Uppaal and PVS. Formal Aspects of Computing Journal **18**(4) (December 2006) 433–458
15. Wang, X., Kwiatkowska, M.Z., Theodoropoulos, G.K., Zhang, Q.: Towards a unifying CSP approach to hierarchical verification of asynchronous hardware. Electr. Notes Theo. Comp. Sci. **128**(6) (2005) 231–246