Modeling and Verifying the FlexRay Physical Layer Protocol with Reachability Checking of Timed Automata

UNIVERSITÄT
DES
SAARLANDES

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für
Mathematik und Informatik
der Universität des Saarlandes

Michael Gerke

Saarbrücken, 2020

Dekan: Univ.-Prof. Dr. Thomas Schuster

Datum des Kolloqiums: 21.12.2020
Vorsitzende des Prüfungsausschusses: Prof. Martina Maggio, Ph.D.
Berichterstatter im Prüfungsausschuss: Prof. Bernd Finkbeiner, Ph.D.,
Prof. Dr. Ernst-Rüdiger Olderog
Akademisches Mitglied des Prüfungsausschusses: Dr.-Ing. Tim Dahmen

"There is a concept which corrupts and upsets all others. I refer not to Evil, whose limited realm is that of ethics; I refer to the infinite."
(Jorge Luis Borges,
"Avatars of the Tortoise",
in Jorge Luis Borges, Donald A. Yates: *Labyrinths: Selected Stories & Other Writings*,
New Directions Publishing, 1964, page 202)

Danksagung

Ich möchte Prof. Finkbeiner, Ph.D., für seine Unterstützung, seine Geduld, und seine Ratschläge danken, die mir erlaubten, die vorliegende Dissertation zu verfassen. Besonderen Dank schulde ich auch Dr. Hans-Jörg Peter und Prof. Dr. Rüdiger Ehlers, von denen ich während meiner Zusammenarbeit mit ihnen viel gelernt habe. Mein Dank gilt auch meiner Familie und meinen Freunden und Kollegen, die mir während meines Promotionsstudiums den Rücken stärkten und mir immer wieder Kraft zum Weitermachen gaben. Schließlich möchte ich mich auch bei Schloss Dagstuhl, dem Leibniz-Zentrum für Informatik, dafür bedanken, dass ich die vorliegende Arbeit neben meiner dortigen neuen Tätigkeit noch vervollständigen konnte.

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).

Abstract. In this thesis, I report on the verification of the resilience of the FlexRay automotive bus protocol's physical layer protocol against glitches during message transmission and drifting clocks. This entailed modeling a significant part of this industrially used communictation protocol and the underlying hardware as well as the possible error scenarios in fine detail. Verifying such a complex model with model-checking led me to the development of data-structures and algorithms able to handle the associated complexity using only reasonable resources. This thesis presents such data-structures and algorithms for reachablity checking of timed automata. It also present modeling principles enabling the construction of timed automata models that can be efficiently checked, as well as the models arrived at. Finally, it reports on the verified resilience of FlexRay's physical layer protocol against specific patterns of glitches under varying assumptions about the underlying hardware, like clock drift.

Zusammenfassung. In dieser Dissertation berichte ich über den Nachweis der Resilienz des Bitübertragungsprotokolls für die physikalische Schicht des FlexRay-Fahrzeugbusprotokolls gegenüber Übertragungsfehlern und Uhrenverschiebung. Dafür wurde es notwendig, einen signifikanten Teil dieses industriell genutzten Kommunikationsprotokolls mit seiner Hardwareumgebung und die möglichen Fehlerszenarien detailliert zu modellieren. Ein so komplexes Modell mittels Modellprüfung zu überprüfen führte mich zur Entwicklung von Datenstrukturen und Algorithmen, die die damit verbundene Komplexität mit vernünftigen Ressourcenanforderungen bewältigen können. Diese Dissertation stellt solche Datenstrukturen und Algorithmen zur Erreichbarkeitsprüfung gezeiteter Automaten vor. Sie stellt auch Modellierungsprinzipien vor, die es ermöglichen, Modelle in Form gezeiteter Automaten zu konstruieren, die effizient überprüft werden können, sowie die erstellten Modelle. Schließlich berichtet sie über die überprüfte Resilienz des FlexRay-Bitübertragungsprotokolls gegenüber spezifischen Übertragungsfehlermustern unter verschiedenen Annahmen über die Hardwareumgebung, wie etwa die Uhrenverschiebung.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Extreme Computerization

Since the 1960s, our world has witnessed the third industrial revolution, also called the digital revolution [Sch16]. Since the development of semiconductors, computers went from mainframes to personal computers, to the internet and ubiquitous computing. At the end of the last millennium, traditional general purpose computers had already been reduced to only a minuscule fraction of all computers in the world, with almost all microprocessors being ubiquitously used in *embedded systems* [Tur99]. Today, computers outnumber humans by more than ten to one [Fur17].

In the fourth industrial revolution, ubiquitous computers communicating with each others are predicted to play an important role, i.a., transforming the methods of production by giving rise to *smart factories*, a development which is also referred to as *Industry 4.0* [Sch16]. Every aspect of our modern civilization, be it transportation, power and water supply, administration, agriculture or production is ever more dependent on communicating computers.

Modern planes are almost exclusively controlled via computers, which relay the pilot's commands. This control paradigm is called *fly-by-wire* and its roots go back more than half a century [Tom00]. More generally, *x-by-wire* stands for the idea to control a critical function of a vehicle via computers relaying the operator's commands to another computer that, in turn, activates some actuator of the vehicle, like, e.g., the brakes. For more than two decades, cars have been containing dozens of microprocessors [Tur99]. To facilitate communication between these microprocessors, they have been connected by communication buses like *controller area network* (CAN) [KDL86] or, lately, FlexRay [Fle05, Fle10b], which will be discussed in more detail in this thesis (For an introduction to FlexRay see Chapter 2).

1.2 Vulnerability of computerized systems

The massive spread of and dependence on computers makes our civilization vulnerable to faults in the design of these computers. With the huge number of microprocessors

that are produced, erroneous designs can affect a sizable portion of our computing infrastructure. For example, it is estimated [Gil18] that at least three billion computers have the *Spectre* and *Meltdown* vulnerabilities, which expose the computers to the risk of certain hacking attacks. Failures of computer systems or attacks against them are economically very costly: In 2012, Gene Kim and Mike Orzen [Kri12] estimated the global annual cost of IT failure at 3 trillion USD. And in 2019, Accenture [AB19] estimated the cost of cyber security threads to the world economy at 5.2 trillion USD lost over the next five years in the private sector alone.

But aside from malicious intent or outright erroneous designs, computers—and *systems* composed of networked computers—also need to be resilient to *electromagnetic interference* (EMI). Electromagnetic radiation is emitted by many sources, including electronic devices, power lines, and nuclear reactions, or can stem from solar wind or cosmic rays. Thus, computers will be exposed to various levels of it. While outright shielding from electromagnetic interference has been used for decades [Nie81], this comes with significant cost in terms of extra weight and space, which limits this approach when it comes to ubiquitous embedded systems. Another approach is the use of hardware that has been hardened against electromagnetic interference [dW08]. When a certain basic level of protection against electromagnetic interference is reached, ordinary electromagnetic interference does not cause any obvious problems to the system anymore. Of course, if a strong enough electromagnetic source, say, a strong electromagnetic pulse like from the detonation of a nuclear bomb, is nearby, most circuitry, even if hardened and moderately shielded, will still suffer from electromagnetic interference.

However, it becomes very hard to know whether problems that occur in the operation of a system containing embedded networked computers stem from electromagnetic interference or from other sources, as electromagnetic interference short of burning out circuits does not leave any traces. For example, Keith Armstrong [Fle10a] names electromagnetic interference with the drive-by-wire system as a likely culprit for accidents with self-accelerating cars.

Long wires basically act as antennae, which makes them especially susceptible to electromagnetic interference. Systems composed of several computers which are connected by long wires consequently need to be designed with electromagnetic interference in mind. Communication protocols that are resilient to transmission errors which have been introduced by electromagnetic interference are thus a necessary addition to shielding and hardening of the hardware, if safety critical applications like x-by-wire have to rely on correct operations of such systems.

1.3 Confidence in computerized systems

A *system* designed for a certain task or set of tasks, be it composed of a single computer or a network of computers, may its functionality be encoded in specialized hardware or software executed on general purpose hardware, is expected to perform its task with a certain reliability. To achieve this, it is carefully designed using state of the art

techniques. However, careful design alone is not enough to give us confidence that the system will in fact perform as is expected. The system needs to be *verified* if one is to have confidence that it actually meets the expectations. The most common form of doing so is to test the system. This means that the system is supplied with various inputs, and it is observed whether its outputs are as expected. A successful test just proves that the system behaves as expected for the particular input that was tested. As most useful systems, especially embedded systems, could be subject to an enormous range of inputs, testing it can only strengthen confidence in the system, but it usually cannot verify that the system will always behave as expected.

With the ever growing complexity of *systems*, the need for automatable methods of increasing confidence in their "correct" operation grows as well. In order to test many inputs, the testing needs to be done automatically, as manual testing quickly becomes unfeasible. This entails that a description of what it means for a system to behave as expected is needed, allowing automatically generated tests to check whether the system's output constitutes expected behavior or not. To capture the notion of expected behavior, the "correctness" of a system is defined using a *specification* which describes the system-behavior that is considered correct.

If this specification is written down using a formal language, like logic, it provides a formal notion of correctness of a system. This not only allows to automatically make sense of arbitrary tests, but also opens up the road to trying to prove a system to be correct for every input. This could be achieved by testing all possible inputs, but almost all systems have too many possible inputs to test each of them in a realistic amount of time with a realistic amount of resources. Most systems performing important tasks have so many possible inputs that it is impossible to test each of them even given unlimited resources, as a near infinite number of scenarios would have to be tested: It would need more time than our universe can provide, given our current knowledge of the universe. And if the range of potential inputs is infinite, testing simply cannot be used to cover all possible inputs in finite time at all.

Absent the possibility to test each individual input, one can turn to formal methods to try to check sets of inputs at once and thus cover all possible inputs. As the system can however only react to an individual input at a time, one cannot test the system itself that way. However, when a formal representation of the system or of a system behavior can be acquired, formal methods can then be used to formally establish its correctness with respect to the specification by covering all possible inputs, e.g., with a number of sets of inputs.

If the system is itself defined in a formal way, like in a programming language, this endeavor can make use of the system definition. However, many systems comprise physical components whose behavior first needs to be described in a formal way, delivering a model of their behavior. Even the behavior of the code of a computer program is only well defined if one assumes an underlying computational model of the system that is going to execute the code.

The model of the system does not need to treat every detail of its behavior, it only needs to give an abstracted view sufficient for the intended use. So a hardware model often does not need to treat voltages. Instead, the model then represents an

abstracted view described in terms of boolean logic. Whether the formal model accurately describes the behavior of a non-formal system is of course not formally proven, and strengthening confidence in the model remains an engineering task.

Given a model and a specification, the process of verifying the correctness of the model's behavior with respect to the specification can be formalized and, hopefully, automated as well. The process could take the form of a mathematical proof, as in deductive verification. Or, a model could be described as a directed control flow graph. This allows to look at the control flow through the model given the set of all possible inputs: Starting in all initial control states of the model, at every branch in the model one partitions the set with a certain property if the branch depends on the input having this property, and follows all branches with the respective sets. Thus, all path through the model could be followed, each with a set of inputs exactly fulfilling the requirements of the path leading to this branch and following it. If a set becomes empty, one can stop following it. If one comes back to a branch one has been to before and the set is a subset of the set one had when being at that branch before, one can also stop following this set, as one has already explored all paths from there. This process is an instance of *model checking*. Model Checking can cover all possible paths through a model thus discovering all possible behaviors. Given a finite model, this process can terminate, but will still often consume vast amounts of time and memory. Models for model checking will often be described in terms of *automata*, and the specification of correctness will either be described in terms of an automaton as well or some temporal logic statement over the path taken though the model's automaton.

Model checking requires efficient algorithms and data-structures, and the confidence in a model can be strengthened by using a sound modeling methodology. This thesis treats these pillars of verification, presenting novel approaches to each of them. The development of these approaches was driven by the bid to verify the FlexRay physical layer communication protocol, which will be used as a motivating example throughout this thesis.

1.4 Starting Ground of this Work

As described in Section 1.5, the problem of verifying real-time system models and, more specifically, asynchronous communication protocols like FlexRay, has been attacked from many angles before.

Temporal logics like *Computation Tree Logic* and *Timed Computation Tree Logic* or *Duration Calculus* have been used to describe systems and properties [OD08]. Process algebras like *Communicating Sequential Processes* (CSP) [Hoa78] have been employed [WKTZ05].

Manual proof efforts have been supported by theorem provers like PVS [ORSvH95] or Isabelle/HOL [NPW02, Pau94] and even model checkers like NuSMV [CCGR00]. Alas, manual efforts have been very labor intensive, often inflexible and sometimes reliant on unrealistic assumptions, like the absence of transmission errors due to external interference.

Transition systems like *Petri Nets* and *Statecharts* have been employed to model systems, and so have *Timed Automata* (see Chapter 3). The latter allow the usage of various automated verification tools like UPPAAL [BDL04] or KRONOS [Yov97]—but the complexity of their verification efforts rises exponentially with the number of clocks in the model.

Automated verification has shown a lot of promise, as it enables quick verification of changes to a system and reduces human error and human labor. For timed systems modeled in timed automata, the complexity of the model quickly limits the applicability of model checkers like UPPAAL and KRONOS, especially in terms of the timing complexity due to the inherent exponential complexity in the number of clocks, but also in terms of the discrete complexity, i.e., the number of locations and possible values of discrete finite variables—due to their semi-symbolic approach being vulnerable to explosion of the discrete state space.

These approaches only allow to verify systems of limited complexity or more complex ones on a rather simplified level, or have to rely on massive manual effort. The work presented here sets out to enable the automatic verification of complex systems, like the FlexRay physical layer protocol, using a timed automata model and model checking. The challenge is, thus, to reduce the complexity of the model without sacrificing precision through oversimplification or overly limiting assumptions, while also improving the resilience of automatic verification of timed automata against discrete complexity.

1.5 Related work[1]

The analysis of communication protocols is an active field. To name two works as an example, Brown and Pike [BP06] use SAL [MOR+04] to increase the degree of automation in proofs of the physical layer protocols 8N1 and Biphase Mark. Invariants of the latter are derived by Vaandrager et al. [VG06] using UPPAAL [BDL04]. These invariants enable a semi-automatic correctness proof using the PVS [ORSvH95] proof assistant. Both protocols are not designed for an unreliable physical environment, in contrast to the FlexRay physical layer protocol.

FlexRay has generated a lot of interest. In a comparison of FlexRay with the older bus protocol standards MIL-STD-1553 [Con00] and TTP/C [KG93], Srinivasan and Lundqvist [SL02] conclude that in real-time systems, FlexRay should in the long run become the protocol of choice.

The interest in FlexRay has spread beyond its automotive origins into the aerospace community: Both Paulitsch and Hall [PH08]—who discuss fault containment in FlexRay, propose the addition of a suitable bus guardian, and advocate further investigation into FlexRay's dependability—as well as Heller and Reichel [HR09] see FlexRay as a strong field bus candidate for use in an aeronautic context. The latter examine worst case scenarios of jitter and glitches[2] to derive signal integrity criteria for the per-

[1] This section contains parts already published in [GEFP10, GEFP12a, GEFP12b]

[2] *Jitter* refers to erroneous bit values introduced by the non-synchronicity interfering with hardware

formance of the physical layer. They use numerical simulation to evaluate the effects of increasing the maximal length of a cable between two communicating FlexRay nodes, which in planes quickly surpasses the 24 m FlexRay is designed for and that are sufficient in cars. Their evaluation of FlexRay using cable lengths of more than 100 m with added lightning-strike-protection circuits, as common in an aeronautic context, leads them to advocate the use of an alternative physical layer, such as RJ485, and adapting its interface to the FlexRay protocol. The work [GEFP12b, GEFP10, GEFP12a] which is presented in Chapters 10 and 11, provides with the model checking of a FlexRay model and its parameter analysis just the push button technology needed to re-evaluate the effect of changes in hardware assumptions such as those from a transfer from an automotive to an aeronautic environment.

The subproject *Automotive* of the *Verisoft* project attempted a first formalization [Böh06, Ger07] of FlexRay and its formal verification, resulting in a deductive "paper and pencil" proof-sketch presented in [BBG+05] that indicated correct message transfer by the physical layer protocol. The proof had to use the assumption of a reliable physical layer without glitches, which eventually lead me to abandon the deductive approach in favor of a model-checking based verification effort that does include glitches. The manual verification effort was however extended to a more comprehensive proof machine-checked with the interactive theorem prover Isabelle/HOL [NPW02, Pau94] by Schmaltz in [Sch06], which gave a formal model of communication between asynchronous hardware registers, and in [Sch07], which integrated this model with the bit clock alignment mechanism (see Section 2.2.2) proving correct message reception in the combined model, and demonstrating the error-proneness of pencil-and-paper approaches.[3] Parts of the proof were automated using the NuSMV model checker [CCGR00]. Knapp and Paul [KP07] outline a pervasive correctness proof combining the asynchronous hardware model with bit clock alignment and a clock synchronization similar to the network idle time adjustment in FlexRay (see Section 2.1.2) to demonstrate the collision freedom of their TDMA bus access scheme, which is very similar to the one employed in the static segment by FlexRay (see Section 2.1.1). Moreover, they integrate their FlexRay like serial *f-interface* with an *instruction set architecture* (ISA) level programming model for an electronic control unit (ECU) with such an f-interface. Alkassar, Böhm and Knapp [ABK08b] use Isabelle/HOL to provide a formal proof for the setup from [KP07] naming their ECUs with f-interfaces *automotive bus controllers* (ABCs). They focus on the correctness of their synchronization and their TDMA scheme, discharging many proof obligations to NuSMV, often combining it with the domain reducing IHaVeIt [TA08, Tve05] preprocessor. They emphasize the importance of taking real-time into account and lament the very low degree of automation available for their verification effort wherever con-

requirements. A *glitch* is an erroneous bit value spontaneously introduced due to unspecified reasons like electromagnetic interference. See Section 10.4 for a more detailed description of these phenomena.

[3] In [BBG+05], the strobecounter is reset to 1 (000) instead of the 2 (001) that would be required by the protocol. This lead to the 6th voted value being strobed instead of the 5th one. However, [Sch07] shows that only the 4th (which is strobed in [Sch07]) and the 5th (as required by the protocol) can be proven to be unaffected by jitter (in the absence of glitches).

tinuous real-time is involved. In [ABK08a] the same authors extend the ABC setup from [ABK08b] with a mechanism for initial synchronization during the startup, and for (re-)integration of ABC nodes into an ongoing communication in an ABC network. The startup and integration procedure is visibly inspired by FlexRay's integration method through listening to the bus (see Section 2.1.2). However, the differences are marked, as the whole startup machinery in FlexRay—that is used to avoid or at least detect and resolve collisions before establishing an approximation of a shared view of time—is replaced by individual timers for each node telling the nodes that first power up when they can safely initiate a synchronization, avoiding collisions by setting the timer to a sufficiently different time in each node. The other nodes do only integrate into the communication started by the first nodes. They also allow for nodes to stop sending or never start due to a failure of the node. This is compensated by doing away with designated synchronization nodes, and allowing one of the non-failing first powered up nodes to initiate synchronization. The introduction of possible failures comes with limitations, though, as erroneous sending, or failing of all first powered up nodes is excluded, and it is assumed that all non first powering up nodes know that they are not a first powering up node, even if there is not yet any communication on the bus. With these assumptions, a formal proof of correct message transmission between non-failing nodes in a cluster that has been started by only a subset of the constituent nodes and may contain gracefully failing nodes is provided and has been checked with Isabelle/HOL, discharging some proof-obligations to NuSMV. Endres, Müller, Shadrin and Tverdyshev [EMST10] give a short overview of the verification task and a test deployment on *field-programmable gate arrays* (FPGAs) for a distributed system in the form of a cluster of ECUs, as verified by Tverdyshev [Tve09], connected by ABCs. Such a cluster is used by Schmidt [Sch11] as the foundation for a small verified real-time operating system for the ECUs. Müller in his dissertation [Mül11] and together with Paul [MP11] combines all the results from [BBG+05, Sch06, Sch07, KP07, ABK08b], fixing and adjusting the previous results into a gate level description of an automotive bus controller which is deployed on an FPGA and a machine checked proof of correct message transmission in a correctly configured cluster of such controllers, leaving the incorporation of the extensions from [ABK08a] into their comprehensive setup to future work, which, to the best of my knowledge, has not been carried out yet. However, all these manual and semi-manual deductive proof efforts include jitter, but exclude glitches. Moreover, the manual effort required to re-prove parts of the overall proof if parameter assumptions where to be changed, e.g., in the hardware model, is non-negligible, as these proofs are semi-manual and not fully automatic. A fully automatic proof by model checking can in these cases quickly verify whether the requirements for proofs talking about the higher levels are still met after a change in assumptions on the hardware, the only manual effort required being the changing of the model, and even that could be easily automated for many parameters in a parameterized model, if the need to check a lot of different values would arise.

1.6 Previously published work

This thesis presents a modeling and model checking process that spanned a considerable amount of time. Of course, an effort of that magnitude was not undertaken alone. I owe a considerable dept of gratiude to my collaborators which worked with me on this effort, namely Hans-Jörg Peter and Rüdiger Ehlers, and to my advisor Bernd Finkbeiner.

This thesis thus contains parts that have previously been published in joint work together with Ehlers and Peter, and often Finkbeiner. In particular, the following publications are an integral part of the work presented in this thesis:

Model Checking the FlexRay Physical Layer Protocol [GEFP10], published together with Ehlers, Finkbeiner and Peter, presents the FlexRay model designated model † in Chapter 10, introduces the scenario that model † is based on, which is in this thesis described in Chapters 2 and 10, and presents the results obtained analyzing the model with 32-bit Uppaal version 4.0.6 (which could not handle the variants with an even bigger dicrete state space because it could not address more than 4 GiB of memory) which are presented in Chapter 11. While it is impossible to quantify exactly how much each author contributed to the ideas of the paper and their development, the reason I am the first author of the paper, deviating from the alphabetical order, is that I contributed all the central elements: the scenario and the model † itself and I also applied Uppaal to evaluate the model under changing parameters. However, without the contributions of my co-authors, the paper would never have reached the refinement neccessary for a successful publication.

Automatic Protocol Verification with Parametric Physical Layers [GEFP12a], also published together with Ehlers, Finkbeiner and Peter, is reporting on my work extending and improving the work presented in [GEFP10]. It presents the improved model designated model ‡ in Chapter 10, elaborates on the scenario the models † and ‡ are based on, here described in Chapters 2 and 10, and analyzes model ‡ with 64-bit Uppaal version 4.1.4 under variations of hardware parameters and with various glitch models, as presented in Chapter 11. That report was almost entirely my work, with my co-authors mostly providing parts carried over from [GEFP10] and helping with proofreading and some finishing touches.

FlexRay for Avionics: Automatic Verification with Parametric Physical Layers [GEFP12b], also published together with Ehlers, Finkbeiner and Peter, presents lessons learned from the modeling and verification effort, here presented in Chapter 10. It also describes the scenario on which the modeling effort is based, here described in Chapters 2 and 10, and reports some robustness results on FlexRay, here reported in Chapter 11. Again, I am the first author of that paper out of alphabetical order because I provided the scenario, the model and its analysis, as well as the lessons learned. My co-authors were indispensible in helping to present these lessons learned with the naive models introduced in that paper for explaining them, and in helping to bring the paper to a level of sophistication that waranted publication.

Making the Right Cut in Model Checking Data-Intensive Timed Systems [EGP10], published together with Ehlers and Peter, presents an algorithm for fully symbolic real-time model checking based on the *clock zone map* (CZM) data-structure, both of

which are presented in Chapter 6. Most of the paper, especially the algorithm itself, was really joint work. Certain parts owe more to contributions of some author than of the others: I'd say that the idea for counterexample generation owes most to Rüdiger Ehlers contributions, who's insights into BDDs were also invaluable. Hans-Jörg Peter, who was the driving force behind the endeavour to create an efficient fully symbolic real-time model checker in the first place, provided the framework for the prototype implementation and his expertise in programming made the prototype implementation of the approach efficient enough to be viable. My input was most important for the data-structure (which was developed from my work for my master's thesis [Ger10], there used in a semi-symbolic approach) and the FlexRay benchmark used in the evaluation of the algorithm. The work from [EGP10] needs to be included in this thesis because the approach presented in Chapter 7 extends it, so familiarity with it is necessary for understanding that chapter.

The work from [EGP10] is also extended to the data structure of *constraint matrix diagrams* (CMDs) in work published in *Fully Symbolic Timed Model Checking using Constraint Matrix Diagrams* [EFGP10] together with Ehlers, Peter, and Daniel Fass. However, the only part of that paper for which my input was more important than the input of the other authors is the FlexRay benchmark. Ehlers, Fass and Peter provided the most important input for most of that paper, so I only mention it in this thesis as the work which generated the most interest from the line of work based on the work presented in [EGP10].

Chapter 2

The Motivating Case Study: FlexRay Physical Layer Protocol

Contents

Abstract. The FlexRay bus protocol uses *time division multiple access* (TDMA) with a periodic schedule. Each iteration (*communication cycle*) of this schedule is divided into segments. The *static segment* is divided into *slots* of a fixed size during which exactly one node is allowed to write to the bus. The *dynamic segment* uses similar but very small slots that grow dynamically larger if they are used. As the size of the segments is fixed, the last dynamic slots are omitted if a slot in the dynamic segment is used and thus grows larger. Messages are send as *frames* in a predefined format, which allows synchronization of the local views of time with respect to the schedule. In the *physical layer protocol*, frames are sent as formatted bit streams where each bit is send for eight clock cycles as a so called *bit cell*. This allows the receiver, which samples the bus in each clock cycle, to synchronize its view of the position in the stream of sampled values with respect to the bit cell. The sample-stream is flattened into a stream of voted values by a majority vote in a window sliding over the sample-stream, and a voted value from the middle of each bit cell is chosen (*strobed*) to reconstruct the bit-stream that was sent. The verification of the resilience of this physical layer protocol against patterns of glitches (bit-flips) on the bus during the static segment will be the motivating example throughout this work.

2.1 Introduction to FlexRay

FlexRay is a bus protocol. It was developed by a consortium of mayor companies in the automotive industry and their suppliers. The core members of the consortium were BMW, Bosch, Daimler, Freescale, General Motors, NXP Semiconductors, and Volkswagen. The FlexRay protocol was intended to replace *controller area network* (CAN) and *time triggered protocol* (TTP) as the vehicle bus of choice. The FlexRay consortium was disbanded in 2009, and FlexRay is now standardized in the ISO norms 17458-1 through 17458-5.

As x-by-wire functionality was intended to be operated over the FlexRay bus, real-time guarantees and fault-tolerance where required.[1] FlexRay also provides a high data rate of 10 Mbit/s,[2] as usage for in-car entertainment systems and the like was also intended. As modern cars can have a large number of embedded computers, called *electronic control units* (ECUs) in the following, the FlexRay bus supports a flexible architecture of 2 channels, A and B, connecting the ECUs in a bus, star, or mixed bus and star architecture. Independent of the architecture, the communication medium will be referred to as *bus* in the following. The connected entities in a FlexRay network are called its *nodes*.

Sections 2.1.1–2.1.4 give a general description of the FlexRay protocol to provide context to the description of the FlexRay *physical layer protocol* in Section 2.2, the verification of which is the motivating case study driving the development of the approaches presented in this thesis, as described in Section 2.3.

[1] The first big industrial application of flexray was its usage in the pneumatic damping system of BMW's X5 in 2006, which paved the way for full utilization of FlexRay in BMW's 7 Series in 2008.

[2] Configuration parameters for slower bit rates of either 2.5 Mbit/s or 5 Mbit/s are also supplied in the FlexRay specification, see [Fle05, Appendix B.1].

2.1.1 Bus access organization[3]

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

FlexRay organizes communication into *communication cycles*, as explained in [Fle05, Chapter 5]. At any given time, the communication is essentially one way: one node sends and the others listen, or none is sending. The *timing hierarchy* of FlexRay is shown in Figure 2.1. Each communication cycle is divided into four segments: the *static segment*, the *dynamic segment*, the *symbol window*, and the *network idle time*. During the static segment, a static schedule applies, which gives only one node write-access to the bus at a time. This allows for hard worst case execution time estimates if communication is involved, because the access to the bus is guaranteed if the schedule says so. The same schedule is programmed into all nodes. This *static segment* of the cycle is followed by a *dynamic segment* where controllers are allowed to try to send a message outside the normal schedule. The dynamic segment allows communication based on priorities. Each node has its unique place in this hierarchy, and all nodes know this place, so the priority is already fixed during the configuration of the network. This enables nodes to send faster or sent more data, or even to send only if the bandwidth is available and so on. A node can try to access the bus, but its access could fail if nodes of higher priority already used up the available transmission time in the dynamic segment. The end of the cycle consists of a small *symbol window*, and finally a *network idle time*. The symbol window is reserved for communication related to establishing the communication in the first place or for re-establishing it. The network idle time is not used for communication, but acts as a buffer separating the communication rounds. The length of its network idle time can be adjusted by a node in order to re-synchronize the start of the next communication round with the rest of the network.

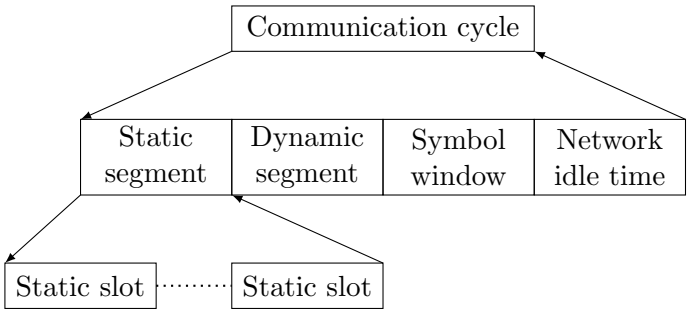


Figure 2.1: Schedule of the FlexRay communication cycle.[4]

[3]This section contains parts already published in [GEFP12b, Section II].

[4]This figure is inspired by [Fle05, Figure 5-1]. It is based on figures already published in [Ger05, Figure 2] and [Ger10, Figure 5.2]. It has already been published in [GEFP12b, Figure 1].

FlexRay uses three notions of time to formulate the makeup of a communication round.

The static segment is divided into *static slots*, each assigned to a node that has exclusive write-access to the bus during this slot, which always begins and ends with a small idle time to avoid collisions. The static segment thus provides access to the bus in a *time division multiple access* (TDMA) scheme. The dynamic segment is divided into *minislots* providing a *flexible* FTDMA scheme. Both the TDMA and the FTDMA exclude collisions [ABK08b, ABK08a].

The duration of static slots, the dynamic segment, the symbol window, and the network idle time is defined in terms of *macro ticks*. The dynamic segment's minislots have a minimum duration defined in terms of macroticks, but if a minislot is used for transmission, its length will be extended. If the initial idle time in a minislot is followed by a transmission, the length of the slot is extended until the transmission is finished and some idle time has been added in the end. Of course, the length of the transmissions is limited. And as the length of the dynamic segment is fixed, the number of minislots in a dynamic segment varies if communication takes place. The length of the idle times is also defined in terms of macroticks.

The duration of a macrotick is defined in terms of *microticks*. The clock synchronization of a node can change the number of microticks in a macrotick in a node when performing rate correction. The duration of a microtick is defined in terms of ticks of the sample clock. The number of *sample ticks* per microtick is configurable as either 1, 2, or 4 and depends on the bit rate and the duration of a sample tick, which is derived by multiplication with 1, 2, or 4 from the ticks of the oscillator. In the following, an oscillator with a standard rate of 80 MHz, a sample tick multiplier of 1, and a microtick length of 1 sample tick will be assumed, providing a 10 Mbit/s data rate.

2.1.2 Synchronization and startup

The local view of time of a node is synchronized with the other nodes by adjusting the number of microticks in its macroticks and by shortening or lengthening its network idle time, as detailed in [Fle05, Chapter 8]. Some nodes are designated as synchronization nodes. The arrival time of their messages is used to calculate the difference to the point in time at which they were expected to be received and adjust the local view of time accordingly.

When the network is powered up, some node will be commanded to wake up a channel, see [Fle05, Chapter 7]. The channel is woken up via sending a sequence of symbols, the *wakeup pattern*. This pattern contains long idle periods, which allow for collisions with other symbols to be detected and resolved. Nodes receiving a wakeup pattern will wake up and start to listen in to communication, which allows them to find out the global view of time. This happens when a synchronization node waking up the network performs a *coldstart*: It sends a *collision avoidance symbol* to establish that it is alone in waking up the network, and will then be the only node sending in the first four communication cycles. The information from its synchronization messages, namely which slot and cycle they were send in, will allow the other nodes to deduce an

approximation of the global view of time and join the communication. This mechanism also allows nodes started up later to join an ongoing communication by listening in long enough to make sure they do not disturb communication when they start to send in their alloted slots.

2.1.3 Frames[5]

Messages transmitted via FlexRay are packaged in a format called a *message frame* (or just *frame*), defined in [Fle05, Chapter 4]. The frames have a header and a payload section, and end with a 3-byte *cyclic redundancy code* (CRC) [PB61] calculated from the frame, which allows to detect transmission errors and discard corrupted messages. The 5-byte header—which is again protected by its own 11-bit cyclic redundancy code, allowing to verify the integrity of the most important parts of the header before the reception of the whole frame—contains meta data: The information in which slot and in which cycle it was sent, whether it is a startup frame or a synchronization frame and so on. The payload consists of up to 254 bytes of data. The format of a frame is shown in Figure 2.2.

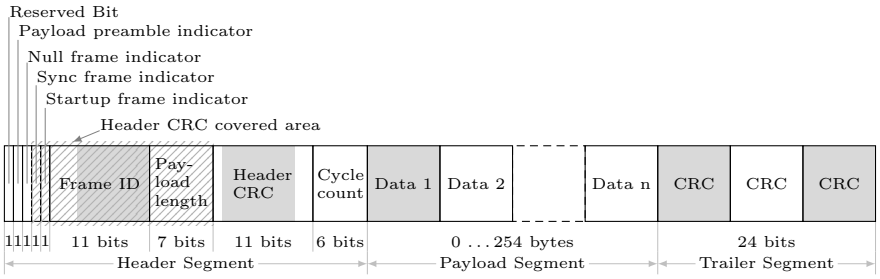


Figure 2.2: Format of a frame.[6]

2.1.4 Controller architecture[7]

The FlexRay controllers' architecture is divided into a *communication controller* attached to the host ECU, and one *bus driver* for each attached channel, connecting the communication media to the communication controller. A controller is based on a layered architecture comprising communicating processes. The architecture can be roughly divided in three layers to which the processes can be related: the *distribution-and-control layer*, the *communication layer*, and the *bit-level layer*. Figure 2.3 shows an overview.

The actual communication between two controllers is handled by the *coding and decoding processes* (CODEC) that drives the bus.

[5] This section contains parts already published in [GEFP12b, Section II].

[6] This figure is based on a figure already published in [Ger07, Figure 5.1], which, in turn, was based on [Fle05, Figure 4-1]. It has already been published in [Ger10, Figure 5.4] and [GEFP12b, Figure 3].

[7] This section contains parts already published in [GEFP12b, Section II].

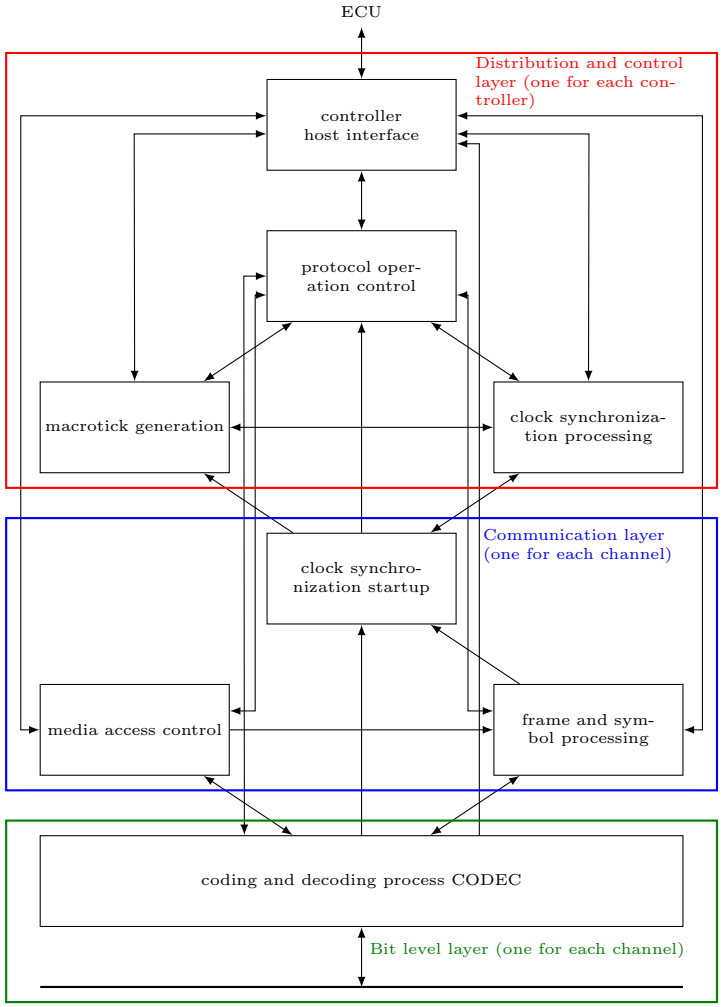


Figure 2.3: Overview on the architecture of a FlexRay controller.[8]

The bus driver is basically told whether to write to the medium and whether to write a high or a low bit, and reports whether the bus is currently high or low. In the following, the bus driver will be considered like a part of the bus, which can be written to and can be read.

The communication controller consists of:

- one *controller host interface* used for configuration of the node and for passing on either messages to be packaged in a frame and sent, or the payload of frames that have been received (see [Fle05, Chapter 9])
- one *protocol operation control* responsible for the internal mode of the node, like

[8]This figure is based on [Fle05, Figure 2-2]. It has already been published in [Ger10, Figure 5.3] and [GEFP12b, Figure 2].

`wakeup`, `normal active` and the like (see [Fle05, Chapter 2])

- one *clock synchronization processing* calculating the clock synchronization values (see [Fle05, Section 8.3])
- one *macrotick generation* supplying the local view of time (see [Fle05, Section 8.7])
- for each attached channel one *clock synchronization startup channel A (or B)* for integrating into a communication schedule during a startup procedure (see [Fle05, Section 8.4])
- for each attached channel one *frame and symbol processing channel A (or B)* for checking the integrity of frames and symbols and supplying status data about them (see [Fle05, Chapter 6])
- for each attached channel one *media access control channel A (or B)* for enforcing the schedule and the priorities with regards to access to the bus and for packaging messages into frames (see [Fle05, Section 5.2])
- for each attached channel one *coding/decoding process channel A (or B)* for generating the stream of bits that is written to the bus from the frames or symbols to be sent, and conversely extracting received frames or symbols from the received stream (see [Fle05, Chapter 3])

It is the last item, the coding/decoding process, that contains the *physical layer protocol* that will be used as a motivating example throughout this work.

2.2 FlexRay physical layer protocol

The physical layer protocol as described in [Fle05, Chapter 3] takes symbols or frames to be transmitted and generates a bit stream that is then written to the bus, bit by bit. The reverse process is used to extract symbols or frames from the received stream. The streams are generated with the help of pre-defined sequences.

2.2.1 Stream format[9]

The bit stream for a symbol is relatively straightforward, as every symbol corresponds to a fixed sequence. Frames on the other hand are more intricate.

A message frame is transmitted as a structured stream [Fle05, Section 3.2.1.1] of bits as shown in Figure 2.4. The encoding of a frame in a bit stream starts by prefacing it with a *transmission start sequence* (TSS), which consists of a sequence of low bits and precedes every transmission, and is followed by a *frame start sequence*. The length of the TSS depends on the structure of the overall network and may vary between 3 to 15 bits [Fle05, Sections B.2.1]. As the bus is high idle, a long enough sequence of low bits

[9] This section contains parts already published in [GEFP12b, Section II].

can be recognized no matter how badly synchronized the `strobecounter` values of the individual controllers are. 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, as the FSS is just inserted to make sure badly synchronized receivers still receive the first bit of the following sequence, which is also a high bit. Afterwards, all the bytes of the frame are attached, but each prefixed by 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 as described in Section 2.2.2.

The stream ends with a *frame end sequence* (FES), which is followed by a variable length *dynamic trailing sequence* (DTS) in the case of a frame sent during the dynamic segment. The FES consists of one low bit followed by one high bit.

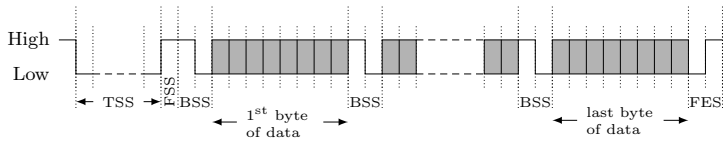


Figure 2.4: Format of a message bit stream.[10]

2.2.2 Stream transmission

In order to enable resilience against transmission errors, redundancy is added while transmitting the stream by sending each bit for eight sample ticks, effectively sending eight samples as one *bit cell* for each bit, or more precisely, pulling the bus to the corresponding value for eight consecutive sample ticks, as shown in Figure 2.5.

Voting

The receiver samples the value of the bus with each sample tick, producing a stream of samples. This stream is then smoothened by *voting*: Not the received sample stream is considered by the rest of the protocol, but the stream of *voted values* that are the result of a majority vote over the last five received samples. Thus, if in the middle of a stream of low samples, a single high sample is received due to an error, this value will never be visible to the rest of the protocol because there will be four low samples in the sliding *voting window* containing the last five received samples. However, this means that in the error-free case, a change on the bus will only be noted with a delay of two sample ticks, as three samples of the new value have to be received to form a majority against the old values in the voting window. In the absence of errors, this is not a problem as each bit is represented by eight consecutive identical samples in the

[10]This figure is based on [Ger10, Figure 5.7], which was based on [Ger07, Figure 4.2], which, in turn, was based on [Fle05, Figure 3-2]. It has already been published in [GEFP10, Fig. 2], [GEFP12a, Figure 4], and [GEFP12b, Figure 4].

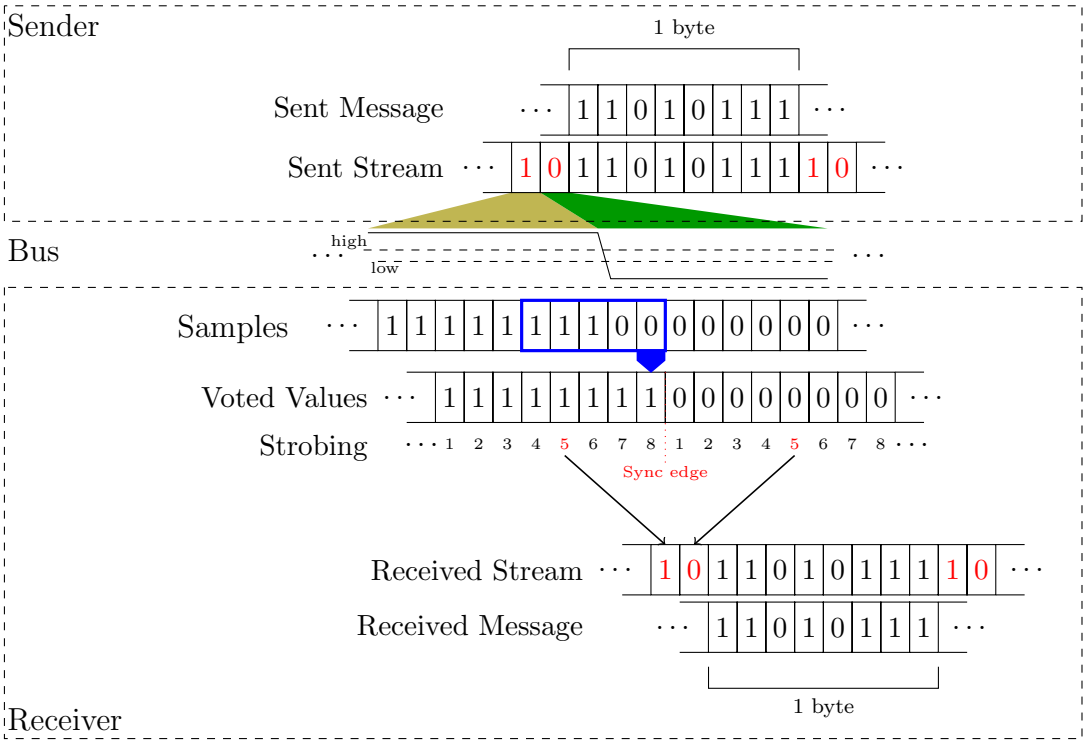


Figure 2.5: FlexRay physical layer protocol

sample stream. Note that this procedure consequently produces eight voted values for each bit.

Strobing

To reconstruct the individual bits, *bitstrobing* is used: From each *bit cell* of eight consecutive voted values corresponding to the same bit, the fifth voted value is picked (*strobed*) and the rest is discarded. In the absence of errors, the resulting stream of strobed values is identical to the generated stream that was sent, allowing to easily extract the sent symbol or frame.

Bit clock alignment

As only the macrotick view of time is synchronized and the physical layer protocol operates on the basis of sample ticks, the communication cannot assume synchronization. Thus, to identify the borders of the bit cells, simple counting of local sample ticks is not sufficient, as the local oscillators could be imperfect to a certain degree.[11] Addi-

[11] [Fle05, Table A-1] specifies that the oscillators may not deviate from their standard rate by more than 0.15 %, which is the value assumed in the following. However, [Fle06b, Table 12-3] and [Fle06a, Section 3.7.1] are more restrictive, assuming a deviation of no more than 0.05 %, which they claim to be achievable by using high quality components.

tional synchronization measures are necessary, giving rise to the *bit clock alignment* mechanism. This mechanism deduces the start of certain bit cells from the stream and adjusts the value of the *strobecounter*, which is used for counting the voted values in a bit cell in order to find the fifth one. More precisely, when a *bit synchronization edge* is detected in the stream of voted values as shown in Figure 2.5, the last received voted value of this edge pattern (i.e., the value after the synchronization edge) is considered the first of a new bit cell and the strobecounter is reset to the appropriate value to reflect this. This edge pattern consists of a high voted value followed by a low voted value, and edge detection is enabled during the reception of high bits outside the actual frame bytes, i.e., during the inserted sequences of the stream. Byte start sequences, which consist of a high bit followed by a low bit, do contain such an edge, so during frame decoding, bit clock alignment is performed before the reception of each byte. During symbol reception, bit clock alignment is of minor importance, as the patterns allow recognition of a symbol even if a certain distortion due to non-aligned bit clocks occurs.

Error resilience claim

The event of a value on the bus that was not sent, like a high value sampled from a bus during the sending of a low sequence, is called a *glitch*. The FlexRay specification [Fle05, Section 3.2.7] claims that the physical layer protocol attempts to enable resilience against one glitch that is not longer than a sample clock period in a bit cell. But it then goes on and explains in a footnote that sometimes two such glitches in a bit cell can be tolerated, but also sometimes none.

This gives rise to the motivating example, namely using formal methods to make this claim more precise.

2.3 Verification of the physical layer protocol

The work together with Rüdiger Ehlers, Bernd Finkbeiner and Hans-Jörg Peter described in Chapters 10 and 11 and published in [GEFP10], resulted in a formal verification of some of my models of the physical layer protocol which include an error model that assumed a pattern of glitches a lot more precise than the vague error resilience statement from [Fle05, Section 3.2.7] using the Uppaal tool [BDL04].

2.3.1 Results on the reliability of the physical layer protocol

If only one sample in any sequence of four consecutive samples received from the bus before voting (see Section 2.2.2) can be nondeterministically affected by a glitch, an absence of errors in the received message stream was verified. Another pattern against which resilience was verified was two arbitrary glitch affected samples in the whole stream, including close together. A relationship between the possible error patterns and the size of the voting window, which is suggested by insight into the protocol, could also be quickly verified by minor changes to the model: if only one glitch can

occur in a sequence of consecutive samples, this sequence can be as short as 3 for a voting window of size 3 (which will not tolerate 2 arbitrary glitches), but needs to be at least 5 or 6 for voting window sizes of 7 or 9, respectively. Moreover, the error resilience does not only depend on the glitch patterns, but also on the hardware parameters that govern the drift between the local oscillators of the sender and receiver and govern the process of information transmission on the bus, which combine to another source of errors, called *jitter*. Binary search was employed to find the most liberal constraints on certain parameters that will leave error resilience against specific glitch patterns intact, if that hardware parameter varies from the standard configuration, as documented in [GEFP10].

These results were extended in work with the same co-authors and published in [GEFP12a, GEFP12b], as described in Chapters 10 and 11. The capability of the newer 64-bit versions of Uppaal [BDL+11] to address more memory and access to a computer with 2 orders of magnitude more memory allowed to model check a variety of models with different glitch pattern models. As an improved result, the presence of 2 glitches[12] in each sequence of 88 consecutive samples was found to leave error resilience of the protocol intact. Analyzing the robustness of the protocol with a given glitch model to changes in the parameters allowed to determine that the glitch model with 2 arbitrarily placed glitches in 88 consecutive samples has a more constraining effect on the tolerance towards parameter variation than a model with 2 adjacent glitches in 88 consecutive samples. Moreover, the introduction of a model with a glitch model formulated in terms of real-time durations instead of affected samples replicated the behavior of the model with a glitch model formulated in terms of affected samples: the durations of the acceptable glitches turned out to be exactly such that at most 1 (or 2 resp.) samples could be affected, and the unaffected period needed to be so long that the minimum of unaffected samples was 3 (or 86 resp.), or 87 samples at most affected by one more glitch, depending on whether it was a 1-in-4 or a 2-in-88 scenario.

2.3.2 Family of Benchmarks

During the effort of verifying the FlexRay physical layer protocol, several models with different glitch, jitter, or message models were created and verified.[13] This gave rise to a family of FlexRay model checking benchmarks. The benchmarks with explicit message length have a very high discrete complexity, but come with a more optimistic error model that has jitter covered by the glitch patterns as well. This reduces continuous complexity at the price of not being able to explore scenarios where jitter and glitches combine to bring about a protocol error. However, being able to scale the discrete complexity by scaling the number of bytes in the message makes this benchmark especially suited for the evaluation of the abilities of reachability checking approaches to handle the explosion of the discrete state space. The benchmarks with separate

[12] Here and in the following, *glitch* is used to refer to a glitch affected sample when speaking about discrete sample glitches, in order to improve readability.

[13] The timing behavior of the hardware is modeled on values taken from the Nangate Open Cell Library [Nan09].

jitter and glitch models reduce discrete complexity by abstracting from the length of the message. This might introduce spurious errors in the sense of finding actually safe configurations of the error model unsafe by looking at scenarios that exploit message lengths longer than allowed by the FlexRay specification, possibly even infinite. They also might result in spuriously too pessimistic hardware parameter variation tolerance results for the same reasons, thus possibly underestimating the resilience of the FlexRay protocol. However, positive verification results from these benchmarks do also hold for all scenarios with message lengths legal according to the FlexRay specification. The glitch model is either discrete, counting the affected and unaffected samples, or continuous, measuring the duration of the glitches and the time between them, thus either having more discrete or more continuous complexity. When many consecutive unaffected samples need to be counted, the model gives rise to a considerably sized discrete state space.

2.3.3 Explosion of discrete state space

To verify that the sent message is the received one, memorizing the whole message is unpractical as it is way too much data. Memorizing the bits in transfer, i.e., the bits sent but not yet received, would be better but is still data-intensive. Moreover knowing how much bits can be in transfer requires considerable insight into the protocol. However, as reachability checking will be used, it is enough to have the possibility that a wrongly transmitted bit is being checked for correctness. So it is enough to memorize a single bit, and its position: the position inside the byte, and the number of the byte. However, counting the number of bytes leads to a considerable blowup if the discrete state space is stored explicitly, prohibiting a successful verification with Uppaal and reinforcing the need for a fully symbolic approach, as shown in [EGP10, Table 1]. The amount of insight into the protocol model required to sidestep this problem by using a nondeterministic number of bytes as done in [GEFP12a, Section III.B] is considerable, e.g., the issue of overtaking of bytes needs to be addressed if only the position of a bit in a byte is stored, but the byte itself comes without an identifier, and there a more scenarios represented in the resulting model, which could have introduced spurious errors, thus making measured tolerances more pessimistic.

Counting samples also increases the discrete state space. In case of a 2-in-88 scenario, this can lead to a state space explosion that makes an explicit representation of the state space extremely memory intensive, quickly beyond the point that computers for everyday use can handle nowadays. Furthermore, access to specialized computers with huge amounts of memory can only mitigate the problem for a while, as additional increases in discrete complexity could lead to a state space explosion that pushes the memory requirements beyond any bound. Moreover, models taking more of FlexRay into account than just the physical layer protocol, or other models of industrial systems could have more discrete complexity. In the end, it is not only the complexity of the models that limits the practicability of verification efforts, it is also the limits of the complexity that the verification technology can handle that limit the complexity of the models that are developed.

Thus, it is highly desirable to have push button technologies that are able to handle more data intense models, as presented in [EGP10]. Whether such models are inherently so data intensive or this intensity could be decreased with more insight into the particular model as proposed in Part III is usually not obvious. So the possibility of reducing the data-intensiveness by embarking on an effort of model optimization with unclear returns is by no means a substitute for algorithms and data-structures, as presented in Part II, that are able to do fully automatic reachability model checking on data-intensive models.

Part I

Preliminaries

Table of Contents

Chapter 3

Timed Automata[1]

Timed automata [ACD90, AD94] are basically finite state machines extended with nonnegative real-valued *clock* variables. A detailed presentation of timed automata can be found in the textbook by Clarke et al. [CGP01, Chapter 17].

Formally, a timed automaton is a tuple $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$, with a finite set of locations L, an initial location $l_0 \in L$, a function $I : L \to \mathcal{C}(X)$ mapping each location to an invariant, a finite set of actions Σ, a transition relation $\Delta \subseteq (L \times \Sigma \times \mathcal{C}(X) \times 2^X \times L)$, and a finite set of real-valued clocks X, where $\mathcal{C}(X)$ is the set of clock constraints over X. A clock constraint $\varphi \in \mathcal{C}(X)$ is of the form

$$\varphi = \textbf{true} \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

where c is a constant in $\mathbb{N}_0$ and x is a clock in X. If $I(l) = \textbf{true}$ for all locations $l \in L$, the timed automaton is called *invariant-free*.

A *discrete transition* $\delta = \langle l, a, \varphi, \lambda, l' \rangle \in \Delta$ has a source location l, an action a, a *guard* φ, a *reset* set λ, and a target location l'.

The example of a timed automaton shown in Figure 3.1 has the locations l_1, l_2, l_3 and l_4, the clocks x and y, and the actions a, b and c. Its initial location is l_1. The locations l_1 and l_3 have the invariants $y \leq 5$ and $y \leq 15$, the other locations show no invariants, which means their invariant is **true**. The edge between l_1 and l_2 is labeled with action a, has a reset for clock x, and no guard, which is the same as saying its guard is **true**. The edge between l_2 and l_3 is labeled with action c, has no resets, which means its *reset set* is empty, and has the guard $x \geq 4$.

Clocks are assigned a nonnegative value by a *clock valuation* $\vec{t} : X \to \mathbb{R}_{\geq 0}$ which can be represented by an $|X|$-dimensional vector $\vec{t} \in \mathcal{R}$ (with $\mathcal{R} = \mathbb{R}_{\geq 0}^X$ denoting the set of all clock valuations).

A *state* of a timed automaton is a pair $(l, \vec{t})$ of a location and a clock valuation. As the domain of the clocks is continuous, there are uncountably infinitely many states. The set of all states is called the *state space*.

[1] This chapter contains parts also published in [EGP10, Section 2.1][GEFP12b, Section III.A].

The state of a timed automaton is changed by a transition, which can be of two types:

- In a *timed transition* (or *delay transition*), the same nonnegative value is added to all clocks, making sure no invariant is violated. Formally, $a \in \mathbb{R}_{\geq 0}$ is added to all clocks such that, for each $0 \leq d \leq a$, $\vec{t} + d$ satisfies the location invariant $I(l)$. The timed transition is denoted by $(l, \vec{t}) \xrightarrow{a} (l, \vec{t} + a \cdot \vec{1})$.
- In a *discrete transition* (or *action transition*), the location of the automaton can be changed and clocks can be reset to zero, if the target's invariant and the transition's clock constraint are fulfilled. Formally, for some $a \in \Sigma$, $(l, \vec{t}) \xrightarrow{a} (l', \vec{t}')$ is a transition $\delta = \langle l, a, \varphi, \lambda, l' \rangle$ of Δ such that $\vec{t}$ satisfies the clock constraints of the guard φ of δ, and $\vec{t}' = \vec{t}[\lambda := 0]$ is obtained from $\vec{t}$ by resetting the clocks in λ to 0, and $\vec{t}'$ satisfies the target's location invariant $I(l')$.

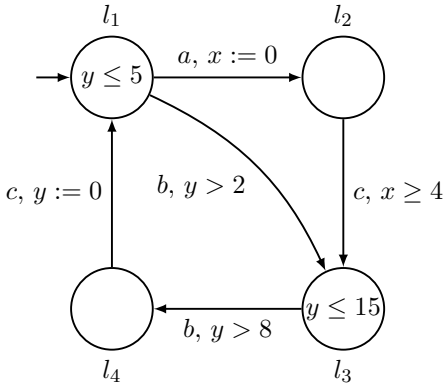


Figure 3.1: Example timed automaton with the locations l_1, l_2, l_3 and l_4, the clocks x and y, as well as the actions a, b and c.[2]

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

For the example timed automaton in Figure 3.1, all executions start in l_1 with $x = y = 0$. Due to l_1's invariant, all executions have to leave l_1 within 5 time units. During that time, the action transition leading to l_2 can be executed, resetting x. After this reset $x \leq y$ holds at l_2 and time can elapse arbitrarily. When at least 4 time units have passed after the reset, the action transition from l_2 to l_3 can be executed unless $y > 15$, in which case the target invariant would not be satisfied.

Formally, the words of the *language* accepted by the automaton $\mathcal{A}$ are finite sequences of transitions $a_1 \dots a_n \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$ such that there is a path $s_0 \xrightarrow{a_1}$

[2]This figure has already been published in [GEFP12b, Figure 5].

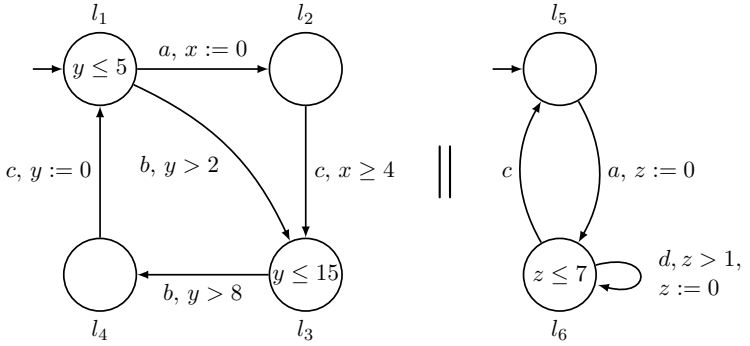


Figure 3.2: Example network comprising two timed automata that synchronize on actions a and c.[3]

$s_1 \dots s_{n-1} \xrightarrow{a_n} s_n$ where $s_0 = (l_0, \vec{t_0})$ is the initial state of the automaton, i.e., l_0 is the initial location and $\vec{t_0} = \vec{0}$ is the zero vector, and for all indices $1 \leq i \leq n$, the individual $s_i = (l_i, \vec{t_i})$ are states of the automaton, and all the $s_{i-1} \xrightarrow{a_i} s_i$ are transitions of $\mathcal{A}$.

As a shorthand, $s_0 \longrightarrow^* s_n$ denotes the existence of a finite sequence $a_1 \dots a_n \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$ of transitions with $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$. A state s is called *reachable* in $\mathcal{A}$ iff $s_0 \longrightarrow^* s$ is in the language of $\mathcal{A}$.

3.1 Composition

Timed automata can be composed into networks. In a network of timed automata, the component automata run in parallel and synchronize on shared actions. The locations of the resultant automaton are the tuples of locations from the *component* automata, thus its initial location is the tuple containing the initial locations of each component automaton. In a composed location, the invariants of each component location have to hold. As the action sets are joined, the transition relation contains the discrete transitions that change only one component location for the actions belonging to one component automaton only, and joint discrete transitions that simultaneously change each component location of the automata that share the action with that shared action.

Figure 3.2 shows an example network of two timed automata, which synchronize on actions a and c. Provided that the clock guards are satisfied, an automaton can only execute an edge labeled with action a or c if the other automaton executes an edge with the same action concurrently. However, if in one of the component automata, the guards of all edges labeled with action a leaving the current state are not satisfied, the other automata cannot execute any edge labeled with action a, even if its guard is satisfied. For the actions they don't share, the automata run asynchronously in the sense that they can independently execute edges labeled with action b or d.

[3]This figure has already been published in [GEFP12b, Figure 6].

Formally defined, for two timed automata $\mathcal{A} = (L_1, l_0^1, I_1, \Sigma_1, \Delta_1, X_1)$ and $\mathcal{A}' = (L_2, l_0^2, I_2, \Sigma_2, \Delta_2, X_2)$ with disjoint clock sets $X_1 \cap X_2 = \emptyset$, the *parallel composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is the timed automaton $(L_1 \times L_2, (l_0^1, l_0^2), I, \Sigma_1 \cup \Sigma_2, \Delta, X_1 \cup X_2)$, where $I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ for all $l_1 \in L_1$ and $l_2 \in L_2$, and Δ is the smallest set that contains

- for $a \in \Sigma_1 \cap \Sigma_2$, $\langle (l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2') \rangle$ if $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$ and $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$,
- for $a \in \Sigma_1 \setminus \Sigma_2$, $\langle (l_1, l_2), a, \varphi_1, \lambda_1, (l_1', l_2) \rangle$ if $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$, and
- for $a \in \Sigma_2 \setminus \Sigma_1$, $\langle (l_1, l_2), a, \varphi_2, \lambda_2, (l_1, l_2') \rangle$ if $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$.

In Part II, only the *global timed automaton* obtained from the composition of the system's component automata is considered.

As technicalities that just occur in the construction of the symbolic discrete transition relation, control-related concepts such as synchronization, parallel composition, or integer variables do not have to be considered in the actual model checking procedure.

3.2 Extended Timed Automata Syntax[4]

Extended timed automata [BDL04] are derived from the classical timed automata, but introduce features like integer variables and special synchronization. Each transition can have an *update expression* to set integer variables in addition to the usual clock resets.

Instead of actions, 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. This allows for normal synchronization as described above and used, e.g., in model ‡ in Chapter 10. However, it also enables the use of *broadcast channels*, where the sender can take the synchronization edge whenever its guard and the invariants permit it, irrespective of the enabledness of receiver transitions, which take the synchronous transition if and only if their guards and invariants permit it when the sender takes the broadcast synchronizing transition. The latter model of synchronization is used in model † in Chapter 10, which also uses a special feature of extended timed automata, so called *committed locations*. These locations are marked with the special invariant `C`, and have to be left before time passes or any non-committed location is left.

Figure 3.3 gives a brief overview of the syntax which will be used in Part III. To improve the readability of complex models, they will be split into segments using the auxiliary syntax shown in Figure 3.4.

[4]This section contains parts already published in [GEFP12a, Section III.A].

Figure 3.3: Syntax of locations and transitions in extended timed automata[5]

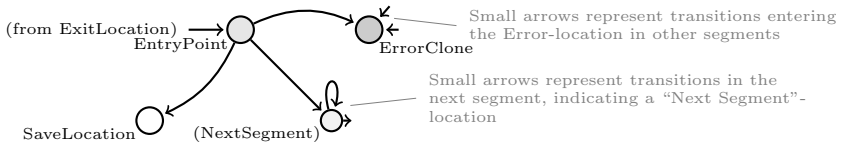


Figure 3.4: Syntax used to describe splitted automata[6]

- **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 3.4, the location **EntryPoint** is also the "ExitLocation" of this segment, as there is a transition from **EntryPoint** to **NextSegment**.

3.3 Finite Semantics

The reachability problem of timed automata is decidable if there exists a *region equivalence relation* [AD94] on $\mathcal{R}$ with a finite index.

Given a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$, the value of a clock $x \in X$ is called *maximal* if it is strictly greater than the *clock ceiling*, i.e., the highest constant c_{max}

[5]This figure is based on a figure already published in [Ger10, Figure 2.1]. It has already been published in [GEFP12a, Figure 2].

[6]This figure is based on a figure already published in [Ger10, Figure 2.2]. It has already been published in [GEFP12a, Figure 3].

any clock is compared to. Note that two different maximal clock values for the same clock thus cannot be distinguished by the clock constraints of discrete transitions or invariants.

The region equivalence relation groups states that can only be distinguished by sequences of transitions starting there by looking at the exact $a_i \in \mathbb{R}_{\geq 0}$ that are added in the timed transitions. Formally, two clock valuations $\vec{t}_1, \vec{t}_2 \in \mathcal{R}$ are in the same *clock region*, denoted $\vec{t}_1 \sim_R \vec{t}_2$, if

- In $\vec{t}_1$ and in $\vec{t}_2$ the same clocks have maximal value. Formally, $\forall x \in X : \vec{t}_1(x) > c_{max} \Leftrightarrow \vec{t}_2(x) > c_{max}$, and
- $\vec{t}_1$ and $\vec{t}_2$ agree (1) on the integer parts of the clock values, (2) on the relative order of the non-integer parts of the clock values, and (3) on the equality of the non-integer parts of the clock values with 0. Stating it formally: For two valuations $\vec{t}_1$ and in $\vec{t}_2$ in a clock region, for all clocks x and y with non-maximal value, it holds that
  - (1) $\lfloor \vec{t}_1(x) \rfloor = \lfloor \vec{t}_2(x) \rfloor$,
  - (2) $\widehat{\vec{t}}_1(x) \leq \widehat{\vec{t}}_1(y) \Leftrightarrow \widehat{\vec{t}}_2(x) \leq \widehat{\vec{t}}_2(y)$, and
  - (3) $\widehat{\vec{t}}_1(x) = 0$ if, and only if, $\widehat{\vec{t}}_2(x) = 0$,

  where $\widehat{\vec{t}}_i(x) = \vec{t}_i(x) - \lfloor \vec{t}_i(x) \rfloor$ for $i \in \{1, 2\}$.

The clock region $\vec{t}$ belongs to is denoted with $[\vec{t}]_R = \{\vec{t}' \in \mathcal{R} \mid \vec{t} \sim_R \vec{t}'\}$. Two states $s_1 = (l_1, \vec{t}_1)$ and $s_2 = (l_2, \vec{t}_2)$ of $\mathcal{A}$ are *region-equivalent* ($s_1 \sim_R s_2$), if (a) their locations are the same ($l_1 = l_2$) and (b) their clock valuations are in the same clock region ($\vec{t}_1 \sim_R \vec{t}_2$). The equivalence class of region-equivalent states that $(l, \vec{t})$ belongs to is denoted with $[(l, \vec{t})]_R = \{(l, \vec{t}') \in L \times \mathcal{R} \mid \vec{t} \sim_R \vec{t}'\}$.

As noted above, sequences of discrete transitions are not affected by a region based abstraction of a timed automaton. Region abstraction is thus a very suitable semantics, essentially not changing the language of the abstracted timed automaton. Formally, if there is a discrete transition $s \xrightarrow{a} s'$ from a state s to a state s' of a timed automaton, then there is, for all states r with $r \sim_R s$, a state r' with $r' \sim_R s'$ such that $r \xrightarrow{a} r'$ is a discrete transition with the same label $a \in \Sigma$. Moreover, a slightly weaker property holds for timed transitions: if there is a timed transition $s \xrightarrow{t} s'$ from a state s to a state s', then there is, for all states r with $r \sim_R s$, a state r' with $r' \sim_R s'$ such that there is a timed transition $r \xrightarrow{t'} r'$ with $t, t' \in \mathbb{R}_{\geq 0}$. The only really observable change is that $t' \neq t$ is possible, as mentioned before.

The *finite semantics* of a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ is thus defined as the finite graph $[\![\mathcal{A}]\!] = (S, s_0, T)$ where

- the symbolic state set $S = \{[(l, \vec{t})]_R \mid (l, \vec{t}) \in L \times \mathcal{R}\}$ of $[\![\mathcal{A}]\!]$ is the set of equivalence classes of region-equivalent states of $\mathcal{A}$, with
- the initial state $s_0 = [(l_0, \vec{t}_0)]_R$, and

- the set $T = \{(s, s') \in S \times S \mid \exists r \in s,\ r' \in s', a \in \Sigma \cup \mathbb{R}_{\geq 0}.\, r \xrightarrow{a} r'\}$ of transitions.

Alur and Dill have shown the finite semantics to be reachability-preserving:

Lemma 3.3.0.1. *[AD94] For a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ there is a finite path from a state $(l, \vec{t})$ to a state $(l', \vec{t'})$ if, and only if, there is a finite path from $\left[(l, \vec{t})\right]_R$ to $\left[(l', \vec{t'})\right]_R$ in $[\![\mathcal{A}]\!]$.*

3.4 Clock Zones

To obtain a coarser finite representation of $\mathcal{R}$, *clock zones* [Alu98] can be used.

A clock zone represents a set of valuations that satisfy certain clock difference constraints. A special clock x_0, which is always evaluated to zero, is introduced to allow direct comparison of a clock to an integer constant by using this *zero-clock* in a clock difference constraint. Formally, a clock zone z is represented by a conjunction of clock difference constraints of the form $x - y \prec_{x,y} c_{x,y}$, where $x, y \in X \cup \{x_0\}$, for an $x_0 \notin X$, $\prec_{x,y} \in \{\leq, <\}$, and $c_{x,y} \in \mathbb{Z} \cup \{\infty\}$. A clock valuation $\vec{t}$ satisfies z, written as $\vec{t} \in z$, if $\vec{t'} = \vec{t} \cup \{x_0 \mapsto 0\}$ satisfies each constraint $x - y \prec_{x,y} c_{x,y}$ from z: $\vec{t'}(x) - \vec{t'}(y) \prec_{x,y} c_{x,y}$. The set of all clock zones is called $\mathcal{Z}$ in the following.

To represent clock zones, *difference bound matrices* (DBMs) [Dil89] can be used. Essentially a DBM is a matrix with dimension $|X \cup \{x_0\}|$ whose entries represent the upper bounds on the difference between the clocks indicated by the indices. For two clock zones $z, z' \in \mathcal{Z}$, DBMs allow an efficient implementation of the operations

(1) intersection $z \wedge z'$, where for each clock difference the stricter of the two upper bounds from z or z' is taken

(2) clock reset $z[\lambda := 0]$, where the differences with a clock in λ are replaced by differences with x_0, and

(3) elapsing of time $z^{\Uparrow}$, where the upper bounds on clocks are removed.

Bengtsson describes these and other DBM operations in more detail in [Ben02].

To use this in forward reachability analysis, a maximal constant extrapolation is implicitly applied after each time elapse to ensure termination. The initial clock zone that only contains the initial clock valuation $\vec{0}$ is denoted with z_0.

Chapter 4

Binary Decision Diagrams[1]

As symbolic data-structure for sets of locations, *reduced ordered binary decision diagrams* (BDDs) [Bry86, BCM+92], which represent boolean functions $f : 2^V \to \mathbb{B}$ for some finite set of variables V, are well established in formal verification. A BDD is a directed acyclic graph with a single root and two leaves: the **true** leaf and the **false** leaf. Its root node and inner nodes are each labeled with the name of a variable. Each non-leaf node has two outgoing edges, where one edge is labeled with a 1 and the other is labeled with a 0. A BDD can be read starting from its root as a series of questions on the values of the variables the nodes are labeled with, finally leading to a leaf that answers with the function's value: In a node, depending on the value of the variable it is labeled with, which is either 1 or 0 in the argument of the represented function, take the edge labeled with the variables value. In the next node, do the same, until you reach either the **true** or the **false** leaf, which represents the value the function returns for the variable assignment used. Burch et al. give more details on BDDs in [BCM+92]. However, for the present work, it is sufficient to keep in mind that BDDs can represent boolean functions as explained in the following, and that the efficiency of the representation depends on the variable order.

Given two boolean functions (BF) f and f', represented by BDDs, for all $x \subseteq V$ their conjunction is defined as

$$(f \wedge f')(x) = f(x) \wedge f'(x),$$

their disjunction as

$$(f \vee f')(x) = f(x) \vee f'(x)$$

and their negation as

$$(\neg f)(x) = \neg(f(x)).$$

This provides the basic boolean operations.

Given a set of variables $V' \subseteq V$ and a BF f, the existentially quantified function $\exists V'.f$ is defined as the function that maps all $x \subseteq V$ to **true** for which there exists some $x' \subseteq V'$ such that $f(x' \cup (x \setminus V')) = \textbf{true}$. This allows to define a boolean

[1] This chapter contains parts also published in [EGP10, Section 2.2].

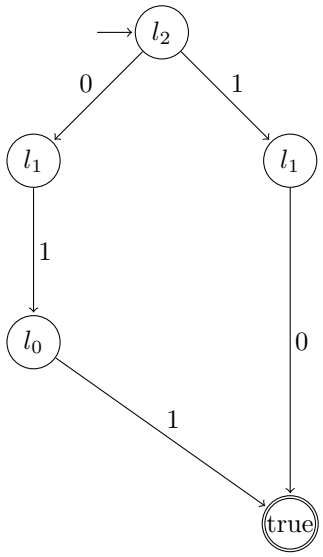


Figure 4.1: Example BDD for $l \in \{101, 100, 011\}$.

function that just ignores the existentially abstracted variables from the function it was derived from, and only looks at the not-existentially-abstracted variables there and asks whether it can be **true**. One can think of it as following both edges in the original function's graph whenever an existentially abstracted variable is associated with a node—and if at least one of the path leads to **true**, the derived function will return **true**.

Given two ordered lists of variables $L = l_1, \ldots, l_n$ and $L' = l'_1, \ldots l'_n$ of the same length, the renaming $f[L/L']$ of L' to L in f is defined as the BF for which some $x \subseteq V$ is mapped to **true** if and only if $f((x \setminus \{l'_1, \ldots, l'_n\}) \cup \{l_i \mid \exists 1 \leq i \leq n : l'_i \in x\}) = \textbf{true}$. This allows to rename several variables in a functions definition at the same time by providing two lists, were each variable in the list L' is renamed to the variable name listed at the same position in list L.

Visualized as a directed acyclic graph, the BF representing a set of locations l could yield a BDD like the one shown in Figure 4.1. Representing the locations $l \in \{5, 4, 3\}$ in binary yields $l \in \{101, 100, 011\}$. The bits of these binary representations can be represented by boolean variables l_i for $i \in \{0, 1, 2\}$, numbered starting with the least significant bit. Note that the **false** leaf of the BDD does not need to be pictured: If an edge for some value of a variable is not shown, it means that it leads to **false**, yielding a smaller, less cluttered, and thus easier to read visualization.

With respect to a variable order, BDDs are canonic. The size of a BDD can vary dramatically with the variable order. Thus, BDD libraries usually bring their own variable ordering and reordering heuristics.

As bounded discrete variables can easily be represented by a few boolean variables, sets of valuations of a bounded discrete variable can be efficiently represented as a boolean function, and hence can be stored in a BDD.

Part II

Data-Structures and Algorithms

Table of Contents

Chapter 5

State-Space Representation and Exploration

Contents

Abstract. This chapter discusses various possibilities to represent the state-space of a timed automaton and their interaction with state-space exploration techniques. To combat the state space explosion problem, infinitely many states can be collected in a set of states described by constraints, e.g., *clock zones* for convex sets of clock valuations. Sets of enumerable states can for example be represented by *binary decision diagrams*. Such symbolic representations are best found separately for the continuous and the discrete aspects of a state and then combined to a fully symbolic representation, e.g, by mapping *difference bound matrices* (DBMs) to *reduced ordered binary decision diagrams* (BDDs). An exploration algorithm working on sets of states were the discrete part is explicit can work efficiently even on complex aspects of the model like transitions enriched with code, and store the already explored states fully symbolically. A fully symbolic algorithm executes symbolic operations on fully symbolically represented sets of states, increasing the workload for a single step while reducing the number of steps. There often is a tradeoff between memory and time consumption when deciding between semi-symbolic or fully symbolic approaches. Exploration algorithms use a *post* operator to find the set of states reachable with a certain set of transitions from a set of reachable states, which is described in Section 5.4 for timed automata. Reachability checking with a counterexample trace is useful for finding bugs, whether a error state is reached from a initial one or vice versa in case of a backwards exploration, or in a combination of both using *target enlargement*. *Counterexample guided abstraction refinement* (CEGAR) uses counterexample traces relating not to bugs in the system but in the model (due to over-abstraction) to iteratively build a model just refined enough to prove the desired property. As a dedicated error location is part of the discrete aspect of the state space, the idea of trying to explore the discrete parts faster at the expense of lagging behind in the exploration of continuous aspects could lead to quicker bug finding.

The natural representation of a state of a system model are just the values of all variables. To represent a set of states, one can thus just enumerate these states. So the most natural approach to exploring the state space of a system model is to start with some initial state, look at its possible successors, and their successors again and so on, and enumerating all the encountered states. However, enumerating all states is for all practical purposes impossible for models with continuous states, as all models which need to use a continuous state component have infinitely many states, because the continuous variable can take infinitely many values.

5.1 Finite representation of infinitely many states

To collect an infinite amount of states in a set, the states cannot be enumerated, but have to be described by constraints. For timed automata [AD94, AD90] models, the popular Uppaal tool [BDL04], among others, uses this approach for the continuous parts of a state, but keeps the explicit enumeration for the discrete parts: Clock valuations are collected into convex *clock zones* [Alu98] that describe a set of clock valuations with a set of constraints on the differences between clock values, represented by a *difference bound matrix* (DBM) [Dil89] (which, in turn, can be represented by minimal constraint graphs [LLPY97]), as described in Section 3.4. All clock valuations from the continuous states that can occur for a given discrete state are collected in a set

of DBMs, called a *federation* [LWYP98]. This is a *semi-symbolic* approach, because it represents the continuous part symbolically, but the discrete part explicitly. The model checking tool KRONOS [Yov97] also supports the use of DBMs together with explicit discrete state.

To tackle the discrete state space explosion, a symbolic approach can also be applied to sets of discrete states. In a *fully symbolic* approach [HNSY94, SB03], both the discrete and the continuous states are represented symbolically. Sets of discrete states can for example be represented by *reduced ordered binary decision diagrams* (BDDs) [Bry86, BCM+92], which are described in Chapter 4.

5.2 Separate Theories—Separate or Unified Data-Structures

Considering a timed automaton model, a state of the model is basically a pair of a discrete state encoded in a location, and a valuation of the clocks. This definition as a pair hints at the underlying structure: timed automata combine concepts from two separate theories, namely finite automata and real-valued clocks. As symbolic representations usually use constraints from the underlying theory to characterize a set, this makes it harder to find a good symbolic representation for sets of states of timed automata. Formulating constraints on one concept in the other theory requires the overhead of embedding the original theory in the other theory. Theories encompassing all the concepts are often to complex to allow for an efficient representation. Thus, the symbolic representations suitable for the discrete states are not necessarily suitable for the continuous states and vice versa. For example, using BDDs for both discrete and continuous states can lead to a blowup of the BDDs due to the magnitude of the clocks and due to interdependencies in the timing behavior [BMPY97].

Yovine's KRONOS [Yov97] also supports the use of BDDs to represent sets of discrete states, but, as Bozga, Maler, Pnueli, and Yovine point out in [BMPY97, Section 2], it is not obvious how to combine them with DBMs. Thus, they instead propose the use of *numerical decision diagrams* (NDDs) [ABK+97] at the price of discretizing the continuous domain.[1] Beyer [Bey01] also uses discretization to enable the use of BDDs for representing states of a restricted form of timed automata, so called *closed timed automata*.

It is hard to use a fixed-point reachability checking approach like Algorithm 1, which

[1] Asarin, Bozga, Kerbrat, Maler, Pnueli, and Rasse also point out in [ABK+97, Section 1] that it is not obvious how a DBM based representation for the continuous states can be combined with a symbolic representation for sets of discrete states. According to [BDM+98, Section **Supported verification techniques**], KRONOS' successor OpenKronos has the capability to employ symbolic representation of states, using DBMs and BDDs. However, the paper is unclear on whether they are used jointly, or alternatively. The latter seems likely, as the same paper [BDM+98, Section **Case studies**] lists the size of the state space as the number of pairs of a control location and a DBM. Furthermore, the latest additions to OpenKronos are listed there as well, and a successful combination of BDDs and DBMs for exact computation of the state space (as opposed to approximations as used in [DWT95]) is not mentioned.

will be described in Section 5.4, if non-canonic representations like *and-inverter graphs* (AIGs) [PSD06] or *conjunctive normal form* (CNF) clause sets are used. Seshia and Bryant [SB03] have to add special so-called *transitivity constraints* on-the-fly during the fixed point computation in order to be able to use *separation logic* formulas encoded in BDDs as a description for sets of states.

One solution to this dilemma is using separate symbolic representations for the separate concepts, each formulated in a suitable theory that allows an efficient representation. The discrete states and the continuous states can both be handled symbolically, but by different data-structures, e.g., as described in Sections 5.4.1 and 5.4.2. This was pioneered by Dill and Wong-Toi in [DWT95], where a set of control locations represented by a BDD was paired with a convex set of clock valuations, represented by a DBM. However, they judged symbolically computing the timed successor states of such pairs exactly to be too cumbersome due to non-convex sets of clock valuations in the result if invariants are taken into account, so approximation was used. This approximation approach is implemented by Yamane and Nakamura [YN04].

In my work [Ger10] and in my work together with Rüdiger Ehlers and Hans-Jörg-Peter [EGP10], a representation with those data-structures was independently developed, but with reversed order in the pairing: pairs of DBMs and BDDs were used, as presented in Chapter 6. This avoids the problem from [DWT95] of non-convex sets of clock valuations being too much hassle by hash-mapping DBMs to BDDs, allowing for rapid lookup of the DBMs in the results of the invariant applications to the results of a successor states computation. Furthermore, this mapping made the application of fully symbolic operations in the exploration algorithm very natural, as the hierarchy between time and location in the state space representation is the same as the one imposed on the operations by the necessary grouping according to the operations timing behavior. This approach has shown a lot of potential for timed game solving as well [EMP10, PEM11].

The promising combination of DBMs and BDDs lead to work developing *constraint matrix diagrams* (CMDs) [EFGP10], a data-structure unifying BDDs and DBMs more thoroughly than the simple mapping approach of the aforementioned works, but not covered in this thesis. CMDs replace the hashmap with a more memory friendly graph structure with partial DBMs that can be jointly used by several DBMs.

Møller et al. [MLAH99] propose *difference decision diagrams* (DDDs), where the nodes are labeled with clock difference constraints, encoding the boolean variables in the form of special constraints. Behrmann et al. [BLP+99] modifies DDDs into *clock difference diagrams* (CDDs) using deterministic interval based branching. Wang [Wan04], in turn, modifies CDDs into *clock restriction diagrams* (CRDs), which use overlapping upper bounds instead of intervals for branching, and are used by the model checker RED.

5.3 Algorithmic Implications

A reachability checking algorithm working with a fully symbolic state space representation can either do the discrete exploration explicitly, like the semi-symbolic approaches, as done in [Ger10], or symbolically, as done in [EGP10].

5.3.1 Operations on Explicit Data

Exploration algorithms working on explicit states are more intuitive, as computing the possible behaviors of the model in a concrete state in order to find the direct successor states is very close to answering the question how the modeled system could evolve from a given state. For backward exploration the same argument applies, as looking for the direct predecessors is also straightforward.

Aside from the easier to grasp concept, working with explicit states enables very fast low level operations on the state, e.g., when computing successor states. If a model enriched with code is used, for example Uppaal's *extended timed automata* [BDL04], the code can manipulate the discrete variables of the explicit state directly.

These advantages carry over to the explicit parts of a semi-symbolic state, i.e., an explicit discrete state paired with a set of symbolically represented continuous states. A simple way of exploring a state space using a fully symbolic state space representation is thus to decode the set on which operations should be executed, work on the enumeration of its semi-symbolic states, and encode the resulting enumerations in a fully symbolic way again as soon as no more operations need to be executed on them, an approach I chose in [Ger10].

The execution of code on a fully symbolic representation of a set of states without enumerating those states is non-trivial. Especially in those cases, it seems advisable to replace code with operations from the core of the modeling language, which, while possibly decreasing readability, makes the operations much more straightforward to compute in a fully symbolic setting.

5.3.2 Symbolic Operations on Symbolic Data-Structures

Working on a symbolic representation of sets of states can enable the use of symbolic operations on these sets. While this is usually less efficient if a single explicit state is concerned, the fact that several explicit states are concurrently treated in one operation can lead to a more efficient overall performance. Chapter 6 demonstrates the effectiveness of this approach which was also published in [EGP10]. The effectiveness of using symbolic operations is apparent if those results are compared to the performance of the approach from [Ger10], where I used a fully symbolic representation only for storage, but a semi symbolic representation to explicitly execute the operations on.

5.4 Reachability Model Checking[2]

A fully symbolic fixed point reachability checking algorithm, like the one from the work [EGP10] presented in Chapter 6, that combines two theories, in the example difference logic (using DBMs, see Section 3.4) and boolean functions (using BDDs, see Chapter 4), assumes a model with states and transitions between them, in the example a labeled transition system, namely a timed automaton as described in Chapter 3.

The transition relation will usually be a big disjunction of all the transitions, where each transition is a conjunctive formula describing the state, some conditions on the enabledness of the transition and the next state. In the example, the edges in the labeled transition system can be labeled with constraints that determine their enabledness.

Transitions that can be executed symbolically together will be grouped into one symbolic operation by a fully symbolic algorithm, e.g., grouped by timing information like guards and resets, as assumed by Dill and Wong-Toi [DWT95, Section 3.2]. In the example, the location switches guarded by the same constraint combination will be encoded as one BDD, say the constraint $a - 0 < 5 \wedge 0 - b < 3$ (or shorter $a < 5 \wedge b > 3$) would have an associated BDD that encodes all the edges that are labeled with this constraint.

Checking reachability boils down to computing the set of reachable states and checking whether some state corresponding to the reachability property is in this set. In the following, a reachability property of the form "there exists an execution of the system that eventually reaches a state where ϕ holds", in CTL: $\exists\Diamond\phi$, is assumed.[3] Here, ϕ is a boolean predicate over the state variables.

The standard fixed point algorithm for this problem starts with the set of initial states and repeatedly applies the post operator until a fixed point is reached, as shown in Algorithm 1. The post operator computes all successor states for a given set of states.

Such labeled transition systems can also label locations with constraints called *invariants*, which determine whether the system is allowed to be in the location. As it is a reachability checking algorithm, such invariants can conjunctively be added to the constraints on all incoming and outgoing edges: The system can only enter the location if the invariant is true, and only leave it while it is true. If it is given that invariants on initial states are initially true, and that invariants on time only impose upper bounds on clocks, an invariant can be applied to all states in a result from the post operator to which it applies before these states are added to the set of reachable states, which allows to remove states that contradict the invariant.

5.4.1 Discrete States

In the context of Algorithm 1, it is advisable to pre-compute a BDD representing the transition relation for use in the post operator when storing the set of reachable states

[2] This section contains parts also published in [EGP10, Section 2.3].

[3] See [BK08, Chapter 6] for a comprehensive treatment of CTL [CE82].

Algorithm 1 Least fixed point construction for computing the set of reachable states R.[4]

$R_0 := \{\text{initial states}\}$
$i := 0$
repeat
  $i := i + 1$
  $R_i := R_{i-1} \cup \mathsf{post}(R_{i-1})$
until $R_i = R_{i-1}$
$R := R_i$

as a BDD. This BDD encodes precisely the pairs of states (s, s') for which there exists a transition from s to s'.

Assuming L to be the ordered list of state variables $l_1, \ldots, l_n$, a state can be encoded as a valuation of these state variables. A set of states thus can be encoded by the BF $f(\{l_i | 1 \leq i \leq n\})$ that maps to **true** for and only for valuations of the l_i that encode a state that is contained in the set. This allows to encode a state–next-state relation using primed versions of the state variables, like the variables in the ordered list $L' = l'_1, \ldots l'_n$. Let the BF $f_{tr}(\{l_i | 1 \leq i \leq n\} \cup \{l'_i | 1 \leq i \leq n\})$ map to true if and only if for each valuation where the l_i variables encode a state s, the l'_i would encode a successor state s' of s if the primes would be removed. More formally, for a BF $f_{s'}(\{l_i | 1 \leq i \leq n\})$ that maps to **true** if and only if the valuation of the l_i encodes a state s' and a similarly defined BF $f_s(\{l_i | 1 \leq i \leq n\})$ encoding a state s, $f_{tr}(\{l_i | 1 \leq i \leq n\} \cup \{l'_i | 1 \leq i \leq n\})$ maps to **true** for exactly those valuations that satisfy $f_s(\{l_i | 1 \leq i \leq n\})$ and $f_{s'}[L/L'](\{l'_i | 1 \leq i \leq n\})$ for each pair (s, s') where s' is a successor state of s. To calculate the set of successor states of a set of states encoded by some BF $f(\{l_i | 1 \leq i \leq n\})$ the transition relation BF is applied to it, the unprimed state variables are removed using existential quantification, and the variables are finally renamed to their unprimed versions, resulting in a BF encoding all the successor states reachable in one step. More formally, the successor states are encoded in $(\exists\{l_i | 1 \leq i \leq n\}.(f(\{l_i | 1 \leq i \leq n\}) \wedge f_{tr}(\{l_i | 1 \leq i \leq n\} \cup \{l'_i | 1 \leq i \leq n\})))[L/L'](\{l_i | 1 \leq i \leq n\})$.

Baier and Katoen elaborate on this construction and its use in the post operator in [BK08, Chapter 6].

5.4.2 Continuous States

In reachability checking for timed automata, the successor state is defined not only by the successor location that can be treated by BDDs, but by the successor clock zone as well. The post operator thus also needs to find the successor clock zone if a transition is taken. Timed transitions are treated implicitly by the post operator.

First, consider the situation in the absence of invariants: From a clock zone z, a discrete transition $\delta = \langle l, a, \varphi, \lambda, l' \rangle$ (followed up by an implicit timed transition) gives

[4] This algorithm is just a pseudo-code description of a well known algorithm. This description has already been published in [EGP10, Algorithm 1] and in [Ger10, Algorithm 1].

rise to the successor clock zone $(z \wedge \varphi)[\lambda := 0]^{\Uparrow}$ through intersection with the guard φ, resetting the clocks in λ and then performing an implicit timed transition by elapsing time. The transition is only enabled if $z \wedge \varphi$ is non-empty.

Now, consider $I(l')$ to be the invariant of the successor location. The transition is only enabled if the successor clock zone before the implicit timed transition (i.e., without the time elapse) satisfies the invariant of the successor location. So, this zone then needs to be intersected with the invariant of the successor location to obtain $((z \wedge \varphi)[\lambda := 0]) \wedge I(l')$. The implicit timed transition can be incorporated as well, but is restricted by the invariant, resulting in $(((z \wedge \varphi)[\lambda := 0]) \wedge I(l'))^{\Uparrow} \wedge I(l')$.

Note that if invariants are restricted to upper bounds on clocks (that are not stricter than ≤ 0), clock resets cannot invalidate them. Furthermore, if invariants are restricted in this way, the invariant of the target location can be incorporated in the guard of each transition such that $\varphi = \varphi \wedge I(l')$ if $I(l')$ does not restrict any clock in λ, as clock resets can only affect constraints containing a clock that was reset. Moreover, for an invariant $I_\lambda(l')$ that consists exactly of those conjuncts from $I(l')$ that do not involve a clock from λ, $I_\lambda(l')$ can then be incorporated into the guard such that $\varphi = \varphi \wedge I_\lambda(l')$. Using $z \wedge \varphi = z \wedge \varphi \wedge I_\lambda(l')$ (from incorporation of the invariant in the guard) and $(z \wedge \varphi \wedge I_\lambda(l'))[\lambda := 0] = (z \wedge \varphi)[\lambda := 0] \wedge I_\lambda(l')$ (from excluding conjuncts concerning clocks from λ from the invariant) this gives $(z \wedge \varphi)[\lambda := 0] = ((z \wedge \varphi)[\lambda := 0]) \wedge I(l')$. These restrictions and changes then simplify the successor clock zone to $(z \wedge \varphi)[\lambda := 0]^{\Uparrow} \wedge I(l')$.

The restriction does not reduce expressivity: Lower bounds can be incorporated into the guards of all incoming discrete transitions to make sure only discrete transitions that do not violate the lower bound are enabled. As timed transitions can only increase the value of clocks, they cannot violate lower bounds, i.e., lower bounds cannot force the automaton out of a location due to timed transitions. Thus, reachability of a location can be restricted to clock zones that do not violate certain lower bounds without resorting to invariants. As for upper bounds, negative clock values are not allowed, so invariants restricting to < 0 effectively remove the locations they are associated with, because they can never be reached—this can be more easily achieved by simply removing those locations.

5.5 Verification and Bug Finding

Demonstrating the existence of some execution which violates a desired property or satisfies an undesired one is the domain of *bug finding*. Finding bugs in a model is very useful for refining the model, e.g., while modeling a system, and even more useful if the found bug is not in the model, but in the system itself. In model based development, bug finding can be used from a very early stage of the development process, identifying bugs quickly to allow for their removal soon after their introduction.

Verification usually demonstrates not the violation, but the satisfaction of a property. It is desirable to combine verification with bug finding such that a failure to verify a property demonstrates the existence of a bug by providing a counterexample to the

desired property, which is, in turn, an example for the found bug.

Interesting positive properties that some system is desired to fulfill are usually universal, i.e., something is supposed to happen eventually (liveness), or something is supposed to never happen (safety). When verifying some universal property, all possible executions of a system model have to be considered in order to prove that the model satisfies the property. On the other hand, finding one execution of the system model that violates the property is enough to demonstrate that a model does not satisfy the property. Existential properties, i.e., that something could possibly happen, are mostly properties that systems are desired to not fulfill. Here, the converse applies: disproving an existential property requires all executions of a system model to be considered, while proving it can be achieved by finding one satisfying execution. Note that this work is mostly concerned with safety properties.

5.5.1 Exploration

In reachability checking, the model usually contains dedicated error locations which are entered when the property is violated.

Fully exploring the reachable state space allows to verify whether an error location is reachable from an initial state.

Forward or Backward

There are basically two options to explore the state space: (1) *Forward exploration* starts from the initial states and looks for all states reachable from there by looking at all possible successor states of states found to be reachable. (2) *Backward exploration* starts from the error states and looks for all states that can reach them by looking at all predecessor states of states from which the error is known to be reachable. This can be combined, e.g., using *target enlargement*, where some set of states from which an error state is reachable is computed and all those states are then marked as error states. This set is usually computed cheaply using backward exploration, e.g., with underapproximation, and only contains a few of the states from which an error is reachable. Then, the forward exploration from the initial states is done in the usual way, but on a model containing an enlarged set of error states.

5.5.2 Counterexample-Guided Abstraction Refinement

To verify models that are very complex, approaches based on automatic abstraction aim at reducing the complexity of the model to check. In *counterexample-guided abstraction refinement* (CEGAR) [CGJ+00], the initially very simple abstraction gives the abstract model more behaviors than the actual model has. Model checking the abstract model provides either proof that the model does not have the unwanted behavior, or delivers a counterexample demonstrating the unwanted behavior in the abstract model. If the counterexample can be concretized in the actual model, an actual counterexample is found. Otherwise, the counterexample is *spurious*, i.e., it demonstrates a behavior that

was added to the model due to the abstraction. This counterexample is then used to automatically refine the abstraction, removing this offending behavior.

CEGAR usually requires many iterations of the refinement loop each producing a slightly less coarse abstraction than the previous one to find an abstraction that is sufficiently fine to prove the property. Essentially, each of these iterations starts with a bug finding step, thus quick discovery of reachable error states to provide a counterexample is desired from a model checking technique used in the refinement loop.

5.6 Data-Structure Tradeoffs

The choice of the data-structure to be used by an algorithm for verification or bug finding needs to take into account several aspects. For example, the algorithm may be most suited to a specific data-structure, as the development of an algorithm is often tightly integrated with the development of a data-structure that is proposed to be used by the algorithm. In a state space exploration approach, data-structures are needed to store two sets of states: the states that have already been explored, and the states whose successors need to be explored. It is possible to use different data-structures for these two sets, as done in [Ger10], but this entails encoding the results of looking for the successors of states in the one set into a different data-structure for storage in the other set.

5.6.1 Memory Consumption vs Runtime

These two roles of the used data-structures, storing the set of states whose successors need to be explored and storing the set of the reachable states, give rise to a trade-off: A data-structure optimal for a quick exploration of the successor states might for example use an explicit or semi-symbolic representation for a set of states, which will consume a lot of memory. A data-structure optimal for storing may states using only little memory might for example use a fully symbolic representation for a set of states, which will only allow for a slower successor computation routine.

5.6.2 Symbolic Operations on Symbolic Data-Structures

That a symbolic representation is slower when computing a single successor state does not mean it needs to be slower when computing several successors, if symbolic operations can be used.

Finding all successors of all states from a set of states can be achieved in a single symbolic operation, as will be demonstrated in Section 6.1 on page 56. Even if not all immediate successors can be found in a single symbolic step, at least all immediate successors which result from a set of transitions that can be executed together symbolically can be computed in a single symbolic step, e.g., as described in Section 6.1.1. If enough states can thus be treated by a single symbolic operation, a few slow symbolic successor computations needed to explore a reachable state space can still be faster

than many more explicit successor computations that, while very fast individually, treat only a single explicit or semi-symbolic state. So over the whole runtime of an algorithm, symbolic operations on symbolic data-structures can be faster as well as less memory intensive than an explicit or semi-symbolic approach, as demonstrated in Table 6.2 on page 65.

5.6.3 Preferring some Theory over the Others

It can improve the efficiency of a state space exploration if certain parts of the state space are explored preferably, as will be described in Section 6.1.4 on page 60. Furthermore, in the context of bug finding, it might be advantageous to prefer exploration steps in the theory most likely to quickly lead to a bug, e.g., the theory encoding the locations.

The set of states whose successors are to be explored has subsets of states whose successors can be calculated in one symbolic operation. As described in Section 6.1.4, it can speed up the exploration of the whole state space if these subsets grow fast. The bigger these subsets become, the more states can be treated in one symbolic operation. If the successor computation can be done in one symbolical step for all states that fulfill certain constraints from one theory, it can thus be advantageous to prioritize exploration in the other theory: Finding more states that are separated from the already discovered states in this other theory alone grows the subsets, while discovering states separated from the already discovered states in the first theory creates new subsets.

Prioritizing a Data-Structure

In Chapter 7, I propose to prefer exploration in one theory over the other in the exploration algorithm. As the data-structure used is the combination of two data-structures (BDDs and DBMs) for the two theories (boolean functions and difference logic) that treat the two aspects of a state (location and continuous time), the preference for a theory can be achieved by preferring exploration in one data-structure over the other.

Chapter 6

Making the Right Cut in Model Checking Data-Intensive Timed Systems[1]

Contents

[1]This chapter constitutes a modified version of work already published in [EGP10].

Abstract. The success of industrial-scale model checkers such as UPPAAL [BDL04] or NUSMV [CCGR00] relies on the efficiency of their respective symbolic state space representations. While difference bound matrices (DBMs) are effective for representing sets of clock values, binary decision diagrams (BDDs) can efficiently represent huge discrete state sets. In this chapter, a simple general framework for combining both data-structures is introduced, enabling a joint symbolic representation of the timed state sets in the reachability fixed point construction. Especially in the analysis of models with only few clocks, large constants, and a huge discrete state space (such as, e.g., data-intensive communication protocols), the presented technique turns out to be highly effective. Additionally, the presented framework allows to employ existing highly-optimized implementations for DBMs and BDDs without modifications. The approach is evaluated and compared with UPPAAL [BDL04] and RED [Wan04] using a prototype implementation on two benchmarks: A model of the FlexRay physical layer protocol and a model of Fischer's mutual exclusion protocol.

6.1 Fully Symbolic Real-Time Model Checking

This section presents the timed state set representation and the extension of the basic fixed point algorithm for computing the set of reachable states in order to make it applicable to this representation. The representation is described in a general way and abstracted from the actual choices of data-structures for representing clock zones (CZ) and discrete location sets (LS), which allows to consider the approach seperately from the details of implementation choices. The actual implementation of the approach (as described in Section 6.3), uses DBMs and BDDs. However, the general idea is applicable to all suitable data-structure types (such as, e.g., CDDs [BLP+99]). Thus, future alternatives for storing clock zones and location sets can also be used with the presented approach.

To simplify the presentation, the case of invariant-free timed automata is presented first and the application of the algorithm on an example network is demonstrated. Afterwards, the idea is extended to include support for invariants. Possible optimizations to the algorithm are discussed at the end.

Sets of CZ/LS pairs permit representing the timed and discrete parts of sets of states separately, and provide the basis for the approach. The sets of states $\mathcal{S}$ in the fixed point computations are defined as partial functions $\mathcal{S} : \mathcal{Z} \rightharpoonup 2^L$ from the set of clock zones $\mathcal{Z}$ to the powerset of the set of locations L. For such a so-called *clock zone map* (CZM), a state $(l, \vec{t})$ is in $\mathcal{S}$ if for some $z \in \mathcal{Z}$, $\vec{t} \in z$ and $l \in \mathcal{S}(z)$. Note that the choice of z is not required to be unique.

6.1.1 Computing the Reachable States using CZMs

In the following, the adaptation of the basic fixed point construction for computing the set of reachable states given in Algorithm 1 on page 49 to work with CZMs is described. The first step is to partition the overall transition relation of the system: for each combination of clock guards and resets that occurs along some transition, a separate transition relation $T[\varphi, \lambda]$ containing all transitions corresponding to the guard/reset

Algorithm 2 Computing the set of reachable states R represented as a CZM. The post operator is parametrized by the transition relation used.[2]

1: **for all** guard/reset pairs (φ, λ) in the system **do**
2: **compute the transition relation** $T[\varphi, \lambda]$
3: **end for**
4: $R := \{z_0^{\Uparrow} \mapsto \{l_0\}\}$
5: $W := \{z_0^{\Uparrow}\}$
6: $R' := R$
7: **repeat**
8: $R := R'$
9: $W' := \emptyset$
10: **for all** $z \in W$ **do**
11: **for all** guard/reset pairs (φ, λ) **do**
12: $L := \mathsf{post}_{T[\varphi,\lambda]}(R[z])$
13: $z' := (z \wedge \varphi)[\lambda := 0]^{\Uparrow}$
14: **if** $R'[z'] \not\supseteq L$ **then**
15: $R'[z'] := R'[z'] \cup L$
16: $W' := W' \cup \{z'\}$
17: **end if**
18: **end for**
19: **end for**
20: $W := W'$
21: **until** $R = R'$

pair (φ, λ) is built. This separates timing concerns from the discrete transitions of the system and makes it easy to compute successor clock zones from a given source clock zone and some guard/reset pair. When BDDs are used for representing location sets, it is not necessary to enumerate the product locations in the global timed automaton explicitly if the system is given as a network of timed automata, as the synchronization between the components can be encoded symbolically.

When the transition relations are built, the standard fixed point computation is performed. However, it iterates over all such guard/reset pairs in every step, in order to apply the pairs' associated transition relation. After each discrete transition, the set of possible timed transitions that can follow is computed as well, in order to obtain the successor clock zone. Algorithm 2 shows the details of these steps.

The pre-fixed point is stored in R, where $R[z]$ represents the locations associated with clock zone z in R. The algorithm iterates over this set of reachable states, in each new round looking at the pre-fixed point from the previous round. For every clock zone in the domain of R, it computes successor locations L and clock zones z' for each guard/reset pair (φ, λ) in the transition relation. If the new CZ/LS pair (z', L)

[2]This algorithm has already been published in [EGP10, Algorithm 2]. A pseudo-code version of the underlying approach has already been published in [Ger10, Algorithm 2].

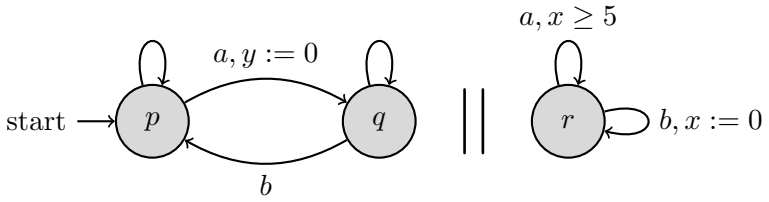


Figure 6.1: An example network of timed automata.[3]

is not already contained in the pre-fixed point, it is added to the next pre-fixed point R'. To make the computation more efficient, the changes in the CZM are tracked in the waiting set W. This allows to avoid re-considering CZ/LS pairs which have not changed since the previous round of the algorithm. The added inner loop, in which all guard/reset pairs are iterated over, is of course additional work as compared to semi-symbolic approaches. But unlike model checkers keeping the discrete part of the system explicit, this approach can compute timed successor zones for many locations at the same time.

The presented algorithm's correctness is guaranteed by the fact that it is essentially a classical reachability fixed point algorithm. For every number of steps $n \in \mathbb{N}$ and timed state $(l, \vec{t})$ that is reachable from the initial state in n discrete steps, the pre-fixed point R contains this state after at most n iterations of computing these pre-fixed points. The termination of this algorithm is obvious, as clock regions are never split and the number of sets of these is finite, as is the number of discrete locations, and the algorithm terminates when no new clock regions or discrete locations are found.

6.1.2 An Example

Consider the parallel composition of the timed automata depicted in Figure 6.1. The product automaton has two locations pr and qr, and two clocks x and y. There are three guard/reset pairs $(\mathbf{true}, \emptyset)$, $(x \geq 5, \{y\})$, and $(\mathbf{true}, \{x\})$. Each of them is associated with one of the following three transition relations:

- $T[\mathbf{true}, \emptyset] = \{(qr, qr), (pr, pr)\}$,
- $T[x \geq 5, \{y\}] = \{(pr, qr)\}$, and
- $T[\mathbf{true}, \{x\}] = \{(qr, pr)\}$.

When the algorithm is run on this example, $\big\{(x = y = 0)^{\Uparrow} \mapsto \{pr\}\big\} = \big\{(x = y) \mapsto \{pr\}\big\}$ is the initial value for R:

- $R = \big\{(x = y) \mapsto \{pr\}\big\}$ and
- $W = \{(x = y)\}$.

[3]This figure has already been published in [EGP10, Fig. 1].

Then, in the fixed point computation, it iterates over the transition relations and obtains

- $R = \big\{\{(x = y) \mapsto \{pr\}, (x \geq 5 \wedge y \leq x - 5) \mapsto \{qr\}\big\}$ and
- $W = \{(x \geq 5 \wedge y \leq x - 5)\}$.

as the transitions synchronizing on a are enabled when $x \geq 5$, resetting y to 0. In the second round, it adds $(x \leq y) \mapsto \{pr\}$ to R obtaining

- $R = \big\{(x = y) \mapsto \{pr\}, (x \geq 5 \wedge y \leq x - 5) \mapsto \{qr\}, (x \leq y) \mapsto \{pr\}\big\}$ and
- $W = \{(x \leq y)\}$.

as from states with the location qr, the transitions synchronizing on b are enabled, resetting x to 0. The algorithm then terminates in the following round, as no new states are discovered, providing $R = \big\{\{(x = y) \mapsto \{pr\}, (x \geq 5 \wedge y \leq x - 5) \mapsto \{qr\}, (x \leq y) \mapsto \{pr\}\big\}$ as the CZM representation of the set of reachable states in the system.

6.1.3 Adding support for invariants

Algorithm 2 is not able to deal with invariants. The invariants active at some point in the run of a system depend on the system's current location. To add the handling of invariants to the fixed point construction, it is necessary to take into account the location data from the discrete part of the state when computing the successor clock zones. In the FlexRay physical layer protocol models, the number of clocks and the number of distinct invariants is small. Hence, it is feasible to enumerate all possible invariants of the product timed automaton as a pre-computational step.

Let $\mathcal{I}$ be the set of all invariants appearing in the product automaton of a timed system. Assume that each invariant is given in minimal form. Consider the example where precisely the invariants $x \leq 3$, $x \leq 4$, and $y \leq 2$ are associated to some locations of three different components of the input network. In this example, the set of all invariants of the product automaton is $\mathcal{I} = \{\mathbf{true}, x \leq 3, x \leq 4, y \leq 2, x \leq 3 \wedge y \leq 2, x \leq 4 \wedge y \leq 2\}$. Now consider the function $C : \mathcal{I} \to 2^L$ that maps each invariant onto the set of locations in which precisely the given invariant must hold. One can easily compute $\mathcal{I}$ and C in a preprocessing step that does not need to construct the product automaton.

In the fixed point construction, the computed successor locations need to be split according to their associated invariants. This is achieved by splitting them according to the locations mapped to in C. Thus, the successor clock zones can be computed taking the respective invariants into account. If the initial location has an active invariant, this invariant also needs to be taken into account when computing the initial zone. Algorithm 3 is a version of Algorithm 2 with the necessary changes to handle invariants.

Algorithm 3 Algorithm 2 with added invariant treatment.[4]

```
 1: for all guard/reset pairs (φ, λ) in the system do
 2:   compute the transition relation T[φ, λ]
 3: end for
 4: R := {z_0^⇑ ∧ I(l_0) ↦ {l_0}}
 5: W := {z_0^⇑ ∧ I(l_0)}
 6: R' := R
 7: repeat
 8:   R := R'
 9:   W' := ∅
10:   for all z ∈ W do
11:     for all guard/reset pairs (φ, λ) do
12:       L := post_{T[φ,λ]}(R[z])
13:       for all i ∈ I do
14:         L' := L ∩ C(i)
15:         z' := ((z ∧ φ)[λ := 0] ∧ i)^⇑ ∧ i
16:         if R'[z'] ⊉ L' then
17:           R'[z'] := R'[z'] ∪ L'
18:           W' := W' ∪ {z'}
19:         end if
20:       end for
21:     end for
22:   end for
23:   W := W'
24: until R = R'
```

6.1.4 Possible Optimizations

The performance of the presented technique could be improved by deviating from the strict rule of computing one pre-fixed point after the other. All newly encountered CZ/LS pairs (z, l) can be directly stored into the pre-fixed point R in the algorithm instead of storing them in some intermediate set R', which is later copied to R when all elements from the waiting set have been processed. This saves computation time in the case that such a z is in the waiting set W but not yet processed in the respective round of the fixed point computation. In this case, by storing the newly reachable locations for z into R, they are also taken into account when z is finally drawn from the waiting list in the respective round of the algorithm and their exploration is not delayed until the next round, resulting in a lower number of steps in total until the fixed point is reached.

Note that this also allows to use a waiting queue instead of a set. Then, in line 10 of Algorithm 2 (and Algorithm 3), the zones are *popped*, so every zone from W that was picked is removed. This opens the door for optimization strategies based on the ordering

[4] This algorithm is based on [EGP10, Algorithm 3] and [EGP10, Algorithm 2].

of the queue. For example, the waiting queue can be modified such that clock zones are drawn from it prioritized by their first appearance in R. This way, the exploration of new clock zones is delayed such that the progress in computing the reachable discrete states for zones encountered earlier can be forwarded to successor zones more efficiently. This optimization idea inspired the approach presented in Chapter 7.

6.2 Guided Counterexample Generation

Algorithms 3 and 2 are only capable of computing the set of reachable states. As checking if it contains some given goal state is trivial after it has been computed, this suffices to make the approach presented suitable for a typical verification task for timed systems: checking that no error state is reachable.

For cases in which some error state is reachable, however, obtaining a *counterexample*, i.e., a sequence of transitions from some initial state to some error state, is desired to help the designer of a timed system to improve the model or even to document a genuine flaw in the system. Therefore, most modern model checking tools can generate counterexamples.

For Algorithm 2, finding a counterexample relies on storing the pre-fixed points. When an error state is found in the pre-fixed point R_n from step n, a predecessor state of the error state can be picked from R_{n-1} by following all transitions entering the error state backward and picking one that leaves a state that is in R_{n-1}. Such a transition can always be found as at least one such transition needed to be followed in order to discover the error state when the successors of R_{n-1} were computed. For this predecessor state of the error state, a predecessor state from R_{n-2} can be picked an so on, until finally an initial state is reached. The reversed sequence of all the chosen transitions then forms a trace leading from an intitial state to the error state, i.e., a counterexample.

This procedure can also be used to find a counterexample if the error state is discovered with Algorithm 3, but only with some additional meassures.

Suppose that Algorithm 3 terminates early with a set of forward reachable states R comprising some error states E. Then, a nonimproved fixed point construction is executed to compute a sequence of pre-fixed points $F_0, \ldots, F_n$, where each F_i is restricted to R and $E_f = F_n \cap E \neq \emptyset$. Note that for all $0 \leq i \leq n$, the set F_i contains only states that are reachable after exactly i steps.

This allows to compute a counterexample using a *backward nonimproved* fixed point construction producing a sequence of backward reachable sets of states $B_n, \ldots, B_0$ with $B_n = E_f$. For each $0 < j \leq n$, B_{j-1} is computed by picking one particular semi-symbolic state (i.e., a zone and one concrete location) from B_j, compute its predecessors, and restrict them to F_{j-1}. After each iteration j, a transition connecting a state in B_{j-1} and B_j is picked, and added to the counterexample. The computation of the predecessors can be done in a symbolic way similar to the techniques described in this chapter for computing the sucessors.

6.3 Experimental Results

6.3.1 Prototype Implementation

The approach presented in this chapter has been implemented by Hans-Jörg Peter and Rüdiger Ehlers in a prototype model checker using the UPPAAL-DBM library [Ben02] for representing DBMs and the CUDD library [Som09] for representing BDDs. To allow a fair comparison with UPPAAL [BDL04] and RED [Wan04], the prototype tool reads automata-based specifications as input. The first step in its execution is to call the tool NOVA from the SIS toolset [SSL+92] as a back-end for finding efficient assignments of control locations to BDD variable valuations. Then, the guard/reset pairs of the given timed system are collected and, for each pair, the BDD representing the symbolic transition relation over the discrete control structure is computed (using the assignments obtained in the first step). In the last step of the preparation phase, the possible invariant combinations are collected and, for each combination, the BDD representing the associated locations is computed.

The actual fixed point computation of the reachable states is implemented as described in Section 6.1. For the state space representation, a hash map that maps DBMs to BDDs is used. No fixed BDD variable ordering is provided a priori nor any other insight into the model is used to optimize the BDD representation. Instead, the implementation only relies on the automatic on-the-fly reordering heuristics implemented in the CUDD library.

6.3.2 A FlexRay Physical Layer Protocol Model

I created a benchmark that investigates the physical layer protocol of the FlexRay protocol as introduced in Section 2.1 starting on page 12, where a message is transmitted during a so-called *static segment* from a sending ECU to a receiving ECU. As a crucial correctness property, it is required that there is no deviation of the message received from the message sent. The FlexRay physical layer protocol as described in Section 2.2 starting on page 17 is modeled in the benchmark, the important details of the model variant used in this chapter are described in the following.

Clocks. Since the receiving and sending ECU are running asynchronously, two clocks are used to model the timing behavior. The length of a clock cycle may deviate by at most 0.15 % from the standard rate.

Bit Stream Format. The actual payload of a transmission between two ECUs has a maximal length of 262 bytes. It is embedded into a structured bit stream that consists of (1) the initial 15 bit *transmission start sequence* (TSS), (2) the 1 bit *frame start sequence* (FSS), (3) the individual bytes of the payload, each prepended with a 2 bit *byte start sequence* (BSS), and finally (4) the 2 bit *frame end sequence* (FES). Thus, the maximal bit stream length is 2638 bits. Note that modeling the length of the message

increases the discrete complexity of the benchmark, thus enabling a smooth scaling of said discrete complexity by scaling the number of bytes in the payload of the message.

Error Model. As a reasonable error model, in any sequence of 5 consecutive bits on the bus, 1 bit might be flipped. However, in this chapter, instantaneous transitions on the bus[5] are assumed and registers are assumed to sample values correctly ignoring *setup* and *hold* times (both assumptions reduce timing complexity), thus these bit flips do not only have to account for glitches, but also for jitter. Therefore, the resulting benchmark model does not yield results on tolerable glitch patterns, as it cannot distinguish between the effects of glitches and those of jitter (see Section 2.3 for details on tolerable glitch patterns).

6.3.3 Model Checking the FlexRay Model

I modeled the physical layer protocol of the FlexRay protocol [Fle05] as a network of timed automata[6], as described in Section 6.3.2, for usage with UPPAAL[7] [BDL04], RED[8][Wan04], and the prototype model checker. As a safety property, the reachability of a dedicated error location, which the receiver enters upon an uncompensatable deviation of the received from the sent bit stream, is checked. Table 6.2 on page 65 shows the results of this evaluation. Unfortunately, for every payload length, RED runs out of memory (e.g., for the smallest instance it hits the 4 GB limit after 18 minutes).

The most striking observation is that the prototype overall needs much less memory than UPPAAL or RED, which allows to verify the full payload length of 262 bytes. In fact, while the prototype model checker's memory consumption always stays below 1 GB, UPPAAL's memory and time consumption increases linearly in the length of the payload, resulting in running out of memory with a payload length of 34 bytes or more. It is also noteworthy that in most of the cases the presented approach also outperforms UPPAAL w.r.t. the running time. An oscillation effect can be observed in the running times and space consumptions of the implementation which is caused by the variable reordering and caching heuristics of the CUDD library. This BDD-related phenomenon is also observable in other contexts (see, e.g., [BGJ+07]). Nevertheless, the number of symbolic exploration steps increases linearly in the length of the payload and the set of encountered clock zones reaches its fixed point at a payload length of 24.

6.3.4 Model Checking the Fischer Protocol

In addition to the FlexRay case study from Section 6.3.3, the Fischer mutual exclusion protocol is also considered, which is a standard benchmark from the timed model

[5] This means in Figure 2.5 on page 19, the transition between high and low on the bus would be vertical instead of sloped.

[6] The models are available at `http://www.avacs.org/Benchmarks/Open/flexray.tgz`

[7] Version 4.0.11, running with aggressive space optimization – option -S2

[8] Version 8.100511

checking domain with a small discrete state space and one clock per component. Table 6.1 shows that the existing model checking techniques implemented in Uppaal and RED perform better than the prototype on this benchmark.

This is, however, not surprising, as the Fischer protocol does not fall into the class of systems whose verification the approach aims at. This chapter presented a specialized technique for timed systems with a large discrete state space but only a few clocks, an important class of models that comprise, e.g., data-intensive asynchronous communication protocols. The Fischer protocol model, on the other hand, has a large number of clocks (one per component), but only few locations, thus the standard semi-symbolic state space representation used in Uppaal is already quite effective here. Also, RED's symmetry reduction is beneficial for this particular protocol.

	CZM model checker				Uppaal		RED		
Benchmark	Steps	Zones	Time	Memory	Time	Memory	Steps	Time	Mem
Fischer 3	26	19	0 s	81 MB	1 s	0	5	0 s	29 MB
Fischer 4	258	149	0 s	81 MB	1 s	0	3	0 s	21 MB
Fischer 5	3156	1496	1 s	108 MB	0 s	37 MB	5	0 s	21 MB
Fischer 6	42528	17426	32 s	156 MB	0 s	37 MB	5	1 s	45 MB
Fischer 7	612531	227522	17 min	302 MB	1 s	37 MB	5	1 s	66 MB
Fischer 8	TIMEOUT				3 s	38 MB	5	3 s	105 MB
Fischer 9	TIMEOUT				15 s	42 MB	5	8 s	174 MB
Fischer 10	TIMEOUT				65 s	56 MB	5	19 s	303 MB

Table 6.1: Comparison of the prototype with Uppaal and RED on the (timing-intensive) Fischer protocol benchmark.[9]

6.4 Conclusion

DBMs and BDDs impressively demonstrate their effectiveness in model checkers such as Uppaal and NuSMV. However, since NuSMV can only handle pure discrete models and Uppaal does not have a symbolic representation for the discrete part of the state space, both tools fail in verifying timed systems with large discrete control structures.

This chapter presented a fully symbolic approach to timed model checking which combines DBMs with BDDs as presented in [EGP10]. In contrast to other approaches, the presented technique neither suffers from a loss of modeling precision (it remains in the classical timed automata framework) nor leads to blow-ups in the BDDs (it avoids the encoding of timing interdependencies in the BDDs).

The presented approach was further improved in work presented in [EFGP10] by transcending CZMs' simple mapping relationship between DBMs and BDDs to create *Constraint Matrix Diagrams* (CMDs), a fully integrated graph based fully symbolic timed state space representation.

[9] Parts of this table have already been published in [EGP10, Table 2].

		CZM model checker				Uppaal	
Payload	Correct	Steps	Zones	Time	Memory	Time	Memory
1	Yes	6566	1858	86 s	252 MB	23 s	88 MB
2	Yes	8606	2498	2 min	251 MB	69 s	205 MB
3	Yes	10423	3142	7 min	527 MB	2 min	325 MB
4	Yes	12143	3782	2 min	251 MB	3 min	436 MB
5	Yes	13863	4422	4 min	312 MB	4 min	563 MB
6	Yes	15583	5062	5 min	415 MB	5 min	675 MB
7	Yes	18647	5706	4 min	312 MB	6 min	786 MB
8	Yes	20367	6346	6 min	311 MB	7 min	930 MB
9	Yes	22087	6986	5 min	259 MB	8 min	1 GB
10	Yes	23807	7626	6 min	311 MB	8 min	1 GB
11	Yes	26872	8270	5 min	262 MB	9 min	1 GB
12	Yes	28592	8910	12 min	526 MB	10 min	1 GB
13	Yes	30312	9550	13 min	528 MB	11 min	1 GB
14	Yes	32032	10190	14 min	527 MB	12 min	2 GB
15	Yes	35094	10834	8 min	266 MB	13 min	2 GB
16	Yes	36814	11474	5 min	259 MB	14 min	2 GB
17	Yes	38534	12114	5 min	256 MB	15 min	2 GB
18	Yes	40254	12754	5 min	260 MB	15 min	2 GB
19	Yes	43314	13398	11 min	405 MB	16 min	2 GB
20	Yes	45029	14038	5 min	261 MB	18 min	2 GB
21	Yes	46750	14678	6 min	259 MB	18 min	2 GB
22	Yes	48470	15318	5 min	262 MB	19 min	2 GB
23	Yes	50190	15958	4 min	259 MB	20 min	3 GB
24	Yes	51991	16024	7 min	264 MB	22 min	3 GB
25	Yes	52629	16024	5 min	314 MB	22 min	3 GB
26	Yes	52909	16024	5 min	265 MB	24 min	3 GB
27	Yes	53189	16024	35 min	522 MB	26 min	3 GB
28	Yes	53469	16024	7 min	266 MB	25 min	3 GB
29	Yes	53749	16024	6 min	262 MB	25 min	3 GB
30	Yes	54029	16024	6 min	265 MB	28 min	3 GB
31	Yes	54309	16024	16 min	415 MB	29 min	4 GB
32	Yes	54589	16024	6 min	313 MB	31 min	4 GB
33	Yes	54869	16024	28 min	955 MB	31 min	4 GB
34	Yes	55149	16024	13 min	313 MB	MEMOUT	
60	Yes	66230	16024	18 min	520 MB	MEMOUT	
100	Yes	90230	16024	57 min	941 MB	MEMOUT	
150	Yes	120230	16024	30 min	406 MB	MEMOUT	
200	Yes	150230	16024	72 min	938 MB	MEMOUT	
262	Yes	187430	16024	28 min	413 MB	MEMOUT	

Table 6.2: Comparison of the prototype with Uppaal on the FlexRay physical layer protocol case study. The first column shows the length of the payload in bytes. The second column states the obtained verification result. The next four columns show the number of symbolic steps (i.e., applications of the post operator) until the reachability fixed point is reached, the number of distinct clock zones encountered, the running time, and the memory consumption of the prototype model checker. The last two columns show the running time and space consumption of Uppaal. All benchmarks were executed on an AMD Opteron processor with 2.6 GHz and 4 GB RAM.[10]

[10]Parts of this table have already been published in [EGP10, Table 1].

Chapter 7

Underapproximating Lookahead in Symbolic Forward Reachability-Checking

Contents

Abstract. This chapter explores the combination of *Difference Bound Matrices* (DBMs) and *Reduced Ordered Binary Decision Diagrams* (BDDs) during the fully symbolic state space exploration. It proposes a novel algorithm to find more successor states in one symbolic step, at the price of a stronger fragmentation of the representation of the explored state space: If a set of transitions is executed in a symbolic step, this set can be enlarged by adding transitions with not the exact same guard/reset set, but with guards that are implied and side effects that are hidden. This associates more locations with the resulting clock zone, which is smaller than it could be for the additional locations, thus allowing a look at an underapproximation of the zone ahead of the regular discovery of the full clock zone.

7.1 A bitstring/difference-logic model

Models with a real-time aspect, like FlexRay, come with a discrete part encodable as a bitstring, and a continuous part encodable in difference-logic: Values of variables

and the active location can easily and efficiently be encoded in binary. Convex sets of valuations of clocks can be encoded in a conjunction of difference constraints.

7.1.1 Symbolic Data-Structures: DBMs and BDDs

A symbolic representation of bitstrings is possible with *Reduced Ordered Binary Decision Diagrams* (BDDs) [Bry86, BCM+92] (see Chapter 4). Convex sets of clock valuations can be handled symbolically with *Difference Bound Matrices* (DBMs) [Dil89] (see Section 3.4). The idea proposed in this chapter will be presented using *Clock Zone Maps* (CZMs), a mapping from DBMs to BDDs to represent the state space of a reachability checking algorithm working on a network of timed automata, as introduced in the work [EGP10] presented in Chapter 6.

7.2 Growing the BDDs faster

If data, e.g., DBMs, is mapped to BDDs, an optimization from [EGP10, Section 3.4] presented in Section 6.1.4 is based on the principle of growing the BDDs fast before looking at the successors resulting from the data mapping to this BDD. This allows to transfer more discrete information stored in the thus bigger BDD to the possible successors found when looking at the data mapping to the BDD, instead of computing this discrete information again and again for each successor of the original data entry mapping to the BDD.

This principle can be used not just as an optimization, but as the foundation of a reachability checking algorithm for timed automata.

7.3 Algorithm Idea

In a symbolic reachability checking algorithm such as described in Section 5.4 and (more detailed) in Chapter 6 and [EGP10], different theories are combined in each step of the state space exploration. Such an algorithm can prioritize one of these theories over the other. That implies always looking for successors in the prioritized theory, and only looking at the other theory when no further progress can be made otherwise. As an example, if the location switches guarded by the same constraint combination are encoded as one BDD—say the constraint $a - 0 < 5 \wedge 0 - b < 3$ (or shorter $a < 5 \wedge b > 3$) has an associated BDD that encodes all the edges that are labeled with this constraint—the successor function using the BDD associated with $a < 5 \wedge b > 3$ would be repeatedly applied until its fixed point is reached.

If the non-prioritized theory has certain features, this principle can be executed in a stronger way. Assume the non-prioritized theory, difference logic in the example, has constraints that induce an implication-semi-order. In difference logic, order constraints talking about the same variables have an implication-semi-order: For example, $a - b < 5 \implies a - b < 7 \implies a - b < 42$ and so on. This implication-semi-order can be exploited to add implicitly enabled transitions to a symbolic operation that

executes grouped transitions. In the example, the edges associated with the constraint $a < 7 \wedge b > 2$ could be added to those associated with $a < 5 \wedge b > 3$ as $(a < 5 \implies a < 7) \wedge (b > 3 \implies b > 2)$, thus including more enabled transitions in the same symbolic operation.

7.4 CZM Algorithm

In the following, the algorithm idea from Section 7.3 will be explored by presenting a CZM based symbolic forward reachability checking algorithm.

Symbolic Forward Reachability-Checking. A symbolic fixed point algorithm for forward reachability checking of timed automata was presented in the work [EGP10] described in Chapter 6. A mapping from clock zones, stored in DBMs, to the discrete state sets, encoded in BDDs, represents the pre-fixed point. That algorithm uses an encoding of the transition relation in BDDs. For each combination of guard and reset-set that shows up on some transition, a BDD encoding all these transitions is precomputed. During computation of the next pre-fixed point, each of these BDDs is iteratively applied to each of the stored DBMs' associated BDDs. The locations associated with a specific invariant are also encoded in a BDD, and these are then iteratively applied to the result of the application of the transition BDD. This allows to compute the successor clock zone for all the successors computed using these BDDs, as the successor clock zone depends only on the clock zone, the guard, the reset sets, and the invariants of the target location.

7.4.1 Formalization of Algorithm Idea

As explained in Section 7.3, constraints can be semi-ordered.

For a given guard reset-set pair $\langle \varphi, \lambda \rangle$ where the intersection $z \wedge \varphi$ of a clock zone z with the guard φ is non-empty, this semi-order allows to find which guards ϕ implicitly also give rise to non-empty intersections with z if φ does. Looking at all the guard reset-set pairs with the same reset set λ, all transitions with a guard reset-set pair $\langle \phi, \lambda \rangle$ where $\varphi \implies \phi$ are enabled by clock zone z with $z \wedge \varphi$ not being empty. If all transitions with such guard reset-set pairs $\langle \phi, \lambda \rangle$ where $\varphi \implies \phi$ are added to the transition BDD associated with the guard reset-set pair $\langle \varphi, \lambda \rangle$, all locations that are reached additionally are actually reachable. However, before invariants are applied, they will be associated with the successor clock zone $(z \wedge \varphi)[\lambda := 0]^{\Uparrow}$. This clock zone can be smaller than the clock zone $(z \wedge \phi)[\lambda := 0]^{\Uparrow}$ that they would be associated with in the original algorithm, but, as $\varphi \implies \phi$, $z \wedge \varphi \implies z \wedge \phi$. Thus, they are associated with a clock zone representing valuations that are reachable in this location (in the absence of invariants), it may just be that some valuations that are reachable are not represented.[1] Hence, the reachable states are under-approximated in this step.

[1] This can lead to redundant clock zones later on, and it can cause harmful fragmentation, see Section 7.4.1 *Fragmentation*.

This can easily be fixed later by applying the transition BDD associated with $\langle \phi, \lambda \rangle$ to the BDD representing the locations reachable with clock zone z. Through encoding more transitions into the transition BDD for $\langle \varphi, \lambda \rangle$, more locations can be recognized as being reachable early, allowing to grow the BDDs encoding the reachable locations faster.

Moreover, when invariants are replaced by constraints incorporated into guards,[2] the successor clock zone computation does only rely on the clock zone, the guard and the reset-set of the transition. Thus, the effects of guards and resets can, to a certain extend, be precomputed: If a transition BDD associated with the guard reset-set pair $\langle \varphi, \lambda_1 \rangle$ has been applied to the locations associated with clock zone z, the newly reachable locations are associated with the successor clock zone $z' = (z \wedge \varphi)[\lambda_1 := 0]^{\Uparrow}$. These transitions were enabled, i.e., $z \wedge \varphi$ is not empty. Let $\varphi[\lambda_1 := 0]$ denote the conjunction of all constraints from φ that do not refer to a clock in λ_1 and new constraints of the form $c \leq 0$ for each clock $c \in \lambda_1$. For a guard reset-set pair $\langle \phi, \lambda_2 \rangle$ where $\varphi[\lambda_1 := 0] \implies \phi$, $z' \wedge \phi$ is not empty. Thus, from all locations l' reached with clock zone z', transitions with a guard reset pair $\langle \phi, \lambda_2 \rangle$ are enabled.

Moreover, if $\lambda_2 \subseteq \lambda_1$, i.e., all clocks in λ_2 are already reset as they are also in λ_1, and $\varphi[\lambda_1 := 0] \implies \phi$, z' contains only valuations that are reachable in locations l'' reachable from l' by these transitions: In the original algorithm, they would be found to be reachable from states of the form $\langle l', z' \rangle$ with a clock zone $z'' = (z' \wedge \phi)[\lambda_2 := 0]^{\Uparrow}$. Note that all valuations in z' are also in z'':

Proof. To show: $z' \subseteq z''$
Expand z' in z'' to obtain

$$z'' = (((z \wedge \varphi)[\lambda_1 := 0]^{\Uparrow}) \wedge \phi)[\lambda_2 := 0]^{\Uparrow}$$

The zone z'' only looses valuations if the inner time elapse is omitted:

$$((z \wedge \varphi)[\lambda_1 := 0] \wedge \phi)[\lambda_2 := 0]^{\Uparrow} \subseteq (((z \wedge \varphi)[\lambda_1 := 0]^{\Uparrow}) \wedge \phi)[\lambda_2 := 0]^{\Uparrow}$$

Because of $\varphi[\lambda_1 := 0] \implies \phi$, it holds that

$$((z \wedge \varphi)[\lambda_1 := 0] \wedge \phi) = (z \wedge \varphi)[\lambda_1 := 0]$$

thus,

$$((z \wedge \varphi)[\lambda_1 := 0] \wedge \phi)[\lambda_2 := 0]^{\Uparrow} = ((z \wedge \varphi)[\lambda_1 := 0])[\lambda_2 := 0]^{\Uparrow}$$

holds. As furthermore $\lambda_2 \subseteq \lambda_1$, the clocks in λ_2 are already reset:

$$((z \wedge \varphi)[\lambda_1 := 0])[\lambda_2 := 0] = (z \wedge \varphi)[\lambda_1 := 0]$$

[2] Replacing invariants by adding them to all incoming or outgoing transitions will not make a previously reachable state unreachable. The semantics of invariants also only enable the incoming or outgoing transitions if the invariant is fulfilled. Of course, additional states can be reachable in a thus transformed model, e.g., automata can remain in a location even though the invariant would be violated in the original model. Replacing the invariants by adding them to the guards thus leads to an over-approximation of the reachable state-space.

this allows to prove that all valuations in z' are in z'':

$$((z \wedge \varphi)[\lambda_1 := 0]^{\Uparrow} \subseteq (((z \wedge \varphi)[\lambda_1 := 0]^{\Uparrow}) \wedge \phi)[\lambda_2 := 0]^{\Uparrow}$$

□

Thus, after the execution of the transitions associated with the guard reset-set pair $\langle \varphi, \lambda_1 \rangle$, all transitions associated with such guard reset-set pairs $\langle \phi, \lambda_2 \rangle$ can be executed. The transitions will be enabled for the clock zone z'. The argument above applies to the locations reached by this step as well, thus the step can be repeated until no further locations are found. States with locations reached in this way and a valuation from z' are indeed reachable. Of course, z'' might contain more valuations, so this algorithm delivers an underapproximation of the reachable valuations. This will be fixed later, as the clock zone z'' will be found upon application of the transition BDD associated with $\langle \phi, \lambda_2 \rangle$ to the BDD representing the locations reachable with clock zone z'. But again, this allows to grow the BDDs representing the locations faster, as it takes a look ahead at locations that are reachable, but would only have been found later when their exact clock zones are computed, if the approach from [EGP10] had been used. This gives rise to the description of the approach as an "underapproximating look-ahead".

Note that this approach does not only apply to the example of DBMs and BDDs, but can be generalized to a setting of continuous state sets mapping to discrete state sets, where transitions between discrete states are guarded by constraints on the continuous states, and transitions can have side-effects on the continuous state (like resets in case of clock zones).

Extended post operator

Generally, if the enabledness of a transition only depends on its guard φ, let the semi-order G_I be defined through implication: $G_I(\varphi, \phi)$ if and only if $\varphi \implies \phi$. G_I can easily be extended to guard side-effect pairs, e.g., guard reset-set pairs, as follows:

$$G_I(\langle \varphi, \lambda \rangle, \langle \phi, \lambda \rangle) \text{ if and only if } G_I(\varphi, \phi)$$

This provides the notion of "guard also fulfilled, same side effects".

When building the symbolic transition relation, each guard side-effect pair will map to a set of transitions. For a guard side-effect pair $\langle \varphi, \lambda \rangle$, this set will be the smallest set that contains all the transitions labeled with $\langle \varphi, \lambda \rangle$ for the conventional symbolic post operator. For the post operator extended by G_I, it will additionally contain all the transitions labeled with a guard side-effect pair $\langle \phi, \lambda \rangle$ such that $G_I(\langle \varphi, \lambda \rangle, \langle \phi, \lambda \rangle)$.

Algorithm 1 can be extended with the extended post operator as shown in Algorithm 4. The iteration over the guard side-effect pairs is made explicit to clarify the input to the extended post operator. So for each guard side-effect pair $\langle \varphi, \lambda \rangle$, the successor states reachable in one step from the pre-fixed-point with the transitions associated with $\langle \varphi, \lambda \rangle$ will be added to the next pre-fixed-point. Moreover, due to the extended

Algorithm 4 Computing the set of reachable states R using a post operator extended with G_I.[3]

$R_0 := \{\text{initial states}\}$
$i := 0$
repeat
  $i := i + 1$
  $R_i := R_{i-1}$
  for all guard side effect pairs $\langle \varphi, \lambda \rangle$ **do**
    $R_i := R_i \cup \mathsf{post}_{G_I}(R_{i-1}, \langle \varphi, \lambda \rangle)$
  end for
until $R_i = R_{i-1}$
$R := R_i$

Algorithm 5 Computing the set of reachable states R using a post operator extended with G_I by iterating over the states in the pre-fixed-point.[4]

$R_0 := \{\text{initial states}\}$
$i := 0$
repeat
  $i := i + 1$
  $R_i := R_{i-1}$
  for all guard side effect pairs $\langle \varphi, \lambda \rangle$ with associated discrete transition relation $T_{\langle \varphi, \lambda \rangle}$ **do**
    for all pairs of sets of continuous states and discrete states $\langle z, d \rangle$ in R_{i-1} **do**
      $R_i := R_i \cup \mathsf{post}_{G_I}(\langle z, d \rangle, \langle \varphi, \lambda \rangle)$
    end for
  end for
until $R_i = R_{i-1}$
$R := R_i$

post operator, also the discrete states reachable in one step from the pre-fixed-point with transitions associated with guard side-effect pairs $\langle \phi, \lambda \rangle$ where $G_I(\langle \varphi, \lambda \rangle, \langle \phi, \lambda \rangle)$ will be added to the next pre-fixed point. In this setting of symbolic state space representation, the state space is represented by a set of pairs of sets of continuous states, e.g., clock zones, and sets of discrete states, e.g., BDDs. So the discrete states found due to including the transitions labeled with $\langle \phi, \lambda \rangle$ will be paired with only a subset of the continuous states that they would have been reachable with, namely with the set of continuous states that would result if those transitions had been associated with $\langle \varphi, \lambda \rangle$ instead.

To make the iteration over the components of the stored reachable state space more explicit, consider the more detailed Algorithm 5. This allows to give a more

[3] This Algorithm extends Algorithm 1. It is inspired by Algorithm 2.
[4] This Algorithm is a more detailed version of Algorithm 4. It is inspired by Algorithm 2.

formal description of the extended post operator:

$$\mathsf{post}_{G_I}(\langle z, d\rangle, \langle \varphi, \lambda\rangle) := \begin{cases} \emptyset & \text{if } z \wedge \varphi \text{ is empty} \\ \{\langle \lambda(z \wedge \varphi)^{\Uparrow}, T_{G_I(\langle \varphi,\lambda\rangle)}(d)\rangle\} & \text{otherwise} \end{cases}$$

where

$$T_{G_I(\langle \varphi,\lambda\rangle)}(d) := T_{\langle \varphi,\lambda\rangle}(d) \cup \bigcup_{\langle \phi,\lambda\rangle \text{ with } G_I(\langle \varphi,\lambda\rangle,\langle \phi,\lambda\rangle)} T_{\langle \phi,\lambda\rangle}(d)$$

is a precomputed G_I-extended transition relation, $\lambda(x)$ means applying the side effects in λ to x, and $(x)^{\Uparrow}$ means executing all continuous steps (e.g., passing time) that are independent of discrete steps.

Note that Algorithm 5 can associate some discrete states with several sets of continuous states, some of which can be fully contained in others of them, or even be the same. This fragmentation of the state space will be reduced by treating the set of reachable state not a set of pairs, but a mapping from sets of continuous states, e.g., clock zones, to sets of discrete states, e.g., BDDs; i.e., each set of continuous states will only show up once, but some sets may still be fully contained in others. The remaining fragmentation of the state space will be that some discrete states can be associated with several sets of continuous states, some of them being redundant. Adapted like this, the approach of Algorithm 6 does not introduce continuous states that would not have been introduced otherwise. In the first iteration it does not even introduce sets of continuous states that would not otherwise have been introduced as sets of their own, it just associates more discrete states with the sets of continuous states.[5] Thus, it makes the discrete state sets grow faster. As this growth is due to redundancy being introduced, it can be argued that the growth is not helpful, as each of the discrete states additionally found to be reachable would have been found later in the iteration over all guard side-effect pairs before the computation of the next pre-fixed point.[6] However, an idea from [EGP10, Section 3.4] is also applicable here: As it is not necessary to preserve the meaning of the index i (as in R_i containing the states reachable in i steps), states newly found to be reachable can be immediately added to the pre-fixed point that is used for finding new states. All it takes is the small change in selecting the $\langle z, d\rangle$ in the inner loop not from the previous pre-fixed point R_{i-1} but the current pre-fixed point R_i as demonstrated in Algorithm 6. This then allows to use the bigger discrete state sets already in the computation of the current pre-fixed point, as opposed to only in the computation of the next pre-fixed point. Thus, the symbolic operations can be applied to bigger sets earlier, allowing to potentially find the fixed point earlier, i.e., calculating fewer pre-fixed points.

To add treatment of location invariants to the approach, invariants are enforced before merging the newfound states with the set of already known to be reachable states. In the timed automata setting used in this chapter, invariants may only enforce upper bounds on continuous variables, that could only be violated when the $(x)^{\Uparrow}$ operation in

[5] It can however add to fragmentation in later iterations by introducing subsets of sets of continuous states as sets of their own, as discussed in Section 7.4.1 *Fragmentation*.

[6] See Section 7.4.1 *Fragmentation* for an argument why this approach can be a bad idea.

Algorithm 6 Computing the set of reachable states R by iterating over the states in the "next" pre-fixed-point R_i and treating invariants.[7]

$R_0 := \{\text{initial states}\}$
$i := 0$
repeat
  $i := i + 1$
  $R_i := R_{i-1}$
  for all guard side effect pairs $\langle \varphi, \lambda \rangle$ with associated discrete transition relation $T_{\langle \varphi, \lambda \rangle}$ **do**
    for all pairs of sets of continuous states and discrete states $\langle z, d \rangle$ in R_i **do**
      $\{\langle z', d' \rangle\} := \mathsf{post}_{G_I}(\langle z, d \rangle, \langle \varphi, \lambda \rangle)$
      for all pairs $\langle \iota, d_\iota \rangle$ of invariant ι and set of locations d_ι where ι has to hold **do**
        if $d' \cap d_\iota \neq \emptyset$ **then**
          $R_i :=$ merge $\{\langle z' \wedge \iota, d' \cap d_\iota \rangle\}$ with R_i
        end if
      end for
    end for
  end for
until $R_i = R_{i-1}$
$R := R_i$

the post operator relaxes their upper bounds, as the location invariants are required to be a part of the guards on all transitions connecting to their location and clock resets cannot violate upper bounds. More formally, for all guards φ of incoming edges of a location l with invariant ι_l, $\varphi \implies \iota_l$. In particular, if $\{\langle z', d' \rangle\} = \mathsf{post}_{G_I}(\langle z, d \rangle, \langle \varphi, \lambda \rangle)$, for all invariants ι_l associated with a location $l \in d'$, $\varphi \implies \iota_l$ and thus $z' \wedge \iota_l \neq \mathbf{false}$ as $z \wedge \varphi$ is not empty, ι_l can only impose upper bounds, and λ cannot empty a set of continuous states or violate upper bounds by setting variables to the minimal value 0. With $\lambda(z \wedge \varphi)$ not being empty, the relaxation of upper bounds in $z' = \lambda(z \wedge \varphi)^{\Uparrow}$ can violate ι_l, but $z' \wedge \iota_l$ cannot become empty. In a more general setting, only invariants that are implied by all incoming guards and cannot be violated by side effects are allowed, so only the continuous operations independent of discrete transitions may invalidate invariants. As invariants are based on location, i.e., on discrete state, they may vary for the members of the set of discrete states being merged, thus resulting in different sets of continuous states resulting from the application of different invariants to the set of continuous states.

Discrete Lookahead

The only side effects, e.g., resets, of a transition besides the change of the discrete state are made explicit by the set of side effects λ_1 assigned to the transition. Let

[7] This Algorithm extends Algorithm 5. It is inspired by Algorithm 3.

the semi-order R on sets of side effects be defined by hiding: $R(\lambda_1, \lambda_2)$ if and only if for any state x, the effects of applying λ_1 hide any consecutive application of λ_2, formally $\forall x.\lambda_1(x) = \lambda_2(\lambda_1(x))$. Let $\lambda(\varphi)$ denote the guard equivalent with "φ held before applying λ, and λ has been applied": $\lambda(\varphi)(y) = (\exists x.(x \implies \varphi) \wedge (\lambda(x) = y))$ where $\lambda(x)$ means applying the side effects in λ to x. As an example, consider $\varphi[\lambda := 0]$ from above: the resets are applied using the new $c \leq 0$ constraints for all clocks $c \in \lambda$ and, assuming φ was not contradictory, there was a valuation $\vec{v}$ such that $\vec{v} \vDash \varphi$, and consequently $\vec{v}[\lambda := 0]$ satisfies all the constraints from φ that do not talk about a clock from λ, so in this example, $\lambda(\varphi) = \varphi[\lambda := 0]$.

This gives rise to the semi-order G_S, where

$$G_S(\langle\varphi, \lambda_1\rangle, \langle\phi, \lambda_2\rangle) \text{ if and only if } R(\lambda_1, \lambda_2) \wedge (\lambda_1(\varphi) \implies \phi),$$

providing the notion of "guards also fulfilled, side-effects already taken care of".

The empty guard side-effect pair is special in the sense that its associated transitions are always enabled, and they have no side effects which would have to be taken care of. Thus, for all guard side-effect pairs $\langle\varphi, \lambda_1\rangle$, $G_S(\langle\varphi, \lambda_1\rangle, \langle\mathbf{true}, \emptyset\rangle)$ is fulfilled. To grow the initial reachable state space, which just has the continuous state set $0^{\Uparrow}$, it is advisable to first saturate the transitions associated with the empty guard side-effect pair. Algorithm 7 provides the context for the saturation post operator $\mathsf{post}_{G_S}(d, \langle\varphi, \lambda\rangle) := T_{G_S(\langle\varphi,\lambda\rangle)}(d)$ where

$$T_{G_S(\langle\varphi,\lambda\rangle)}(d) := \bigcup_{\langle\phi,\lambda'\rangle \text{ with } G_S(\langle\varphi,\lambda\rangle,\langle\phi,\lambda'\rangle)} T_{\langle\phi,\lambda'\rangle}(d)$$

During the state space exploration, after executing the transitions associated with a certain guard side-effect pair as well as those associated with a different guard but the same side-effects and the guard is surely fulfilled, as indicated by the semi-order G_I, the set of states L is found to be reachable. From those states, some other discrete states might be surely reachable, without changing the continuous state. Those discrete states are added to the newly reachable states using a fixed point iteration, saturating the set of surely reachable discrete states of the resultant continuous state.

This allows the set of discrete states that is associated with the resultant continuous state to be bigger, enabling a faster grow of the discrete state sets later on; this comes, however, at the cost of additional discrete successor computations during the saturation phase.

Fragmentation

In the original Algorithm 1, there may be redundant sets of continuous states associated with discrete states, if the representation of continuous states used can overlap or be included in one-another. Reducing this redundancy is often unfeasible, as for example checking for inclusion of a clock zone and it's associated discrete states in another such pair is expensive and would be required regularly. Also, two touching or overlapping convex clock zones associated with the same set of discrete states might

Algorithm 7 Saturating the discrete state space exploration for each discovered set of continuous states. Note: semicolon for sequential execution.[8]

$R_1 := R_0 := \{\{\langle 0^{\Uparrow} \wedge \iota_{l_0}, \{l_0\}\rangle\}$;
$\{\langle z, d\rangle\} := \mathsf{post}_{G_I}(\langle 0^{\Uparrow} \wedge \iota_{l_0}, \{l_0\}\rangle, \langle \mathbf{true}, \emptyset\rangle)$;
$i := 1$; $d' := d$;
repeat
  $d := d \cup d'$; $d' := \mathsf{post}_{G_S}(d, \langle \mathbf{true}, \emptyset\rangle)$;
until $d' \subseteq d$
for all pairs $\langle \iota, d_\iota\rangle$ of invariant ι and set of locations d_ι where ι has to hold **do**
  if $d \cap d_\iota \neq \emptyset \wedge (z \wedge \iota \neq \emptyset)$ **then**
    $R_1 :=$ merge $\{\langle z \wedge \iota, d \cap d_\iota\rangle\}$ with R_1;
  end if
end for
repeat
  $i := i + 1$; $R_i := R_{i-1}$;
  for all guard side effect pairs $\langle \varphi, \lambda\rangle$ with associated discrete transition relation $T_{\langle \varphi, \lambda\rangle}$ **do**
    for all pairs of sets of continuous states and discrete states $\langle z, d\rangle$ in R_i **do**
      $L := \mathsf{post}_{G_I}(\langle z, d\rangle, \langle \varphi, \lambda\rangle)$;
      if $L \neq \emptyset$ **then**
        $\{\langle z', d\rangle\} := L$; $d' := d$;
        repeat
          $d := d \cup d'$; $d' := \mathsf{post}_{G_S}(d', \langle \varphi, \lambda\rangle)$;
        until $d' \subseteq d$
        for all pairs $\langle \iota, d_\iota\rangle$ of invariant ι and set of locations d_ι where ι has to hold **do**
          if $d \cap d_\iota \neq \emptyset \wedge (z \wedge \iota \neq \emptyset)$ **then**
            $R_i :=$ merge $\{\langle z' \wedge \iota, d \cap d_\iota\rangle\}$ with R_i;
          end if
        end for
      end if
    end for
  end for
until $R_i = R_{i-1}$
$R := R_i$

be representable in one convex clock zone, but again, regular checks for this would be prohibitively expensive. So this type of fragmentation seems unavoidable. However, the approach presented in this chapter might cause a lot of additional fragmentation of the former type. The clock zones initially found by the approach described in this chapter will only be clock zones that would have been found anyway, they just have

[8] This Algorithm extends Algorithm 6 and is thus inspired by Algorithm 3.

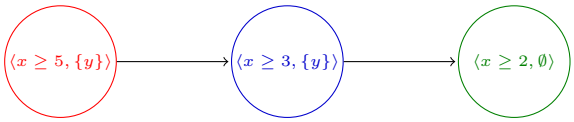


Figure 7.1: Example iteration order tree induced by $G_S(\langle x \geq 3, \{y\}\rangle, \langle x \geq 2, \emptyset\rangle)$, $G_S(\langle x \geq 5, \{y\}\rangle, \langle x \geq 3, \{y\}\rangle)$ and $G_S(\langle x \geq 5, \{y\}\rangle, \langle x \geq 2, \emptyset\rangle)$

more associated discrete states. Lets call those discrete states the clock zone z's *additional discrete states*, as opposed to the *regular discrete states* that would also have been associated with z using the original Algorithm 1, and lets call a clock zone that is associated with a discrete state as a regular discrete state a *regular clock zone* of this discrete state. However, some of the additional discrete states of z might have enabled outgoing transitions with a guard reset pair $\langle \varphi, \lambda \rangle$ that would not be on any of the outgoing transitions of z's regular discrete states. If these transitions are executed, the intersection of their guard φ with z might be more restrictive than the intersection of the guard φ with all regular clock zones z' of these additional discrete states that are associated with z and have an outgoing enabled transition with the guard reset pair $\langle \varphi, \lambda \rangle$. Then, applying the symbolic transition associated with $\langle \varphi, \lambda \rangle$ to z's associated locations does not result in an empty set, but in a set of locations now associated with a clock zone arising from $z \wedge \varphi[\lambda := 0]\Uparrow$, which might be a clock zone that would not have arisen at all in the original Algorithm 1. This clock zone is redundant, namely it is contained in clock zones $z' \wedge \varphi[\lambda := 0]\Uparrow$ that arise if the transitions associated with $\langle \varphi, \lambda \rangle$ are applied to some such regular clock zone z'.

This additional fragmentation can be harmful by making the state space representation bigger than necessary because of added redundant data entries, thus also making iterating over all the entries in the state space representation slower.

Iteration order

To make the most of the potential benefit of the lookahead, the choice of in what order to pick the guard side-effect pairs in the for-all clauses in the algorithm can be made informed instead of arbitrary. The algorithm could choose the guard side-effects pairs to explore first such that it explores the ones that are implied by others before it explores the implying ones. A semi-order G on the guard side-effect pairs induces a forest with shared nodes, where the nodes are labeled with the respective guard side-effect pairs, as shown in Figure 7.1. This forest will be explored, giving rise to the order of exploration of the guard side-effect pairs for the reachability checking.

The use of such an exploration-order-control using this forest has two advantages: First, the extended transition relation for the guard side-effect pairs at the roots will be bigger if the right semi-order, e.g., G_I, is used, making the discrete state space known to be reachable grow faster. Second, the exploration order of the forest can be adjusted, allowing for several optimization possibilities. For example, searching the roots first allows to grow the discrete state space very fast.

If depth-first search is employed on a forest induced by G_S and Algorithm 7 is used,

after the respective root's guard side-effect pair's transition relation has been saturated, no new discrete states can be found to be reachable from the newly discovered discrete states using the transition relations associated with the guard side-effect pairs from nodes on a path from this root to a leaf, because these discrete transitions are already all executed during the saturation phase until no new reachable discrete states are found.

However, it is still necessary to eventually explore the tree below the root in order to find the exact continuous states. Moreover, the transitions further down in the tree can give rise to new locations reachable in one step as well,[9] if they are reachable from the pre-fixed point with those transitions, but are not reachable with said transitions from the newly discovered to be reachable states. The fact that the newly reachable states do not give rise to more reachable states below the root allows to store the results temporarily at the edges of the forest, and only add them to the set of states from which to explore when backtracking over the edge. That will enable working from a smaller set of reachable states without overlooking reachable discrete states. These smaller sets of states are smaller in such a sense as they can contain fewer clock zones with associated discrete-state-BDDs, thus hopefully reducing the number of BDDs to iterate over.

Note that this temporary storage method would also allow to mitigate one of the drawbacks of the approach presented in this chapter, namely fragmentation of the state-space due to many locations being associated with several clock zones where some are contained in others: When backtracking, the nodes closer to the root give rise to clock zones included in the ones given rise to by nodes further to a leaf on a path to this leaf, so the locations could be associated with the biggest of these clock zones when backtracking: When backtracking over an edge, add the associated results to the ones stored at the node (initially none). When backtracking away from a node over an incoming edge, remove the locations from results associated with the node from the edge and then add the stored results from the node to the ones from the edge. Thus, if a location is associated with a clock zone surely included in a clock zone that is also associated with this location, the location can be removed from the included clock zone to reduce redundancy.

Pre-computing the transition relations and the semi-orders

In order to build the symbolic transition relations, each transition has to be added to some transition relation according to its guard side-effects pair. This can be done by iterating once over all transitions. During this iteration, the semi-orders G_S and G_I need to be computed using the implication-semi-order on guards, and the implication-semi-order on side-effects.

In the case of timed automata, the semi-order on guards is a semi-order on conjunctions of clock constraints. For each clock x, the guard is either not concerned with x, or the guard contains exactly one constraint concerning x of one of the following

[9] Note that those states could be discovered as well during saturation, if the forest was induced by $G_S \wedge G_I$.

forms: $x < c$, $x \leq c$, $x == c$, $x \geq c$, or $x > c$, where c is some constant in $\mathbb{N}_0$ and $x == c$ is represented by $x \leq c \wedge x \geq c$. The constraints concerning x are semi-ordered in the obvious way:

- $x < c \implies x < c'$ if and only if $c \leq c'$
- $x < c \implies x \leq c'$ if and only if $c \leq c'$
- $x \leq c \implies x < c'$ if and only if $c < c'$
- $x \leq c \implies x \leq c'$ if and only if $c \leq c'$
- $x == c \implies x < c'$ if and only if $c < c'$
- $x == c \implies x \leq c'$ if and only if $c \leq c'$
- $x == c \implies x \geq c'$ if and only if $c \geq c'$
- $x == c \implies x > c'$ if and only if $c > c'$
- $x \geq c \implies x \geq c'$ if and only if $c \geq c'$
- $x \geq c \implies x > c'$ if and only if $c > c'$
- $x > c \implies x \geq c'$ if and only if $c \geq c'$
- $x > c \implies x > c'$ if and only if $c \geq c'$

Now for a guard φ, let $clks(\varphi)$ be the set of clock variables for which φ contains a constraint, and for clock x, let $clkc_x(\varphi)$ be this constraint. So for two guards φ, ϕ, the implication-semi-order is

$$\varphi \implies \phi \text{ if and only if } clks(\phi) \subseteq clks(\varphi) \wedge \forall x \in clks(\phi) : clkc_x(\varphi) \implies clkc_x(\phi).$$

Timed automata have only resets of clocks to 0 as side-effects, so the semi-order R on reset sets is just inclusion: For two reset sets λ_1, λ_2,

$$R(\lambda_1, \lambda_2) \text{ if and only if } \lambda_2 \subseteq \lambda_1.$$

The effect of applying a reset set to a guard is also straightforward:

$$\lambda(\varphi) := (\bigwedge_{y \in \lambda} y == 0) \wedge (\bigwedge_{x \in clks(\varphi) \setminus \lambda} clkc_x(\varphi))$$

Combining this, the implication semi-order on guards and resets applied to guards is:

$$\begin{aligned} \lambda(\varphi) \implies \phi \text{ if and only if } & clks(\phi) \subseteq clks(\varphi) \wedge \lambda \subseteq clks(\phi) \\ & \wedge \forall x \in clks(\phi) \setminus \lambda : clkc_x(\varphi) \implies clkc_x(\phi) \\ & \wedge \forall y \in \lambda : clkc_y = (y == 0) \end{aligned}$$

In the following, an example algorithm of how the semi-orders and the iteration order tree can be computed will be presented. During the initial iteration over all transitions, the BFs for the symbolic transition relation are created.

In essence, an approach with quadratic runtime can just compare every guard/reset-set pair with all the others after the transition relations have been built. Then, it can just join the appropriate transition relations to build the G_I and G_S transition relations. If a representation of the semi-order without transitive edges is needed, as it may lead to more informed heuristics, the runtime becomes cubic.

However, this can be improved using several optimizations or heuristics. For each pair of a set of clocks and a reset-set which derive from the pairs of guards and reset sets by extracting the sets of clocks mentioned in the guard, other such pairs can be excluded immediately from the consideration. Given a pair $\langle clks, \lambda_1 \rangle$ of a set of clocks $clks$ and a reset set λ_1, for G_I a pair $\langle clks', \lambda_2 \rangle$ can be quickly excluded if (a) the reset set is not the same ($\lambda_1 \neq \lambda_2$) or if (b) some clocks are not treated in the original set ($clks' \not\subseteq clks$). Only if these checks do not exclude a pair, then (c) the actual guard/reset-set pairs $\langle \varphi, \lambda_1 \rangle$ with $clks = clks(\varphi)$ and $\langle \phi, \lambda_2 \rangle$ with $clks' = clks(\phi)$ have to be checked using $\exists x \in clks' : clkc_x(\varphi) \not\Rightarrow clkc_x(\phi)$ as an exclusion criterion.

Exploiting exclusion criterion (a), guard/reset-set pairs can be grouped according to the reset-sets, and the checks only need to be performed inside these groups. The worst case of this pre-computation, namely that all guard/reset-set pairs have the same reset-set, or there would only be a very small number of reset-sets, would be a good application of the overall approach, as the fewer reset-sets there are, the more likely it is that guard/reset-set pairs are in the G_I and G_S relations, as their requirements on the reset-sets are trivially fulfilled if the reset sets are the same.

For G_S these criteria are slightly different. For convenience, it is assumed that whenever a guard contains a constraint of the form $x == 0$, x is also included in the reset set of that guard reset-set pair. This allows to capture all situations in which a clock is known to be zero after application of a guard reset-set pair using just the reset set. So for the pairs of sets of clocks and reset-sets, the exclusion criteria for G_S are (A) the reset-set is not contained ($\lambda_2 \not\subseteq \lambda_1$), (B) some clocks are not treated in the original set and not reset ($clks' \not\subseteq clks \cup \lambda_1$), and (C) again relies on the actual guard reset pairs as in the G_I case, but uses the modified exclusion criterion $\exists x \in clks' : \lambda_1(clkc_x(\varphi)) \not\Rightarrow clkc_x(\phi)$.

To exploit criterion (A), either an inclusion relation on the reset-sets is constructed, namely a graph representing the transitive reduction of the transitive closure of the partial order given by set-inclusion. As Aho et al. shows in [AGU72], for n sets, this is as expensive as $(n \times n)$-matrix multiplication, for which the best known algorithm is in $O(n^{2.3728639})$, as shown by [LG14]. On this graph, only the pairs associated with reset-sets associated with nodes reachable from the node associated with the reset-set under consideration would have to be checked. Or, as a cheaper alternative, which would of course be less effective, collect the reset-sets according to their cardinality, and only check pairs with reset-sets of smaller size and the pairs with the reset-set under consideration.

To build the inclusion partial order, assume a set r is entered in the order, rep-

resented by a forest with shared nodes, which is the same as a *directed acyclic graph* (DAG). If transitive edges are allowed, a naive quadratic approach comparing each set with all the others will suffice. Otherwise, a breadth first search looks for all nodes where r is included in the nodes set:

- If such a node is found, immediately all children of the node are checked: If none of them has a set that includes r, the node is added to the set of nodes including r, otherwise, the node is not included.
- If a node's set is included in r, the node's set is checked against the set of included sets:
  - If the node's set is included in one of the elements, the check can end early,
  - if the set includes one of the elements, that element is removed from the set,
  - if the check against the set is finished regularly, not early, the node's set is added to the set.
- If a node's set has an empty intersection with r, the nodes children are not added to the to-search FIFO.
- If a node's set intersection with r is non empty, the children are added to the to-search FIFO.

After the FIFO is empty, a new node with set r is created, and edges from all the sets in the set of including sets to this new node are added. Edges from the new node to all sets from the included set are also added. Finally, all direct edges from a node in the set of including sets to a node in the included set are deleted, as they are now transitive to the path through the new node. This algorithm should work in $O(n^3)$. Note that in this case, n is just the number of reset-sets, which is hopefully much smaller than the number of guard/reset-set pairs.

For direct access to a specific node, a hash map mapping reset sets to the corresponding node can be used. Each node $n(r)$ will link to a set of clocks *clks*. This set can be organized again as a partial order DAG, using the subset partial order of the clocks treated in a guard. This structure can be built using the same algorithm as for the rest-set DAG, and also have a hash map for quick access to the node associated with a specific set of clocks. The algorithm would again be in $O(n^3)$, but this time n would be just the number of sets of clocks treated by guards with this specific reset set. If all the number of sets of clocks treated by guards associated with a some reset-set are summed up over all reset-sets, this number is bounded by the number of guard/reset-set pairs, but hopefully much smaller. So the complexity of building these auxiliary structures is not asymptotically greater than building the final structures directly, but it could be much less work in many cases. Each of the nodes would thus be labeled with set of clocks *clks*, and would link to all guards φ with $clks = clks(\varphi)$ that are paired with

the reset-set from the node $n(r)$ on some transition. Of course, these guard/reset-set pairs would also use a hash map to link to their associated transitions.

For building the G_I transition relation, using criterion (a), only for each reset-set, the guards associated with it have to be checked. Using criterion (b), only for the sets of treated clocks that are included, the guards have to be checked. So by iterating over all reset-sets, for a given reset-set, pick a node of the treated clock sets, and pick one of the guards with this set, and compare it against all the others associated with this set of treated clocks, and with all the included sets, i.e., the sets from nodes reachable from there, using criterion (c). It seems advisable to do this in a bottom-up fashion, starting at the leaves of the treated clock sets DAG. That would enable yet another optimization, albeit a small one, namely, marking a node when all the guards associated with all its associated clock sets have been treated and found to be in relation G_I with all the guards that could be reachable from there, enabling an early termination of later checks after this node, as in that case, all transitions further down would already be included in the respective G_I transitions.

Similarly, for building the G_S transition relation, using criterion (A), for each reset-set, all guard reset-set pairs where the reset-set is included have to be checked. This could be done by exploring the reset set inclusion DAG, starting from the node associated with the chosen reset set. Using criterion (B), only for the sets of treated clocks that are included or reset, the guards have to be checked. Along with the treated clock DAGs and the hash-map for direct access to its nodes, a data-structure can be constructed that allows to access the sets of treated clocks for each reset set according to the size of the set: Have the sets ordered into a list according to their size in a descending fashion, and from $|X|$ down to 0, let each number point to the first set in the list which has this size, or a smaller one. In each node of the reset-set set inclusion DAG, only check all sets of treated clocks whose size is smaller or equal to $\mathit{clks} \cup \lambda_1$, using criterion (B). Only the guards that remain after this process need to be checked with criterion (C). Note that a trick similar to the one used for criterion (b) above is not applicable here, as there are multiple sets of treated clock sets involved due to multiple reset sets being involved, the set λ_1 is not known at the time of the pre-computation, and a direct access to a surely included treated clock set to start the search from is not always possible anyway, as the treated clock sets associated with other reset-sets might not be the same.

Iteration Order DAG The iteration order DAG can be built using data collected during the computation of G_I and G_S. For G_S, for each guard reset-set pair $\langle \varphi, \lambda_1 \rangle$, during the computation of the semi-order, all guard reset-set pairs $\langle \phi, \lambda_2 \rangle$ such that $G_S(\langle \varphi, \lambda_1 \rangle, \langle \phi, \lambda_2 \rangle)$ are stored. The same can be done for G_I. If the iteration order is induced by G_I or G_S building the DAG is straightforward. To build the DAG in $\mathrm{O}(n^3)$, for each guard reset-set pair $\langle \varphi, \lambda_1 \rangle$ a node is created, labeled with this pair. Then, for each guard reset-set pair $\langle \varphi, \lambda_1 \rangle$, all the guard reset-set pairs $\langle \phi, \lambda_2 \rangle$ stored for this pair are checked: If a pair $\langle \phi, \lambda_2 \rangle$ is not stored for some other pair stored for $\langle \varphi, \lambda_1 \rangle$, an edge is added from the node for $\langle \varphi, \lambda_1 \rangle$ to the node for $\langle \phi, \lambda_2 \rangle$. If the order

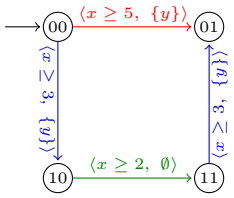


Figure 7.2: An example of a timed automaton with $G_I(\langle x \geq 5, \{y\}\rangle, \langle x \geq 3, \{y\}\rangle)$, $G_S(\langle x \geq 3, \{y\}\rangle, \langle x \geq 2, \emptyset\rangle)$ and $G_S(\langle x \geq 5, \{y\}\rangle, t)$ for $t \in \{\langle x \geq 3, \{y\}\rangle, \langle x \geq 2, \emptyset\rangle\}$

is induced by $G_I \wedge G_S$, the stored sets of guard reset-set pairs from G_I and G_S are first intersected, then the DAG can be built in the same way.

7.5 An Example with a Comparison to Symbolic Forward Reachability-Checking

This section will compare the approach presented in this chapter to the approach from [EGP10] by applying both approaches to the timed automaton shown in Figure 7.2. The reachable state space will be stored in a mapping of DBMs to BDDs, the only side effects are clock resets. The discrete state space is described in terms of the variables l_0 and l_1, where l_0 is the rightmost digit of the locations label bitstring, and l_1 is the leftmost digit, i.e., state 01 will be represented by $l_0 \wedge \neg l_1$.

Both approaches start by creating the symbolic transition relations for each guard reset pair. The CZM approach will iterate once over all edges and add them to a transition BDD for each guard reset pair. The approach presented in this chapter will do the same. But it will additionally calculate the G_S and G_I semi-orders, and build the G_S induced DAG. For each $G_I(a, b)$ that is discovered, it will add the transition BDD for b to the G_I transition BDD for a, which, in turn, will be initialized with the transition BDD for a. Likewise, for each $G_S(a, b)$ that is discovered, it will add the transition BDD for b to the G_S transition BDD for a.

More concretely, the CZM approach will calculate the mapping for the transition BDD:

⟨guard,resets⟩		BF
$\langle x \geq 5, \{y\}\rangle$	$\mapsto$	$\neg l_0 \wedge \neg l_1 \wedge \neg l'_1 \wedge l'_0$
$\langle x \geq 3, \{y\}\rangle$	$\mapsto$	$(\neg l_0 \wedge \neg l_1 \wedge l'_1 \wedge \neg l'_0) \vee (l_0 \wedge l_1 \wedge \neg l'_1 \wedge l'_0)$
$\langle x \geq 2, \emptyset\rangle$	$\mapsto$	$l_0 \wedge \neg l_1 \wedge l'_1 \wedge l'_0$

The underapproximating lookahead approach will additionally calculate the tree shown in Figure 7.1 on page 77, and the mapping for G_I:

⟨guard,resets⟩		BF
$\langle x \geq 5, \{y\}\rangle$	$\mapsto$	$(\neg l_0 \wedge \neg l_1 \wedge \neg l'_1 \wedge l'_0) \vee (\neg l_0 \wedge \neg l_1 \wedge l'_1 \wedge \neg l'_0)$ $\vee (l_0 \wedge l_1 \wedge \neg l'_1 \wedge l'_0)$
$\langle x \geq 3, \{y\}\rangle$	$\mapsto$	$(\neg l_0 \wedge \neg l_1 \wedge l'_1 \wedge \neg l'_0) \vee (l_0 \wedge l_1 \wedge \neg l'_1 \wedge l'_0)$
$\langle x \geq 2, \emptyset\rangle$	$\mapsto$	$l_0 \wedge \neg l_1 \wedge l'_1 \wedge l'_0$

The mapping for G_S will also be calculated:

⟨guard,resets⟩		BF
$\langle x \geq 5, \{y\}\rangle$	$\mapsto$	$(\neg l_0 \wedge \neg l_1 \wedge \neg l'_1 \wedge l'_0) \vee (\neg l_0 \wedge \neg l_1 \wedge l'_1 \wedge \neg l'_0)$
		$\vee (l_0 \wedge l_1 \wedge \neg l'_1 \wedge l'_0) \vee (l_0 \wedge \neg l_1 \wedge l'_1 \wedge l'_0)$
$\langle x \geq 3, \{y\}\rangle$	$\mapsto$	$(\neg l_0 \wedge \neg l_1 \wedge l'_1 \wedge \neg l'_0) \vee (l_0 \wedge l_1 \wedge \neg l'_1 \wedge l'_0) \vee (l_0 \wedge \neg l_1 \wedge l'_1 \wedge l'_0)$
$\langle x \geq 2, \emptyset\rangle$	$\mapsto$	$l_0 \wedge \neg l_1 \wedge l'_1 \wedge l'_0$

The initial state set is $\langle 0 \uparrow, \{00\}\rangle$ in both approaches. The CZM approach will iterate over all stored clock zones, but initially there is only one. To this clock zone and its associated location BDD, all guard reset-set pairs and their associated transition BDDs will be applied. The specific order in which they are applied is not specified in [EGP10]. However, for the optimization used in [EGP10, Section 3.4], this order may be relevant. So lets look at the best case and worst case scenarios separately. In the best case for some clock zone z, if there is a guard reset-set pair whose application to z results in z again, this guard reset-set pair is applied first. If there are several such pairs, the ones associated with transitions leading to not yet known to be reachable locations are applied first. If again there are several of those, the ones leading to new locations from where the other such pairs' associated transitions lead to not yet known to be reachable locations come first, and so on. There seems to be no way of enforcing the best case that does not involve as much computation as applying the best case would save, so the best case is a theoretical construct in the CZM approach. As is the worst case, which is just the reversed best case order.

However, in this example in the first round there is no guard reset-set pair that would preserve $0 \uparrow$. In the first round, the CZM approach calculates the state set:

clock zone	BF
$0 \uparrow$	$\{00\}$
$x \geq 5 \uparrow$	$\{01\}$
$x \geq 3 \uparrow$	$\{10\}$

This will be done with 3 successor clock zone calculations, and three BDD applications, one for each guard reset-set pair. In the second round, in the worst case, the CZM approach calculates the state set:

clock zone	BF
$0 \uparrow$	$\{00\}$
$x \geq 5 \uparrow$	$\{01\}$
$x \geq 3 \uparrow$	$\{10, 11\}$

This will be done with 3 more successor clock zone calculations, and three more BDD applications, one for each guard reset-set pair, for the clock zone $x \geq 3 \uparrow$, discovering a new location. The same is done again for the clock zone $x \geq 5$, which would find that the clock zone does not lead to new states anymore. So in total 9 successor clock zone calculations and 9 BDD applications so far in the worst case. However, in the best case, $\langle x \geq 2, \emptyset\rangle$ is applied to $x \geq 3 \uparrow$ before $\langle x \geq 3, \{y\}\rangle$ or $\langle x \geq 5, \{y\}\rangle$ are applied to it. Thus when evaluating clock zone $x \geq 3 \uparrow$, from the location 11 discovered with $\langle x \geq 2, \emptyset\rangle$ that is again associated with clock zone $x \geq 3 \uparrow$, $\langle x \geq 3, \{y\}\rangle$ discovers the location 01. So the CZM approach can explore the full reachable state space in this

example with just 5 successor clock zone calculations and 5 BDD applications. In this best case, with 9 successor clock zone calculations and 9 BDD applications, the CZM approach calculates the state set:

clock zone	BF
$0\uparrow$	$\{00\}$
$x \geq 5\uparrow$	$\{01\}$
$x \geq 3\uparrow$	$\{01, 10, 11\}$

In the worst case, this is the result after the third round, after 12 successor clock zone calculations and 12 BDD applications, as all guard reset-set pairs have to be applied to $x \geq 3\uparrow$. In the best case, the third round recognizes the fixed point and terminates after a total of 12 successor clock zone calculations and 12 BDD applications. In the worst case, the fixed point is only recognized after a fourth round, bringing the total to 15 clock zone calculations and 15 BDD applications.

The underapproximating lookahead approach will apply the guard reset-pairs based on the iteration order DAG from Figure 7.1. First, $\langle x \geq 5, \{y\}\rangle$ will be applied in the G_I induced form, with one successor clock zone calculation and one BDD application, recognizing the newly reachable states $\langle x \geq 5, \{01, 10\}\rangle$. Then, the G_S induced form will be applied until saturation, resulting in $\langle x \geq 5, \{01, 10, 11\}\rangle$ after one additional BDD application, and then recognizing saturation after another one. Note that already with two BDD operations and one successor clock zone calculation, all reachable locations have been discovered. So a location based error condition would already be met, making this approach very useful for error finding. Second, $\langle x \geq 3, \{y\}\rangle$ is applied in the G_I induced form, with a second successor clock zone calculation, and the 4th BDD application, recognizing the newly reachable states $\langle x \geq 3, \{10\}\rangle$. Note that as this is a depth first situation in the iteration order DAG, so the states just discovered could be ignored. Then, the G_S induced form is saturated, resulting in $\langle x \geq 3, \{10, 11\}\rangle$ after one additional iteration, $\langle x \geq 3, \{01, 10, 11\}\rangle$, after another one, and recognizing saturation after one more. So, with 2 successor clock zone calculations, and 6 BDD applications, the full reachable state space is already discovered. Third, $\langle x \geq 2, \emptyset\rangle$ is applied in the G_I induced form, with a third successor clock zone calculation, and the 8th BDD application, recognizing no newly reachable states. So the G_S induced form does not need to be applied. So, the underapproximating lookahead approach calculates the state set:

clock zone	BF
$0\uparrow$	$\{00\}$
$x \geq 5\uparrow$	$\{01, 10, 11\}$
$x \geq 3\uparrow$	$\{01, 10, 11\}$

This will be done with 3 successor clock zone calculations, and 8 BDD applications. This is a special case, as here $G_S(p, p)$ for all guard reset-set pairs p. As there is no other root of the G_S induced DAG, the fixed point is recognized: no more locations can be found below this root from the newly known to be reachable locations, as they would all have been found during saturation of the G_S induced transition BDDs, and all G_I induced BDDs have been applied to the prefixed point. For those pairs p where $G_S(p, p)$ does not hold, another round of applying them would be needed to see if

benchmark	prototype	runtime (s)	reachable zones	steps
short sample glitch	underapprox	13,513.40	1,452,451	10,467,243
short sample glitch	vanilla CZM	9,147.22	1,452,451	17,563,189
short real-time glitch	underapprox	memout	> 7,648,741	> 45,575,000
short real-time glitch	vanilla CZM	memout	> 7,813,125	> 43,216,000

Table 7.1: Comparison of the underapproximating lookahead approach to the vanilla CZM algorithm. Executed on a Intel Core 2 Duo CPU P8600 Processor with 2.40 GHz and 3.8 GiB RAM running Ubuntu 14.04 LTS.

something changes to recognize a fixed point.

So, in conclusion, in this example, the underapproximating lookahead terminates with 3 successor clock zone calculations and 8 BDD applications, while the CZM approach needs between 12 and 15 of each. The underapproximating lookahead approach also found all locations with just 1 successor clock zone calculation and 2 BDD applications, and the full state space with 2 successor clock zone calculations, and 6 BDD applications. The CZM approach needed between 5 an 12 of each for both these goals.

7.6 Evaluation on FlexRay Benchmark

I implemented the algorithm presented in Section 7.4.1 as a prototype in C++ and integrated it into the Synthia framework [PEM11] (a tool for verification and synthesis for timed automata maintained by Hans-Jörg Peter), together with my prototype re-implementation of the original CZM algorithm from Chapter 6 which served as a reference point. The implementation of both algorithms is identical in the identical parts of the algorithms, and differs just where the algorithms behave differently. This allows a comparison of the approaches on a level playing field.

A benchmark based on the timed automata model ‡ of the FlexRay physical layer protocol as presented in Chapter 10 was created in Uppaal and translated to Synthia's input format using a Python script. As shown in table 7.1, in the benchmark with a short sample glitch as the error model, fragmentation did not cause a problem, but the additional overhead of the underapproximating lookahead slowed the algorithm down compared to the vanilla CZM version. When the model included a third clock in the case of a short real-time glitch, the memory requirements exceeded the available 3.8 GiB, so the algorithm was interrupted before the whole state space was explored. Still, the underapproximating lookahead needed more steps to explore more than 7.6 million states than the vanilla algorithm took to explore more than 7.8 million.

These results confirm that the FlexRay models are not structured in a way that profits from the underapproximating lookahead approach: The timed constraints on transitions are often equality constraints, and generally the model does not have many transitions who's enabledness implies the enabledness of other transitions that are not already synchronizing with this transition. The total lack of fragmentation is

another indicator that not many states of the FlexRay model are explored with an underapproximated clock zone. Thus, a carefully optimized model of a real-world system that avoids unnecessary uncertainties which could give rise to more implications between enabledness conditions of transitions might not benefit from the approach presented in this chapter. For a quick exploration of a draft model with the aim of finding an error, the approach could be better suited, carefully optimized models will most likely suffer from the underapproximating lookahead approach compared to the vanilla CZM approach.

Part III

Modeling Principles

Table of Contents

Chapter 8

Selection of Theories

Contents

Abstract. The choice of theories to be used when modeling a system needs to take several factors into account: The property to be verified needs to be expressible and the system itself needs to be describable after abstraction and overapproximation have been used to make the system model simpler. In the case of verifying a correct message transfer in FlexRay, this property is best described in the model of the protocol, so expressing the reachability of the error state is sufficient. The protocol is stateful with discrete states, and the hardware behavior can be abstracted to discrete states and time. Thus *timed automata* and *timed computation tree logic* (TCTL) are selected, as supported, e.g., by Uppaal.

Formal verification needs a formal language. The formal language needs to be able to express the properties that are to be verified, and to describe the model to be verified itself. The language thus needs to be expressive enough for the verification project. However, if a language is very expressive, it can formulate problems that are hard to solve. It may also contain very complex constructs, that could be computationally expensive to treat. So the language should be as expressive as needed, but also as simple as possible.

Simple statements can be categorized in terms of the theories that contain them. In order to formulate the properties and the model, a set of theories has to be chosen, thus defining the set of statements that can be used.

8.1 Property to Check

The property that is to be verified needs to be formalized. So every aspect of it needs to be phrasable as a statement in a specific theory. For example, if a value is to be

asserted, the theory of equality will be needed, if a sum of two integers should not exceed a value, the theory of linear arithmetics would be adequate.

However, the borderline between the property to check and the model is blurred, so it is possible to shift complexity between the property and the model. Usually, models will contain a lot of complexity due to the complexity of the system, while the property will and should contain considerably less complexity. Thus, a property that takes into account a lot of details from the model is best partially formulated inside the model. Formulating a part of the property inside the model allows to describe the property using these added aspects of the model. For example, a complex error condition of a stateful model could be formulated in terms of a dedicated error state of the model, reducing the part of the property that is not part of the model to a simple statement about the reachability of the error state. Verification tools often have limited expressivity of the accepted property-language, usually much more limited than the model-language. This is a design choice in order to improve human-readability of the property and enables the use of more efficient algorithms and data-structures, thus making it mandatory to move more complex aspects of the property to the model.

8.2 Relevance of System Behaviors

The system will have aspects that cannot be described in all detail in the model because reality is neither fully describable nor fully known. Those aspects can however not be ignored altogether without an argument why they can be ignored or an explicit assumption stating that they can be ignored, but often they can be treated by abstraction. Olderog and Dierks [OD08, p. 27] argue that abstracting details of larger real-time systems is necessary anyway to reduce the size of hard-to-deal-with huge state spaces.

Abstraction can even turn models that would require hybrid variables on top of real-time into easier, pure real-time, models.[1] For example, almost all physical processes take time and are gradual. Consider a physical system transitioning from one stable state to another. To model this, if a notion of time is given, the state of the gradual process could be abstracted into the prior state, the intermediate state, and the posterior state. In the prior and posterior states, the behavior of the system will usually be easily modeled. In the intermediate state, the behavior, even if it is well defined, might be too complex to model precisely, in which case nondeterminism can be used to over-approximate the possible behaviors of the system. In turn, the time aspect might be linked to the exact state of the gradual process. If this gradual process is not modeled in detail, the time-behavior as well cannot be modeled in detail, which can usually be resolved using over-approximation of the duration of the intermediate state at the expense of the prior and/or posterior states, and nondeterminism to resolve the system behavior during this prolonged time, again resulting in an overes-

[1] Olderog and Dierks [OD08, p. 3] advocate to abstract time-dependent continuous physical variables, giving the example of abstracting physical attributes of a train like speed and position into the discrete values *far_away*, *near_by* or *crossing* with respect to a railway crossing to be modeled.

timation of the possible system behaviors. With an accompanying argument how the over-approximation affects the property, thus securing the correctness of this approach, considerable complexity can be avoided in the model.

It is imperative to abstract away aspects of the system to enable the description of the model in a reasonably expressive language using statements from reasonably hard to treat theories. Many behaviors of the system might not be relevant directly for the property that needs to be verified and could be abstracted away if their indirect relevance for the property has been resolved, e.g., by approximation and non-determinism. Selection of the modeling language should thus be done not only after enough understanding of the system is gained to model it at all, but after additional analysis on what can be abstracted away, and what price to pay for those abstractions.

8.3 Case Study: FlexRay

For the verification of the FlexRay physical layer protocol, the property to check is the correct reception of the transmitted message stream. As the reception of the message stream is part of the protocol to be verified, the verification of the stream format is naturally incorporated into the model. Correctness of reception of the message content might seem an easy concept, however, to accurately describe it is most easily done using references to protocol details, e.g., to the relative position of a given bit in the message, or the time taken to transfer a bit from sender to receiver. Thus, it is way easier to incorporate the check for message equality inside the model than leave it inside the property. If message equality was violated, the model could enter its terminal error state. Additionally, after correct reception of a message, the model should enter its terminal safe state. If this is done, the property to check boils down to a simple reachability check of the dedicated error state of the model or a check that the safe state is reached eventually.

The restricted version of *timed computation tree logic* (TCTL) admitted by Uppaal, which allows only very restricted use of quantifiers, is sufficient to express the notion of non-reachability of the error state, namely that on all executions, in all time instances, the model is not in the error state. The property that a message is eventually received can be expressed as well, namely that on all executions eventually the safe state is reached.

Note that, as checking the safety property of non-reachability of an error state is usually easier than checking the liveness property that a safe state is always eventually reached, one can also incorporate the liveness property that some message is eventually received into the model. To do so, the model is constructed such that not receiving a message (after the process of receiving a message has started) causes the model to eventually enter the error state. This is usually a given, as checking whether what is received is what is expected entails reading from the communication medium (thus receiving something, as the medium will be sampled independently of its state and the samples will be interpreted) and entering the error state if it is not what was expected, especially if it is not consistent with a message. This, in turn, allows to

check reception of a message by just adding a check of the liveness property that the reception of a message is always eventually started—which is way easier to check than that this reception always eventually finishes—to the check of the safety property that the error state is never entered. Combined, these two checks provide the property that the message will eventually be correctly received. To get rid of the liveness check that message reception always eventually starts, one could furthermore build the model in such a way that from the initial state the reception of a message will clearly always eventually be started.[2] Obvious tricks such as forcing the receiver model to enter the error state after a certain time if reception of the message has not started would introduce unnecessary complexity and reduce the model's readability, but would make this liveness check redundant as the safety check of non-reachability of the error state would cover the eventual start of the reception in addition to the reception itself.

To model the FlexRay physical layer protocol and the hardware running it, several aspects have to be covered.

FlexRay is a stateful protocol, and the FlexRay specification is given in UML, e.g., using behavior diagrams, thus automata seem to be a natural choice for modeling the protocol. Automata provide a visual representation which is easy to read alongside the UML specification. Compared to UML, automata can also be more easily automatically translated into a precise logical representation.

In order to describe the hardware, the usual boolean abstraction is not sufficient for all aspects, as the hardware is distributed, but not synchronized on the lowest level. Thus, whenever one component communicates with another, non-synchronized one, voltages and durations of voltage changes can no longer be safely ignored. However, using over-approximation and nondeterminism, the continuous voltage can be abstracted to discrete levels. Durations on the other hand are also continuous and have to be treated, thus *timed automata* are a good choice. UPPAAL's modeling language is *extended timed automata*, which includes timed automata.

To model the error resilience of the FlexRay physical layer protocol, an error model is needed as part of the model. Similarly to the hardware model, the errors can have voltage and duration aspects which can be resolved using the same methods as used in the hardware model. The error model can thus be integrated into the hardware model which is modeled as a network of timed automata.

[2] However, while the FlexRay models presented in this work do all eventually start the process of receiving of a message, this is not immediately obvious at first glance, so the eventual start of reception was additionally easily and quickly verified using UPPAAL.

Chapter 9

Tailoring the Model to the Data-Structures and Algorithms

Contents

Abstract. The data-structures used to store the states of the model have a strong influence on the memory requirement of verification. During modeling, the memory requirements can be reduced: For timed automata, the size of the *difference bound matrices* (DBMs) scales quadratically with the number of clocks, so using the same clock for several different but interdependent purposes helps. Tight bounds on discrete variables reduce the discrete state space.

The data-structures to be used during the verification have to be kept in mind when modeling, as they will be used to store the states of the model. This will have a dominant influence on the memory requirements of any automated verification effort. However, the model can and should be tailored such as to reduce the amount of memory required to store a state of the model in a given data-structure.

As timed automata are used in this work to model the FlexRay physical layer protocol, the hardware, and the errors, the notion of real-valued continuous time will be provided by *clocks*. The notion of internal discrete state will be provided by the location of the automata. To make things more readable, the notion of bounded variables is introduced, which, though borrowed from extended timed automata, is easy to translate to timed automata and adds useful syntactic sugar making the models easier to read.

9.1 Modeling Time

A single explicit state of the model's clocks would be a vector giving the concrete real-value of each clock. However, using these directly is clearly infeasible, as there are uncountably many valuations in every non trivial execution of a timed automaton.

As timed automata only refer to time using comparisons to integers, it is possible to collect valuations in a symbolic representation which represents a set of (usually uncountably many) valuations. The effects of changing the state of the model are then taken into account as additional constraints on this set or by changing all constraints in a consistent way. The data-structure of choice for handling clocks, which is also used by UPPAAL, is a *difference bound matrix* (DBM), as described in Section 3.4 on page 35.

DBMs also offer certain efficient operations, like letting time evolve on all clocks with the same speed, or resetting a set of clocks to zero, or intersecting with a constraint. Thus, clocks in the model should only be either reset to zero, not touched, or be compared to a constraint—and should let time pass at the uniform constant rate of 1.

A DBM stores upper bounds of the differences between the various clocks of the model, and between them and zero. Assuming n clocks, this allows to store any convex zone in the n-dimensional space of clock values in a single $(n+1) * (n+1)$-matrix. Non-convex zones would be split into convex parts and stored as a set of convex zones. Thus, the memory requirement for storing convex clock zones grows quadratically with the number of clocks. Moreover, the number of different values of this data-structure that are encountered during a run of a model checker typically grows exponentially with the number of clocks. According to Olderog and Dierks [OD08, p. 18], the exponential complexity in the number of clocks limits the use of timed automata for the verification of larger real-time systems.

Keeping the number of clocks low, as demonstrated in Section 10.5, is thus advisable, if the price to pay for it in terms of discrete complexity is not to high (see Section 11.2). This can be achieved by finding the source of an action in the model. The FlexRay model can logically be divided into four parts: the sender, the receiver, the bus and the glitches.

If the sender triggers certain behaviors, their timing could possibly be modeled using the same clock if they are dependent on each other or on the same root cause. As the sender's behavior is dependent on the ticks of the local oscillator, which, being imprecise, is a source of real-timed behavior anyway, the timing of the behavior of the sender can be modeled in terms of one clock.

As the receiver is not synchronized to the sender, its behaviors cannot directly depend on the clock used to model behavior of the sender. However, behavior in the receiver being dependent on the receivers local oscillator, its time related behavior can be modeled in terms of one clock as well.

Considering the time-behavior of the bus, as the bus is driven by the sender, its behavior can be modeled using the clock used to model the sender as well. However, to model glitches, which are independent of the sender or receiver, either a third clock needs to be introduced, or the glitch can be abstracted to its effects on the receiver, which again would allow to use the clock used in modeling the receiver. As the glitch does only affect the receiver, this abstraction is sound: instead of modeling a glitch, the glitch affected samples are modeled. But as glitches can only occur so often without destroying communication, some notion of time between glitches is needed as well, requiring the introduction of counters if no extra clock is introduced for the glitches,

which, depending on the bounds of the counter variable, can increase the discrete state space significantly. This trade-off does make the introduction of a third clock for modeling glitches advisable in some cases in order to reduce memory consumption and runtime.

9.2 Modeling Discrete State

An explicit discrete state of a system is the valuation of all its variables. It is desirable to have a finite number of discrete states, as it allows to enumerate them during verification. Nevertheless, enumerating all the discrete states should be avoided by using a symbolic representation for the discrete states as well, as even a finite number of discrete states may be to many to enumerate them within a reasonable amount of time, or remember them explicitly using only a reasonable amount of memory.

To keep the number of discrete states finite, the automaton representing the model may only have a finite number of locations, and all discrete variables should be bounded. Not least to increase the readability of the model, the model can be represented by a network of communicating automata, whose product can be computed on the fly in the exploration algorithm, avoiding the work of constructing the complete product automaton before the exploration can start. The locations can be represented by bounded variables as well, as each finite component automaton's locations can be considered to be the possible values of some enumerable data type variable.

If all variables are bounded and the number of locations is finite, they can be represented by boolean variables such that a valuation of the variables and the current location in each component can be represented by a bitvector.[1] Thus, *reduced ordered binary decision diagrams* (BDDs), as discussed in Chapter 4, can be used to represent the state space as well as the transition relation, which allows to apply a BDD based symbolic exploration algorithm like the one presented in Chapter 6.

If possible, bounded variables with a big range should be avoided, as they can blow up the state space considerably.

[1] BDD libraries like CUDD [Som09] bring their own variable reordering heuristics, so determining the order of the boolean variables can be delegated to the libraries.

Chapter 10

Modeling FlexRay[1]

Contents

[1] This chapter contains parts already published in [GEFP12b], [GEFP12a], and [GEFP10].

Abstract. The modeling of an industrially used protocol like FlexRay was an endeavor posing considerable challenges and took a lot of time. The process did not only enable the verification of the resilience of the FlexRay physical layer protocol, but also involved a lot of learning. The lessons learned about the modeling of such a physical layer protocol provide an excellent example to illustrate useful modeling patterns and principles. This chapter presents several aspects of the FlexRay physical layer protocol and its environment and demonstrates the modeling process through several stages, ranging from naive first attempts to the sophisticated optimized models finally used for the verification effort. It elaborates on the underlying principles and patterns employed during the evolution of the models, most importantly abstraction and nondeterminism. The FlexRay physical layer protocol specification makes some assumptions on the underlying hardware. To enable the investigation of these assumptions, the final models allow to evaluate the effect of changes to the physical setting to the fault tolerance of the protocol. These models thus need to be precise enough to allow to describe the intricate real-time interplay between the hardware and the protocol, while also being "small" enough to allow for automatic verification.

Modeling the FlexRay physical layer protocol requires to also model its underlying hardware environment. Being designed for an automotive context, FlexRay makes several assumptions on the hardware it will be deployed on. Not all of these are necessarily true in other contexts. For example, there is interest in FlexRay in the aeronautics community, due to the low level error correction and recognition and the predictable timing properties of FlexRay. And upgrading existing physical layers, for example from CAN bus systems, with FlexRay is attractive, as inexpensive FlexRay hardware is available commercially of the shelf [HR09, PH08]. For X-by-wire, strict requirements regarding the quality of service of the bus protocol have to be met. Thus, investigating how changes to the physical layer affect the protocol is necessary when employing the protocol in a new environment. For example, as a requirement typically met by cars but not by planes, FlexRay assumes a maximal harness length of at most 24 meters [HR09, Fle06a]. To verify that the protocol still has the same fault tolerance in an environment that does not fulfill all assumptions underlying the protocol's specification, the protocol model together with an adjusted hardware model taking the changed assumptions into account will have to be verified again. A parametric hardware model can be quickly adapted to a changed hardware environment by changing the values of some parameters. This adapted model can then be used for the automatic verification of protocol properties under the changed assumptions on the hardware. This approach requires the extension of the timed automaton model by Alur and Dill (see Chapter 3) through the introduction of parameters.

10.1 Parametric Timed Automata Models

In the following, the description of timed automata will be syntactically extended by bounded *integer variables* and *arrays* of integers with a bounded domain. They will be referenced in *integer expressions*, which can use basic arithmetic to arrive at integer values. Calls to pseudo-code functions will also be used for better readability in some

descriptions of early model versions. Boolean expressions of the form $a \prec b$, where a and b are integer variables and $\prec \in \{<, \leq, =, \geq, >\}$ will also be used. For updating integer variables, edges can be extended by *update expressions* of the form $a := x, \ \ldots \ , c := z$ assigning the values of the integer expressions $x, \ \ldots \ , z$ to the integer variables or arrays $a, \ \ldots \ , c$. *Paramters* are integer variables which are treated like constants, and do not appear on the left hand side of an update expression. This yields a parametric model that has to be instantiated to a concrete one by giving all the parameters an integer value before the start of the verification procedure.

10.2 Modeling Principles

Model checking a scenario as complex as a FlexRay physical layer protocol message transfer, and using an off-the-shelf model checker to do so, requires a carefully crafted model. Unlike testing, model checking has to account for every possible behavior of a system. This complete coverage is computationally very expensive: model checking a network of timed automata takes exponential time in the size of the network. To keep this complexity as low as possible, the presented models aim to use clocks and data as economically as possible. Several *modeling principles* were employed. In the following sections, these principles will be introduced in detail using the models as an example. Each principle will also be highlighted in a box together with a short summary description.

10.3 Structure of the Model of the FlexRay Physical Layer Protocol

The scenario of a message transfer in the FlexRay physical layer protocol is quite complex. To ease understanding and thus avoid modeling errors, a monolithic modeling approach is not advisable. Instead, separating the scenario into parts and modeling these individually leads to a collection of much more desirable small models, which are individually easier to understand. Of course, these models need to communicate with each other in order to jointly model the whole scenario, but if the interfaces between them are kept simple enough, it seems a less daunting task to understand several individual components of lower complexity and their interplay, than understanding a huge monolithic model.

Moreover, if the model needs to be changed, it is usually easier to change one component while keeping its interface intact than it is to change a huge model.

Thus, the model will be presented as a network of parametric timed automata.

Separate into Components

For a separation of concerns, better understandability and easier modification, model the scenario as a network of smaller components.

The FlexRay model's components can roughly be divided into the parts pertaining to the protocol, and the parts describing the hardware and the error model, as shown in Figure 10.1.

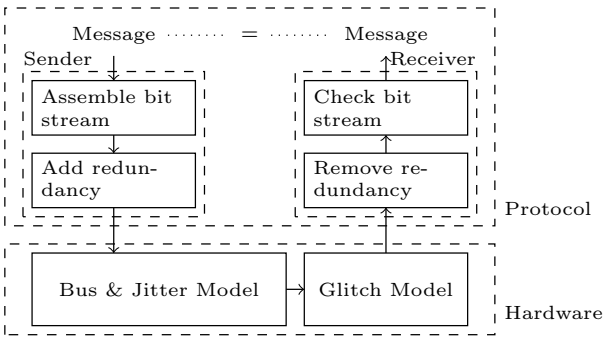


Figure 10.1: The structure of the model. The arrows indicate the flow of information.[2]

The hardware model can be separated in two main parts, on the one hand the actual hardware of the bus and the clocks which introduce jitter, and on the other hand the glitch model that represents interference on the bus. The protocol also has two main parts, the sender 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 received message 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. 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. As the receiver and the sender are implemented as hardware, all their steps are triggered by their respective oscillators (sender clock, receiver clock).

As shown in Figure 10.2, the oscillator components can be modeled as individual automata that synchronize with the respective sender or receiver components. The behavior of an individual oscillator is easy to grasp in such small model components: Each clock cycle takes between `CYCLE_MAX` and `CYCLE_MIN` time units and its end is marked with a tick, which is either for the sender components ($tick_{sd}$) or for the receiver components ($tick_{rc}$). It can also quickly be seen that the two oscillators are independent of each other, their only relation being the same parameters configuring a minimal and a maximal duration of a clock cycle.

The difference between the maximal or minimal duration of a clock cycle depends on the precision of the hardware, and the ideal length of a clock cycle is derived from the frequency of the oscillator and thus also depends on the hardware used. As the

[2]This figure is based on a figure already published in [Ger10, Figure 6.1], [GEFP10, Fig. 1] and [GEFP12b, Figure 8]. It has already been published in [GEFP12a, Figure 1].

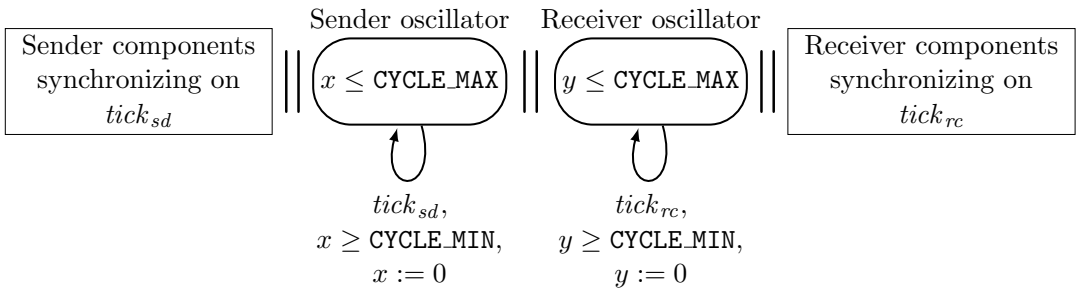


Figure 10.2: Network of parametric timed automata showing the oscillators for sender and receiver.[3]

behavior of the hardware is described in terms of real-time, the model of the hardware is best 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. In turn, this means that certain hardware model parameters can be explored and their effect on the protocol's properties be analyzed, yielding reliable requirements for the hardware.

Parameterized Model

For quick adoption to specific assumptions on the underlying hardware, keep the hardware model parameterized. For quick adoption of protocol changes, keep the protocol model parameterized. A parameterized model allows to easily explore the effect of changing parameters.

10.4 Hardware Environment and Possible Errors

In a FlexRay network, each controller has its own local oscillator supplying the clock signal to its local circuits. These controllers communicate via the shared *bus*. However, on the level of the physical layer protocol, this communication is asynchronous, as each controller only uses its own local oscillator. So we cannot simply ignore the continuous behavior of the hardware at the interface between the components, which would allow to work in an abstracted digital model—we have to face the underlying analogous nature of microelectronics. This becomes visible in the behavior of the bus when driven to a new value by the sender and the behavior of the *register* used to sample from the bus at the receivers side. We can avoid having to deal with this in several components by modeling the behavior of the bus as the value at the input of the sampling register, thus isolating the issue into one component automaton.

[3]This figure has already been published in [GEFP12b, Figure 17].

10.4.1 Ignore Constant Delays in One-way Communication

But how to model the delays to the signal at the receivers sampling register's input introduced by the bus or the circuit design of the sender? Well, they do not necessarily need to be modeled: As visible in Figure 10.1, information flows strictly in one direction: From the sender to the receiver. As there is no feedback from the receiver to the sender on the bus, all constant delays to the flow of the information can be ignored, as they will be invisible to the receiver anyway.

Ignore Constant Delays in one-way Communication

All constant delays, even inside a component, can be ignored (or chosen arbitrarily) if information flows strictly in one direction.

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: from the bus sampler to voting, from voting to strobing and from strobing to the receiver control. The only feedback inside the receiving 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 formated message stream bit before it is used, so a correct implementation of the receiving controller can easily make sure that the signal arrives in time, as can a carefully designed model of the receiving controller's components, allowing to ignore the constant delays inside the receiver as well.

10.4.2 A Register with Asynchronous Input

Consider the hardware scenario described in Figure 10.3, where 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 [GEFP12a, GEFP10, BBG+05, Sch06, Sch07, KP07, ABK08b], as proposed by [Män98], the output of register `Rx` is forwarded 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 constant delays can be ignored as explained in Section 10.4.1, `Rxx` does not have to be modeled: instead, an unstable register `Rx` can immediately be resolved nondeterministically to 1 or 0.

Following the setting of [GEFP12a, GEFP10, BBG+05, Sch06, Sch07, KP07, ABK08b], a *register semantics* is assumed 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 can be used to describe most implementations, just the parameters that describe them have

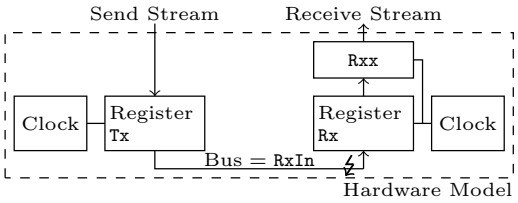


Figure 10.3: Overview of the hardware sub-architecture.[4]

to be changed accordingly. Before diving into the the actual transmission of bit values via the bus, first consider the following general description of the low-level timing behavior of registers.

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

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

If a register is not enabled, its output will stay the same. In the following, it is assumed that the register is enabled. 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, as shown in Figure 10.4. However, there is a certain range of voltage levels between the two thresholds that cannot be interpreted as any Boolean value.

Figure 10.5 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.

The unknown value is assumed to be 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 [KP95, Sect. 5.2], all local inputs can be assumed to be stable.

[4]This figure has already been published in [Ger10, Figure 5.5], [GEFP10, Fig. 13], and [GEFP12a, Figure 12].

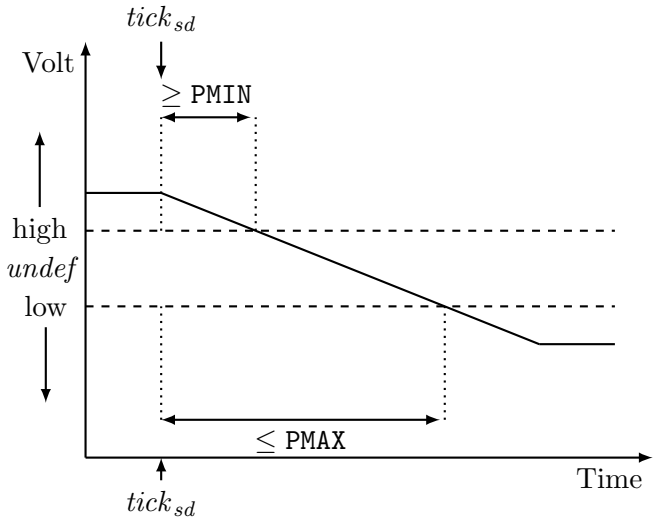


Figure 10.4: Characteristic timing diagram of a transition between voltage levels.[5]

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

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

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

Note that this is a case of continuous behavior that can be abstracted away using continuous time and nondeterminism, as described in Section 8.2: In a voltage change at the input of a register, there is a stable anterior state, a stable posterior state, and a passing intermediate state between the thresholds were behavior is not well defined.

[5]This figure has already been published in [GEFP12b, Figure 18].

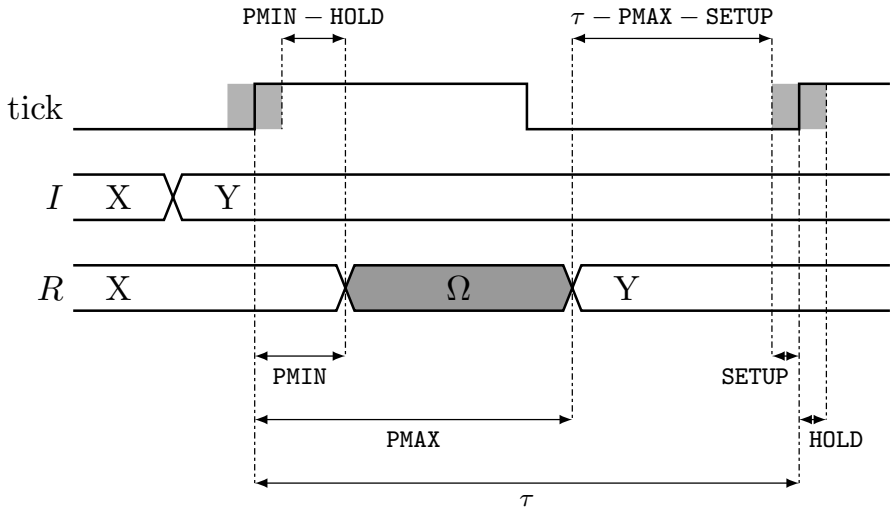


Figure 10.5: Value change scenario of a register R.[6]

Model continuous aspects other than time using nondeterminism and the timing of their transitions between stable states

If a continuous aspect of the scenario has several stable states with a well defined effect on the digital world, model this aspect using continuous time to model the transitions between its stable states. Use nondeterminism to over-approximate effects that are not well defined, e.g., during the unstable transition-states.

As we assume metastability (which could occur in the $X = \Omega$ case) to be resolved nondeterministically by the `Rxx` register, a violation of stability during setup and hold times can be modeled by nondeterministically resolving the register to either 1 or 0. This also accounts for cases in which a transition between voltage levels is faster than the worst case assumed in the definition: As the register's content can be nondeterministically be resolved to the new (or the old) value if its content is read during the unstable period, this includes all behaviors where the register's output already has the new value (or still has the old value) even if the register is not known to be stable at the time.

10.4.3 Error Types

Jitter. Errors introduced through the violation of setup or hold times due to the non-synchronized communication are called *jitter*. Nondeterministically resolving unstable register values to a stable 0 or 1 (as metastability has been excluded) models this type

[6] This figure is based on a figure already published in [Ger10, Figure 5.6], which, in turn, was based on a figure already published in [Ger05, Figure 10], which, in turn again, was based on [BBG+05, Figure 1]. It has already been published in [GEFP10, Fig. 15] and [GEFP12a, Figure 14].

of error which can have either have the effect of receiving a new value a little to early, or the old value for a little too long. *Jitter* will be treated in Section 10.5.

Glitches. However, disturbances of the signal on the bus are another possible error scenario, called *glitches* [Fle05, Section 3.2.2][Fle06b, Section 12.2][Fle06a, Section 3.6.3]. A sample taken from the bus might have been replaced by an arbitrary value. This error scenario can be modeled by assuming an unstable bus: either the bus can be assumed to be unstable when a sample is taken from it, leading to nondeterministically sampling either 0 or 1 independent of the value that was send (called *sample glitch* in this work), or the bus can be set to an unstable value for a certain time period (called *real-time glitch* in this work), resulting in nondeterministically sampling either 0 or 1 if a sample is taken from the bus

(a) during that time period,

(b) closer than the hold time before that time period, or

(c) closer than the setup time after that time period.

If too many glitches occur, the message might be compromised. However, the FlexRay physical layer protocol compensates for infrequent glitches. The two glitch modeling paradigms are explored in Section 10.6:

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

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

10.5 Modeling the Bus and the Sampling from the Bus

Taking the background described in Section 10.4 into account, let's consider the modeling of a propagation of signals along the bus. A timed automaton network for this scenario is shown in Figure 10.6. The model starts in a state where the bus value is stable. When a clock tick occurs (driven by the local oscillator of the sender), the sender can begin to drive a new value on the bus. The receiver does not immediately see the new value. Only after `PMIN` time units, the sender has driven the voltage on the bus beyond the threshold for recognizing the old bus value. After additional $\texttt{PNEW} = \texttt{PMAX} - \texttt{PMIN}$ time units, the value of the bus at the receiver's side has been driven beyond the threshold for the new bus value, as shown in Figure 10.4. In the model, this is implemented by measuring three time spans: the length of the clock cycle, the time to drive the bus beyond the first threshold, and the time to drive it beyond the second threshold. A clock cycle does not always have the same length (there are no perfect oscillators in practice), but its length is always in between `CYCLE_MIN` and `CYCLE_MAX`. Naively, a separate clock is used for measuring each of these time spans.

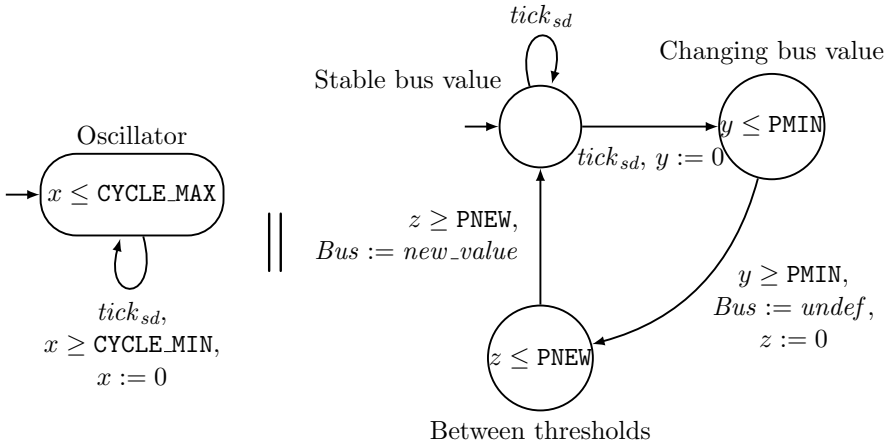


Figure 10.6: Model of the sender's oscillator and the bus, synchronizing on action $tick_{sd}$. One clock is used for the oscillator, one clock for the time to reach the first threshold, and another one for the time to reach the second threshold.[7]

Section 9.1 advocates to reduce the number of clocks by exploiting dependencies between the clocks.

Exploit (Causal) Dependencies Between Clocks

If some event triggers another event, then the clock measuring the timing of the first event can (often) also be used to measure the timing of the second event. Find those dependencies by looking at the source of real-time behavior and try to partition the model into parts dependent on the same source.

In FlexRay, such dependencies start with the two oscillators in the sender and receiver, which each cause various dependent events. As a result, the *entire* physical layer protocol and the hardware can be modeled with just *two* clocks.[8]

An analysis of the model shown in Figure 10.6 reveals dependencies between the clocks: The events on the bus are triggered by the sender. The sender's actions are triggered by a tick from its oscillator. Thus, the events on the bus are triggered by a tick from the sender's oscillator. This allows to describe events on the bus using the clock for generating the ticks of the sender's oscillator. 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, this limits the model 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

[7] This figure is based on a figure already published in [GEFP12b, Figure 9].
[8] This does not include the glitch model, which may need separate clocks.

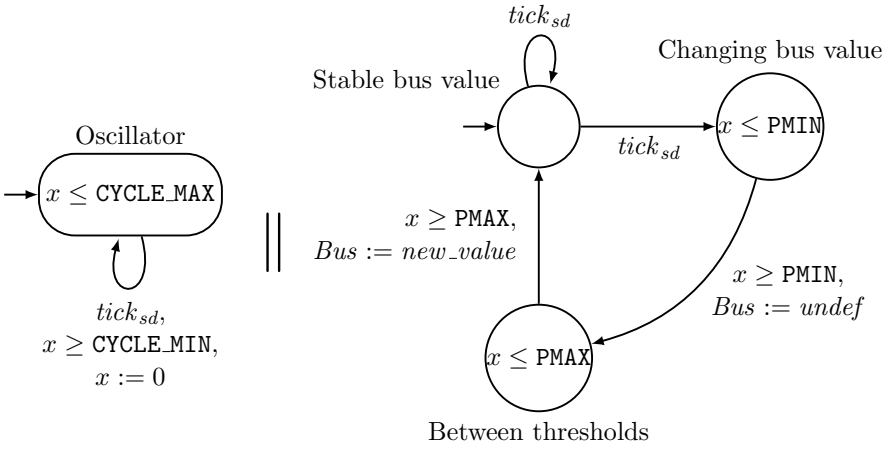


Figure 10.7: Optimized model of the sender's oscillator and the bus, synchronizing on action $tick_{sd}$. Here, just the clock needed to generate the $tick_{sd}$ action of the sender's oscillator is used, under the assumption that PMIN $\leq$ PMAX $\leq$ CYCLE_MIN holds.[10]

smaller than the minimal duration of a clock cycle. As FlexRay operates at 80 MHz,[9] this requirement is easily fulfilled by standard hardware. Observe that it is possible to deduce the values of y and z from the clock value of x. In the model of Figure 10.6, x and y always have the same value, and z is equal to $x +$ PMIN in the "Between thresholds" location, which is the only location in which the value of z is used. Thus, the clocks y and z can be removed by replacing any reference to y by x and replacing any reference to z by $x +$ PMIN. Exploiting the equality PNEW $=$ PMAX $-$ PMIN results in the simplified model shown in Figure 10.7.

So far, the models shown have been toy examples. In the following, automata from models actually used for verifying the FlexRay physical layer protocol will be shown as well, using the syntax described in Section 3.2. There are two separate models:

- an Uppaal specific early model also presented in [GEFP10] that will be marked with † and uses syntax specific to Uppaal (like committed locations or broadcast synchronization), and
- a refined later evolution of the model, also presented in [GEFP12a], not using Uppaal specific syntax, and marked with ‡.

The bus has two stable states, either low (0) or high (1). Figure 10.8 shows the automaton from model † modeling the transmission of a bit value according to the register semantics defined in Section 10.4.2. Recall the structure of the hardware sub-architecture shown in Figure 10.1. The model represents the sender's register Tx's

[9] Here, a configuration of FlexRay with a 10 Mbit/s data rate is assumed. Note that parameters for slower bit rates of either 2.5 Mbit/s or 5 Mbit/s are also supplied in the FlexRay specification, see [Fle05, Appendix B.1].

[10] This figure is based on a figure already published in [GEFP12b, Figure 10].

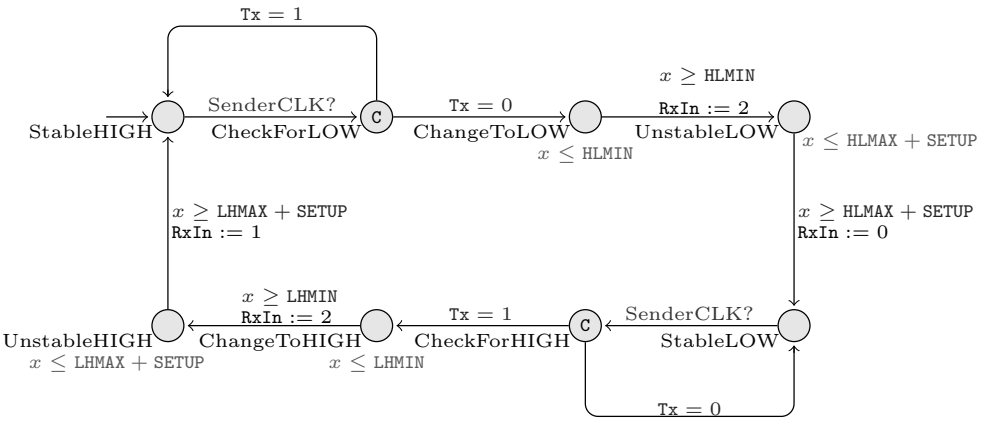


Figure 10.8: Model of the bus.†[11]

content by a variable `Tx`, and the receiver's register `Rx`'s input (which also represents the bus' content) by a variable `RxIn`. As the bus value is high whenever it is idle [Fle05, Section 3.2.4], `RxIn` is initialized with 1 and the automaton starts with the bus in a stable high state.

At every tick of the sender's clock (SenderCLK), 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. This change will be delayed: Initially, the old value is still preserved, then the value on the bus is undefined, and then, finally, the new stable value is reached. Here, an undefined bus content is represented by a value of 2 for `RxIn`. The parameters `HLMIN`, `HLMAX`, `LHMIN`, and `LHMAX` are used to model the delays induced by the hardware: As a conservative approximation, it is assumed that

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

Note that the possible violation of setup times is modeled by extending the unstable period of the bus by the setup time. Thus, if the receiver samples from the bus less than `SETUP` time units after the bus should have reached a stable value, the violation of the setup time will be modeled by sampling an unstable value.

The violation of hold times is checked when the receiver tries to sample at a tick of its local oscillator: Figure 10.9 shows an automaton from model † 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 (ReceiverCLK), `Rx` is updated either

(i) nondeterministically with 1 or 0 if `Rx`'s input `RxIn` changes (which is checked with the help of variable `OldRxIn`) or is undefined (2), or

[11] This figure is based on a figure already published in [Ger10, Figure 6.4]. It has already been published in [GEFP10, Fig. 16].

(ii) with RxIn otherwise.

For the sake of simplicity, the model nondeterministically resolves an unstable or undefined value, which is represented by a value of 2, either to logical 1 or to logical 0. This value is propagated through the consecutive register Rxx.

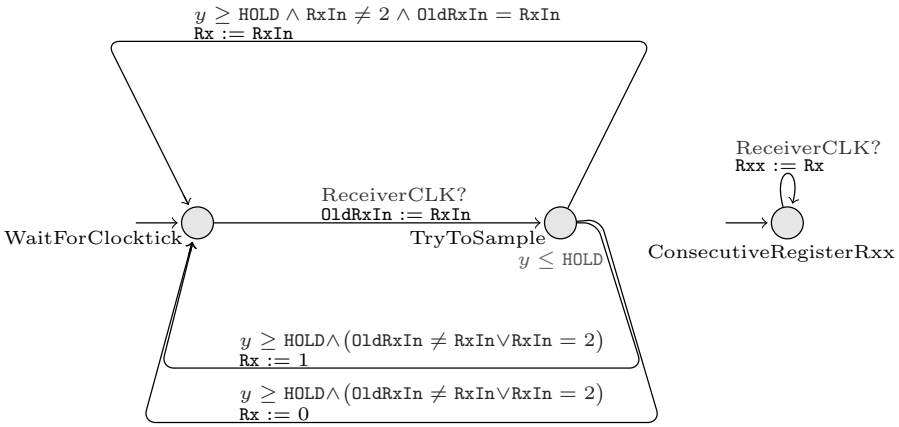


Figure 10.9: Model of the sampling process without glitches.†[12]

Note that the trick of extending the unstable interval of the bus by adding the setup time does not only make it easier to check at the time of a receivers clock tick for a violation of the setup time in the immediate past. It also shows a way of making the model simpler: In essence, the automata shown in Figures 10.8 and 10.9 check for the intersection of the interval were the bus is unstable with the interval were the receiver needs a stable bus.

Check Interval Intersection Using Only One Interval

Check if one of the two intervals to be intersected can be extended by the size of the other interval in the model, removing the need to actually check an interval intersection, such that an interval membership test for one point will suffice.

As the constant delay of the bus is irrelevant as there is no feedback to the sender, the delay can be ignored as explained in Section 10.4.1—but it can also be set to any arbitrary value 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 sender's clock. Here, WIREDELAY $=$ HOLD $-$ PMIN is chosen.

Remember that the register requires a stable input in the sampling interval $[T_r - \texttt{SETUP}, T_r + \texttt{HOLD}]$ 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 has change its value and thus will be unstable at the receiver's side during

[12]This figure is based on a figure already published in [GEFP10, Fig. 17], which, in turn, was based on figures already published in [Ger10, Figures 6.5 and 6.6].

the interval $[T_s + \texttt{WIREDELAY} + \texttt{PMIN}, T_s + \texttt{WIREDELAY} + \texttt{PMAX}]$. Thus, a stable sampling requires:

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

and also:

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

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

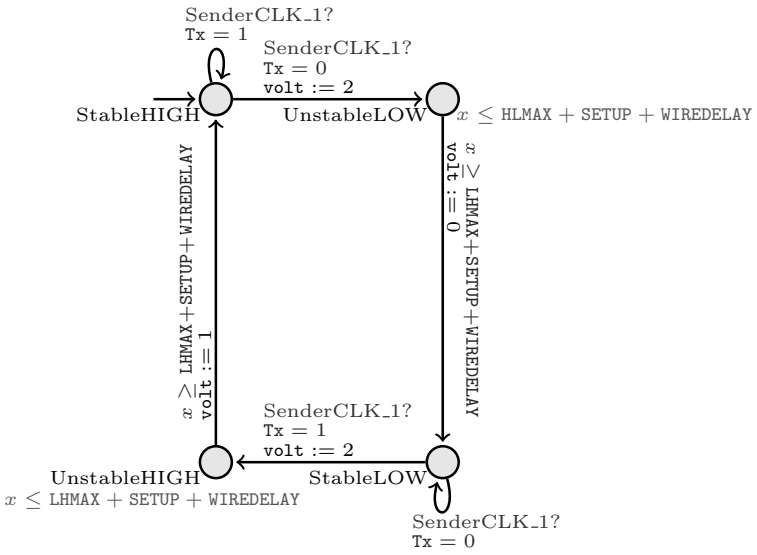


Figure 10.10: Simple model of the bus.‡[13]

In Figure 10.10, showing the bus as modeled in model ‡, `Tx` is checked to see whether the bus should change its value. In case of a change, the variable `volt` is set to the unstable value 2 during the extended interval, and to 0 (or 1) outside the interval, depending on whether the bus should then be in a stable low (or high) state.

This variable `volt` could then be used directly to determine whether the receiver's clock tick occurred during the interval by the automaton from Figure 10.11, showing the receiver's sampling from the bus in model ‡. At every tick of the receiver's clock (ReceiverCLK), 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 the analysis assumes an unreliable physical layer, `volt` can be used by a glitch model to set the value of `RxIn`.

[13]This figure has already been published in [GEFP12a, Figure 15].

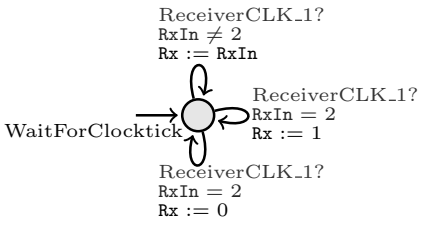


Figure 10.11: Simple model of the sampling process.‡[14]

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 explained in Section 10.4.2. Moreover, as constant delays can be ignored in the absence of feedback, as detailed in Section 10.4.1, model ‡ does not model `Rxx` at all, exposing `Rx` to the rest of the receiver in its stead.

10.6 Glitches

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, they should only be analyzed 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, two approaches are described in the following: either

(a) give the maximal duration of a glitch and the minimum glitch free time after a glitch, or

(b) consider the number of received samples that can be compromised by a glitch in a certain sequence of consecutive samples.

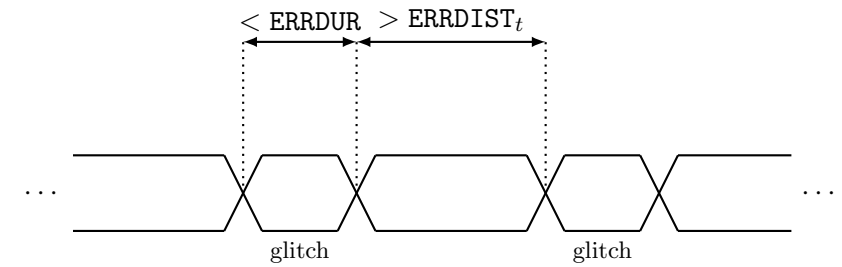
The resulting glitch patterns of the two approaches are contrasted in Figure 10.12.

Approach (a), *real-time glitches*, introduces yet another source of real-time behavior, and is well suited to describe glitches.

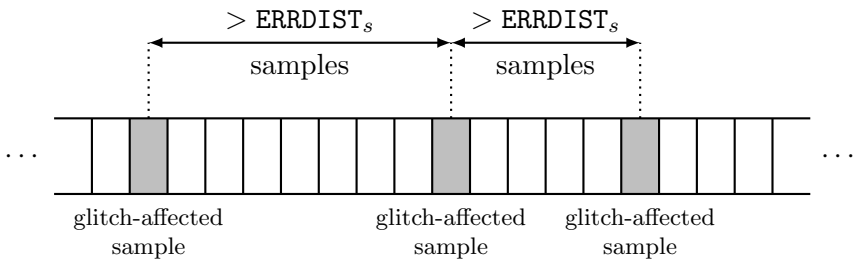
Approach (b), *sample glitches*, adds only discrete behavior, and is well suited to describe the *effects* of glitches on the protocol.

Both approaches will be explored separately in the following, but the choice only affects the glitch model, and not the model of the protocol itself. In model †, which uses only sample glitches, the glitch model is co-located with the model of sampling

[14]This figure has already been published in [GEFP12a, Figure 16].



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



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

Figure 10.12: Real-time vs. sample glitch patterns.[15]

from the bus. In model ‡, the glitch model is independent of the rest of the model, so also its hardware model is unaffected by the choice of glitch model.

10.6.1 Sample Glitches

To avoid handling more continuous time than absolutely necessary, a discrete way to model glitches is to abstract away the disturbance on the bus caused by a glitch and just model the glitch in terms of affected samples, as done in [GEFP10].

More than 2 glitch-affected samples in a voting window of size 5 are not interesting (they positively can change the (possibly strobed) voted value), so there are only two interesting scenarios:

(1) one glitch-affected sample, or

(2) two glitch-affected samples,

both in some sequence of consecutive samples, respectively. Case (2) can be divided into sub-scenarios for a more detailed analysis:

(i) two glitch-affected samples next to each other, or

(ii) two independent glitch-affected samples.

[15] This figure has already been published in [GEFP12b, Figure 19].

Lets consider the model for one glitch-affected sample co-located with the model of sampling from the bus in model †, as shown in Figure 10.13. Note that the automata shown are meant to replace the ones from the glitch free scenario shown in Figure 10.9.

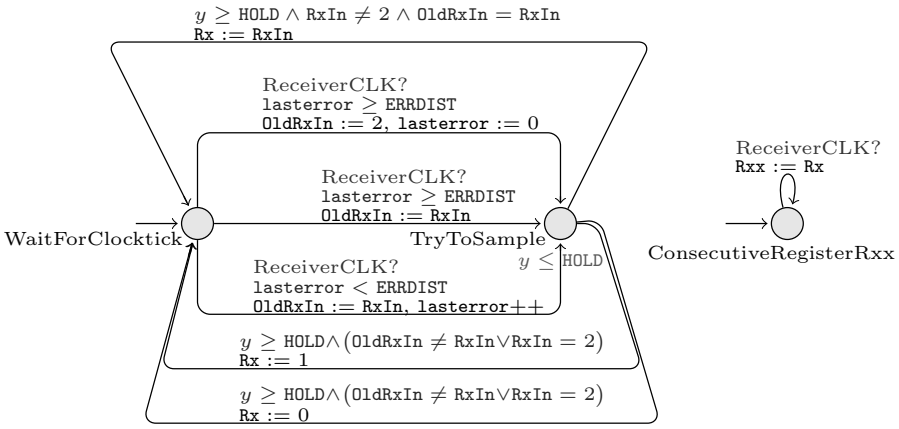


Figure 10.13: Model of the sampling process with up to one glitch-affected sample in every sequence of ERRDIST + 1 consecutive samples.†[16]

For modeling glitches, a variable lasterror is introduced that counts the number of samples without a glitch. To avoid unnecessary complexity, lasterror is bounded by ERRDIST. All glitch free consecutive sample sequences longer than ERRDIST are represented by a lasterror value of ERRDIST.[17] Whenever lasterror $\geq$ ERRDIST, the sampling process nondeterministically decides whether the current sample is affected by a glitch. A bit flip is modeled by assigning a nondeterministic value to the bit, which is achieved by re-purposing OldRxIn to force an unstable or undefined value. For the sake of simplicity, the model nondeterministically resolves an unstable or undefined value, which is represented by a value of 2, either to logical 1 or to logical 0. As this includes all cases in which the sample was resolved to the value it would not have had in the glitch free case, this does not affect a reachability analysis.

For a cleaner separation of concerns, model ‡ uses a separate automaton for its glitch model. Located between the bus model from Figure 10.10 and the model of sampling from the bus from Figure 10.11, this automaton describes how the value of the bus relates to the input of the register sampling from the bus, i.e., how the variable volt relates to the variable RxIn.

If just one sample can be glitch-affected in a sequence of consecutive samples, this automaton looks like the one shown in Figure 10.14, which nondeterministically chooses some sample to be glitch-affected and then lets the next ERRDIST$_s$ samples not be affected by a glitch. The not glitch-affected samples are counted using the variable

[16] This figure is based on figures already published in [Ger10, Figures 6.5 and 6.6]. It has already been published in [GEFP10, Fig. 17].

[17] Note that a large value for ERRDIST can lead to a considerable state space explosion.

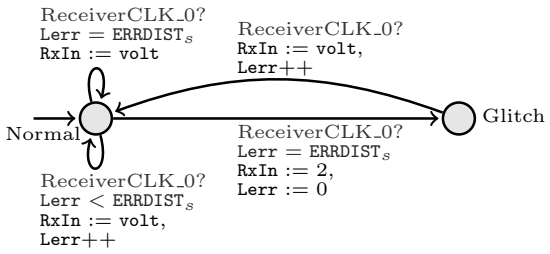


Figure 10.14: Model of sample glitches: at most 1 glitch-affected sample in $\texttt{ERRDIST}_s+1$ consecutive samples.‡[18]

`Lerr` which is used to make sure that a glitch can only occur if the last $\texttt{ERRDIST}_s$ samples were not glitch-affected. In case of a glitch affected sample, `RxIn` is set to an unstable value to be nondeterministically resolved as described above. If the sample is not affected by a glitch, `volt` is forwarded to `RxIn`. The automaton starts in the case where there have not been glitches before, so `Lerr` is initialized with $\texttt{ERRDIST}_s$.

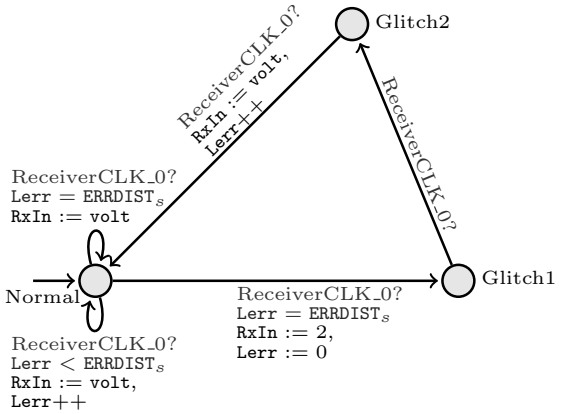


Figure 10.15: Model of sample glitches: at most 2 glitch-affected samples, next to each other, in every sequence of $\texttt{ERRDIST}_s + 2$ consecutive samples.‡[19]

The automaton from Figure 10.14 can be extended as shown in Figure 10.15 to model two glitch-affected samples next to each other where the next $\texttt{ERRDIST}_s$ samples will not be affected by a glitch.

[18] This figure has already been published in [GEFP12a, Figure 19].
[19] This figure has already been published in [GEFP12a, Figure 20].

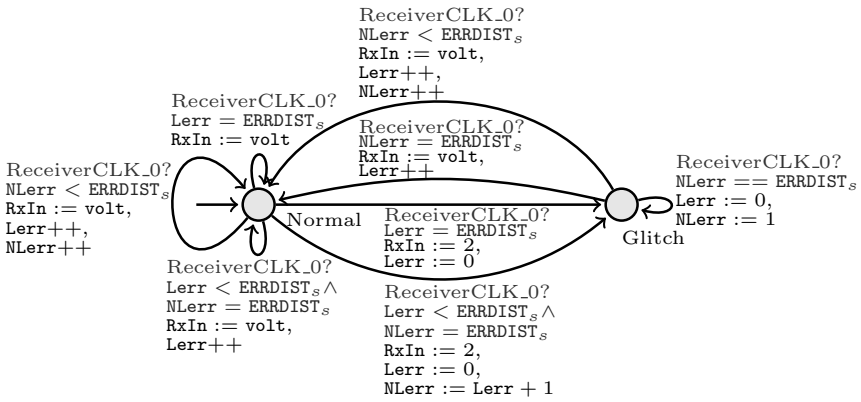


Figure 10.16: Model of sample glitches: at most 2 glitch-affected samples in $\texttt{ERRDIST}_s + 1$ consecutive samples.‡[20]

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.

Replacing the automaton from Figure 10.14 or Figure 10.15 with the automaton shown in Figure 10.16 allows to check for two independent glitches in a consecutive sequence of samples. 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`.

10.6.2 Real-Time Glitches

To reduce the discrete complexity of the model, glitches can also be described in terms of continuous time. While a FlexRay model with sample glitches only needs two clocks, a third clock, a, is introduced to measure the duration of a glitch in case of real-time glitches. Real-time glitches are only used with model ‡ here, as model † does not have a dedicated glitch model automaton that could easily be replaced with an automaton having extra real-time behavior. The versatility of the component-wise modeling approach, advocated in Section 10.3 (and used more thoroughly in model ‡ by having a separate glitch model), makes it easier to experiment with alternative glitch models.

If a glitch occurs, the bus will be unstable for `ERRDUR` time units, as described in Figure 10.12(a). Sampling a value from the bus will be unaffected by the glitch if the sampling occurs at least $\texttt{ERRDUR} + \texttt{SETUP}$ time units after the glitch, due to the

[20] This figure has already been published in [GEFP12a, Figure 21].

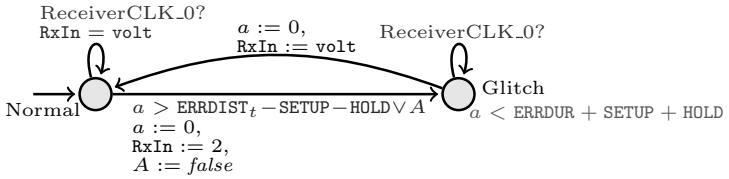


Figure 10.17: Model of real-time glitches: At most one glitch in $\texttt{ERRDIST}_t$.‡[21]

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 the sampling occurs at least HOLD time units before the glitch. Thus, the automaton shown in Figure 10.17 moves to the location Glitch HOLD time units before the occurrence of the glitch, and leaves it ERRDUR + 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 ERRDUR + SETUP time units after the occurrence of the last glitch, there will be no glitch for the next $\texttt{ERRDIST}_t$ − SETUP time units, and thus no move to the location Glitch for the next $\texttt{ERRDIST}_t$ − SETUP − HOLD time units. The boolean variable A, initialized to *true*, is used to allow the occurrence of the first glitch at an arbitrary position, as clocks are automatically initialized with 0 (DBMs only offer efficient support for resetting clocks to 0, and not to any other value, see Section 3.4).

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 10.17 is then replaced by the automaton shown in Figure 10.18. 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 the one of the glitch locations (GlitchB).

10.7 Oscillators

The local oscillators of the sender and the receiver are modeled 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 [Fle05, Appendix A.1].

[21] This figure has already been published in [GEFP12a, Figure 17].

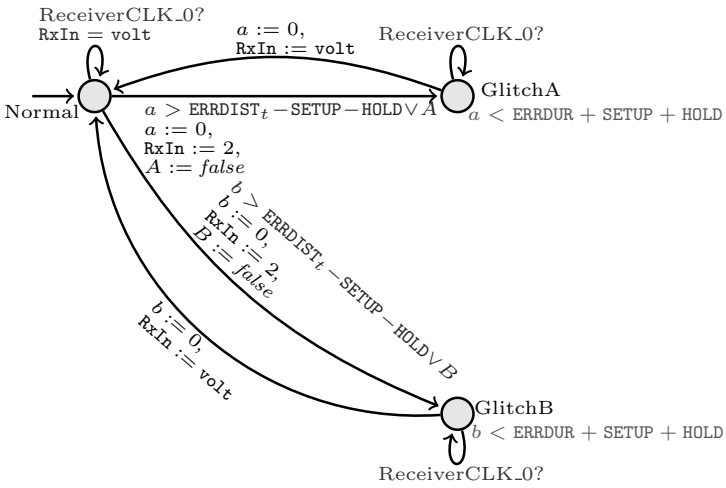


Figure 10.18: Model of real-time glitches: At most 2 glitches in $\texttt{ERRDIST}_t$.‡[22]

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 Figures 10.19 and 10.20. Here, x and y are continuous-valued clock variables.

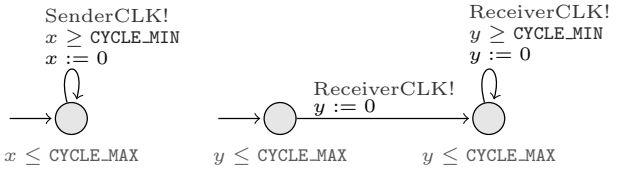


Figure 10.19: Oscillators for sender and receiver.†[23]

The models are parametrized in the length of an ideal clock cycle (which is the same for each controller) by CYCLE. To model the deviation, the parameter DEVIATION is used. This provides a lower and an upper bound for tick-events:

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

Note that the local behavior of the protocol in the sender and in the receiver can be described in discrete terms. Each controller is assumed to work as specified and

[22]This figure has already been published in [GEFP12a, Figure 18].

[23]This figure is based on figures already published in [Ger10, Figures 6.2 and 6.3]. It has already been published in [GEFP10, Fig. 14].

the model focuses 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 [MP11]. In turn, this allows 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.

Replacing Time with Order

To reduce the number of clocks in the model, and thus keep model checking efficient, replacing time with order where possible can be helpful—that is, if this does not introduce too much discrete complexity like large extra counters.

The order in which the data is processed by the different components is important and is fixed using a chain of components. In a setting in which the events are ordered, two different forms of communication between components are available. For asynchronous communication, shared variables can be used, as the order of accesses to the variables is fixed by the order. For synchronous communication, the handshake synchronization channels of timed automata can be used directly. To avoid blocking in synchronous handshake communication, "receiving" components are designed to have an enabled transition labeled with the respective communication channel at any point in time.

Model † uses separate broadcast synchronization channels and committed locations to achieve an order on all steps. For model ‡, the oscillator components as shown in Figure 10.20 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.

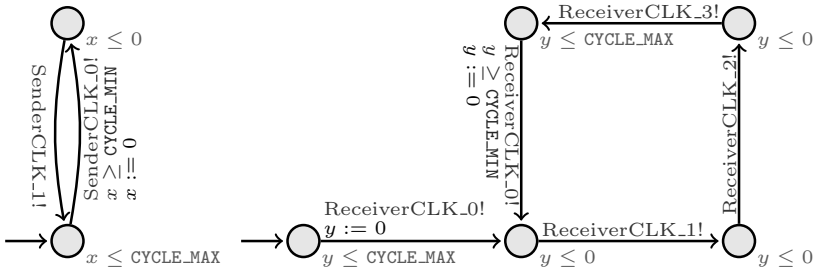


Figure 10.20: Oscillators for sender and receiver.‡[24]

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, allowing to structure the sender's process into two consecutive steps. 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 receiver's clock cycle into 4 consecutive steps.

[24]This figure has already been published in [GEFP12a, Figure 13].

10.8 Modeling the FlexRay Protocol

Recall the description of the FlexRay physical layer protocol from Section 2.2. As explained in Section 10.4.1, the scenario to be considered is one sender connected to one receiver.

10.8.1 Modeling the Sender

The sender has to assemble a redundant bit stream from the message to be sent, as shown in Figure 10.1. The number of possible messages in a transmission protocol is astronomic: in the case of FlexRay, there are more than 10^{600} possible different messages.[25] Protocols typically have a fixed format for messages, such as the format of FlexRay message frames shown in Figure 2.2 on page 15. Moreover, the bit stream to be assembled from the message also has a specific format, as shown in Figure 2.4 on page 18.

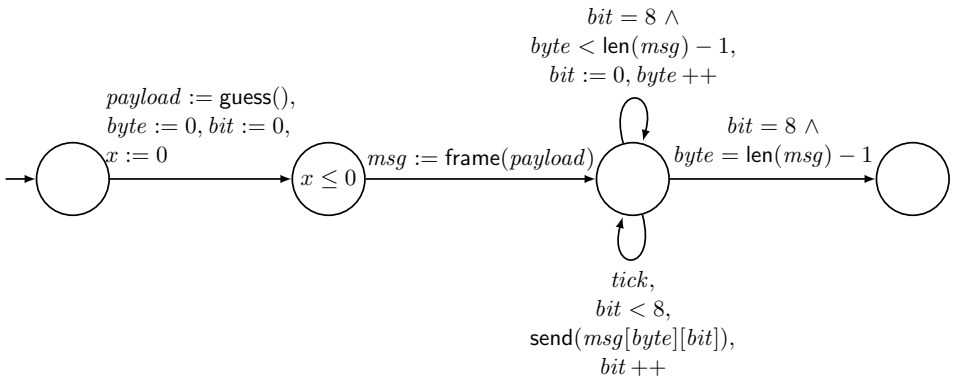


Figure 10.21: Generating a message payload, encasing it in the frame format, and sending it.[26]

Since a natural specification of the correctness of the protocol would require *every* message to be transmitted correctly, the large number of messages immediately translates to an equally large state space: for example, a model might first fix and store the data to be sent, then transmit the message via sender and receiver, and finally compare the stored data against the delivered message. Consider the automaton shown in Figure 10.21. The data to be transmitted is initialized in some array *payload*, whose values are chosen nondeterministically using the (pseudocode) function guess(). Then, pseudocode is used to encase *payload* in the frame format shown in Figure 2.2

[25] Moreover, if the fact that each message byte generates 10 bits in the formated message stream, and every bit in that stream generates on average 8 samples and each of these samples could be affected by a bit flip due to a transmission error (if glitches are to be considered) are taken into account, more than 10^{6000} scenarios need to be considered.

[26] This figure has already been published in [GEFP12b, Figure 11].

on page 15 using the function frame(), which generates a message frame that is stored in an array *msg* containing len(*msg*) bytes. The bits of *msg* are then sent to a message stream generator (not depicted) using the function send(). The generator then guarantees the format seen in Figure 2.4 on page 18.

Such a naive approach would result in a huge state-space for all nontrivial messages. To not blow up the state-space, storing the actual data has to be avoided.

Replacing Data with Nondeterminism

If there are several possibilities for valid values of a variable, one can often let the model forget the actual data, and instead nondeterministically guess the data whenever the model's behavior depends on its value: whenever the choice of a transition in the model depends on the actual data (e.g., the message), it just nondeterministically chooses a transition

This might over-approximate behaviors of the protocol, but if a reachability check still validates the property to be checked (no spurious error was introduced due to the approximation), it is a valid modeling choice that hugely reduces the number of scenarios to be checked (by making similar situations undistinguishable , e.g, situations just distinguished by no longer relevant stored data from the past) as well as the memory needed to store a state of the model.

In case of a physical layer protocol, in addition to the data to be sent, one can often also ignore the format of the message. This format often only contains information for higher protocol levels which is not used by the physical layer protocol. The components needed to enforce this format can be eliminated here, as the modeled protocol layer does not depend on the format.

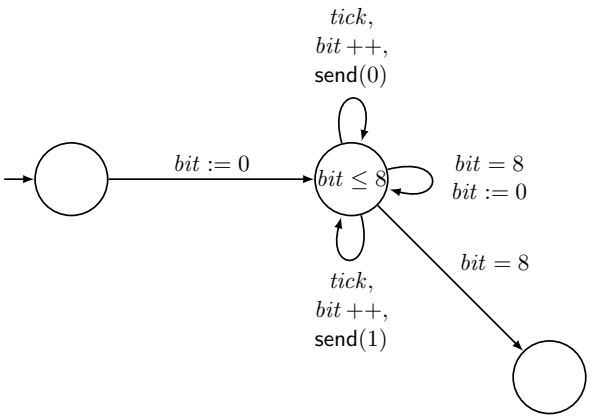


Figure 10.22: Generating and sending of a message while abstracting from its actual length, contents and format.[27]

[27]This figure has already been published in [GEFP12b, Figure 12].

Dropping the idea that only message frames in a legal format are to be transmitted in the model, but rather all messages of a possible length, altogether removes the necessity to store the initial message: the message bits can be just guessed on-the-fly whenever needed. Additionally, the length of the message can also be guessed on-the-fly by nondeterministically deciding whether to send another byte after a byte has been sent. Thus, the message can have an arbitrary length (in bytes) and it is not even necessary to store how many bytes have already been sent. The resulting automaton is shown in Figure 10.22. Its state space is reduced to only 11 states: The automaton could be in the first location, it could be in the second location and *bit* could have any integer value from 0 to 8, or it could be in the last state, where only 8 is a possible value for *bit*.

The drawback of ignoring the message is, of course, that it is no longer possible to specify that the messages are received correctly. It is sufficient, however, to store a *single bit* of the message, as long as the bit is chosen nondeterministically. If a bit is not correctly transmitted, the nondeterminism guarantees that there is at least one trace of the model where this bit is stored. In order to verify that the one stored bit is correctly received, the received bit that corresponds to the stored bit needs to be identified. For this purpose, the model of the sender shown in Figure 10.23 stores the position pos_{stored} of the bit in the message, and the model of the receiver shown in Figure 10.24 counts the received message bits until it identifies the bit corresponding to the stored bit bit_{stored}.

Abstracting from the actual message also keeps the sender process model small and simple, as it is not concerned with the message composition this way. 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, as described in Section 10.8.2.

Both model † and model ‡ thus have a sender model that is mainly concerned with the assembly of the structured stream and adding redundancy to it by sending every bit of the stream as a bit cell of a length of 8 sample clock cycles, while the actual frame bits are guessed on the fly. Both models avoid storing the position of the nondeterministically chosen frame bit to be stored for verification in terms of its position in the message by storing its position in the byte only, fully eliminating the need to count bytes. In fact, the models are identical, the only difference being the clock signal marking the sample ticks, which is `SenderCLK` for model † and `SenderCLK_0` for model ‡. In Figures 10.25 to 10.27, only the automata for model ‡ will be shown.[30]

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 [Fle05, Section B.2.1]. It precedes every trans-

[28]This figure has already been published in [GEFP12b, Figure 13].

[29]This figure has already been published in [GEFP12b, Figure 14].

[30]The Model for † can be obtained by replacing `SenderCLK_0` with `SenderCLK` in these figures, as they are based on figures already published in [GEFP10, Fig. 3 to 5], which, in turn, were closely based on figures already published in [Ger10, Figures 6.10 to 6.12]. They have already been published in [GEFP12a, Figures 5 to 7].

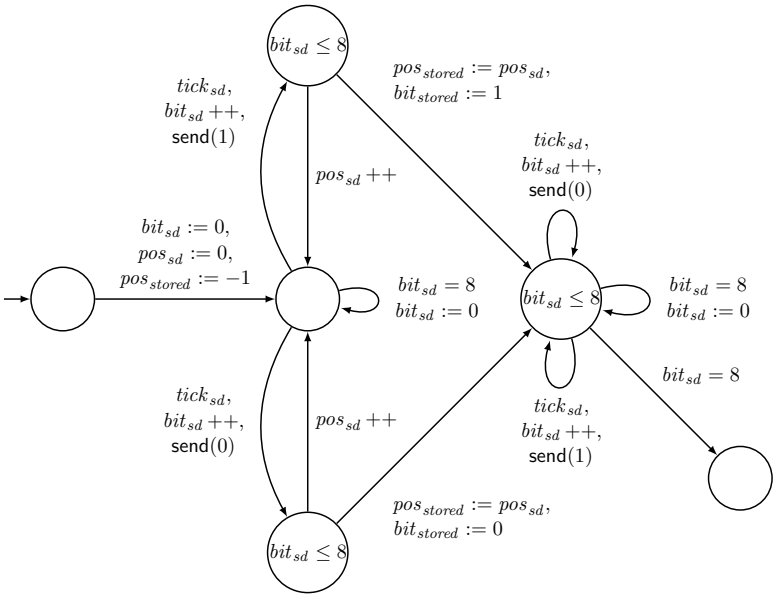


Figure 10.23: Generating and sending of a message while abstracting from its actual length, contents and format. The sender model nondeterministically chooses a bit to store and stores its position as well.[28]

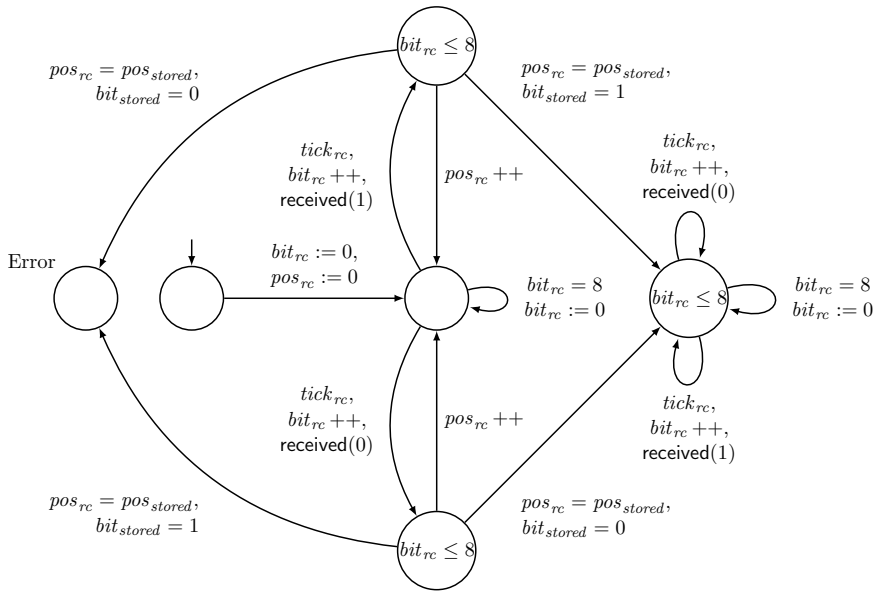


Figure 10.24: Receiving a message while abstracting from its actual length, contents and format. The receiver model checks that the bit received at the position of the stored bit has the same value as the stored bit.[29]

mission. 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. This aspect of the models is shown in Figure 10.25. Note that each stream bit is used to drive the bus through the variable `Tx` for a bit cell of 8 sample ticks.

The bit string of the frame is partitioned into bytes. Each frame byte is prefixed with a *byte start sequence* (BSS). The BSS consists of one high bit followed by one low bit. As shown in Figure 2.5 on page 19, each bit is put on the bus for a bit cell with a duration of 8 sample ticks. The high to low transition in the middle of the BSS is used as a trigger for the bit clock alignment. In the model shown in Figure 10.26, the frame bits are nondeterministically chosen. The decision to store a bit to be verified is also made nondeterministically, and the bit is stored in `savedTx`. Its offset within the current byte is stored in `savedindex`.[31] Note that several bits could be chosen to be stored, each bit overwriting the previous one. After each byte, the model can nondeterministically decide to either end the transmission or go on.

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. In Figure 10.27, the variable `End` is used to signal to the receiver that the bit stream is about to end—an addition to the model required to compensate for the lack of a predefined number of message bytes. This variable will be used to force the receiver to check the correct reception of a FES and then stop checking afterwards only when the message transfer is actually about to end, and not earlier, e.g, due to (hypothetical) errors in receiving some BSS. Note that `End` is only used here in a model for verification purposes, and is not part of the protocol.

The models of the sender do not have to model its internal structure or its processes, just its output. Basically, its purpose in the verification scenario is solely to provide the input for the transmission over the hardware in the expected form and to acquire the data needed to check the correctness of the reception of the transmission by the receiver after the physical layer protocol (which is the focus of the verification effort and thus modeled in detail together with the hardware and the error model) has done its job.

10.8.2 Modeling the Receiver

The receiver samples from the bus, resulting in a stream of samples with a lot of redundancy, as several samples will be taken from each bit cell. Before the steps depicted in Figure 10.1 on page 104 of removing this redundancy, checking the formated stream, and extracting the frame (and extracting the message from the frame, which will not be considered in the model as the frame was nondeterministically generated, see Section 10.8.1), the redundancy is put to use.

[31] The initial value `savedindex` = 8 means “no bit to test”.

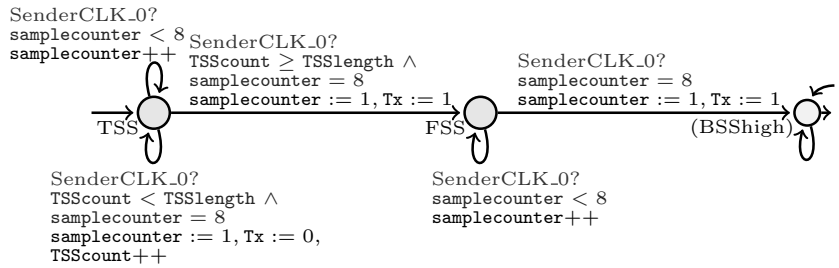


Figure 10.25: Model of the start of the transmission.‡[30]

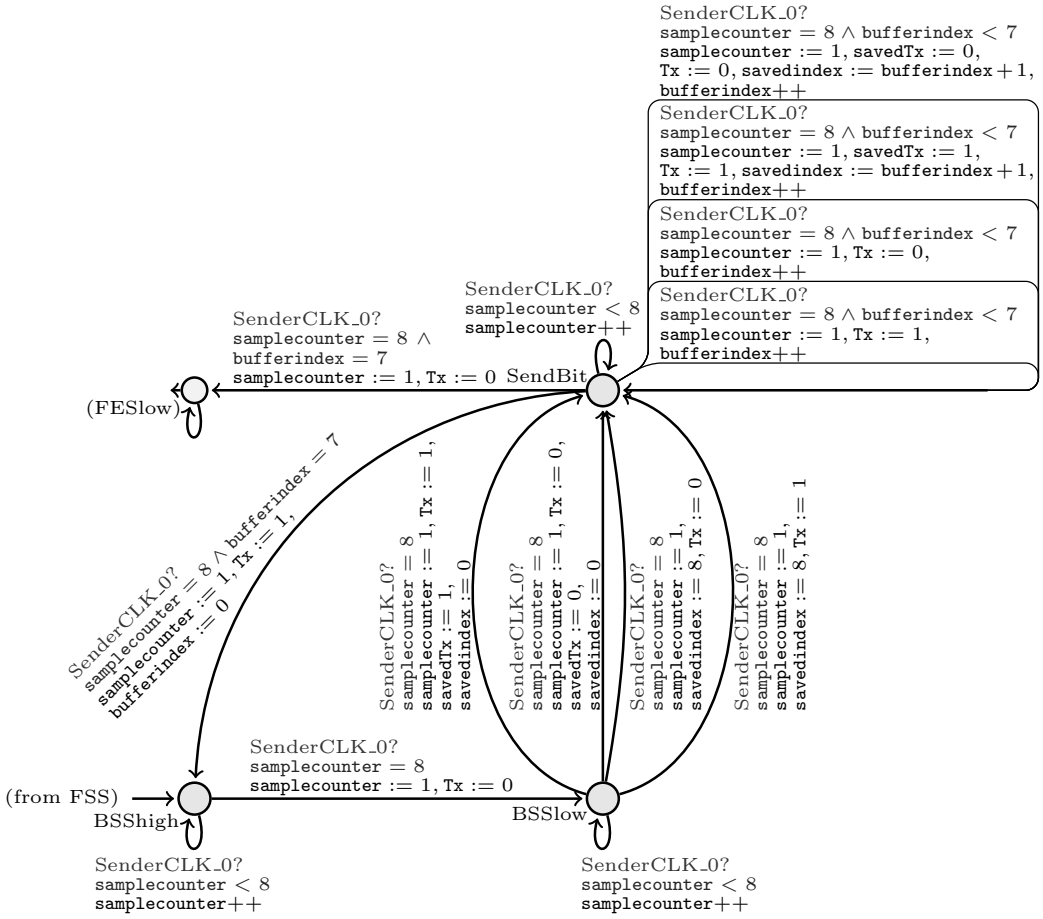


Figure 10.26: Model of the transmission of the message bytes.‡[30]

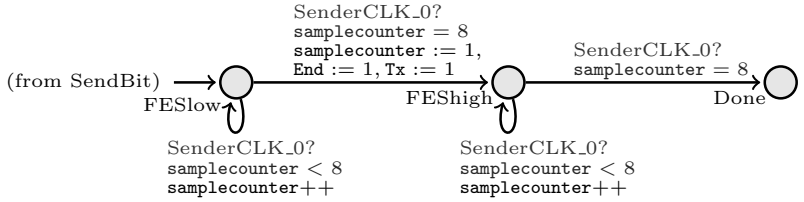


Figure 10.27: Model of the end of the transmission.‡[30]

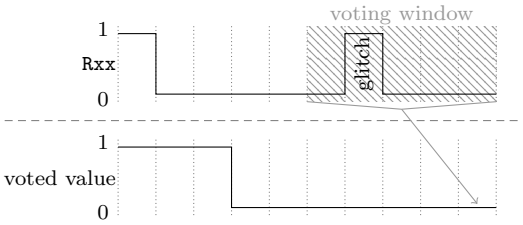


Figure 10.28: Correction of a glitch through majority voting.[32]

Recalling the description of the FlexRay physical layer protocol from Section 2.2, two processes deal with the redundancy:

1. Voting
2. Strobing and bit clock alignment

As shown in Figure 2.5 on page 19, voting uses the redundancy to compensate for glitches, while strobing, which is enabled by the bit clock alignment, gets rid of the redundancy in a way that helps to compensate for glitches and jitter.

Voting

The whole point of having redundancy at all is to compensate for glitches and jitter. Consider Figure 10.28 showing the relationship between the stabilized samples in `Rxx` (see Figure 10.3) from model † (model ‡ directly uses `Rx` for this) and the voted value resulting from the voting process: The five most recent samples always form the so-called *voting window*.[33] 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. Infrequently occurring glitches are mostly filtered out directly. However, voting also introduces a delay of two clock cycles for recognizing a changed value on the bus, as it takes three samples instead of one to make this change visible as a new majority in the voting window.

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, as depicted in Figure 10.29. 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 Section 10.4.2, though only if sampling occurs to close to the change in

[32] This figure is based on a figure already published in [Ger05, Figure 6]. It has already been published in [Ger10, Figure 5.8] and [GEFP10, Fig. 6].

[33] According to the FlexRay standard [Fle05, Section 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 [BBG+05, Sch06, Sch07, ABK08b, KP07, GEFP10, GEFP12a].

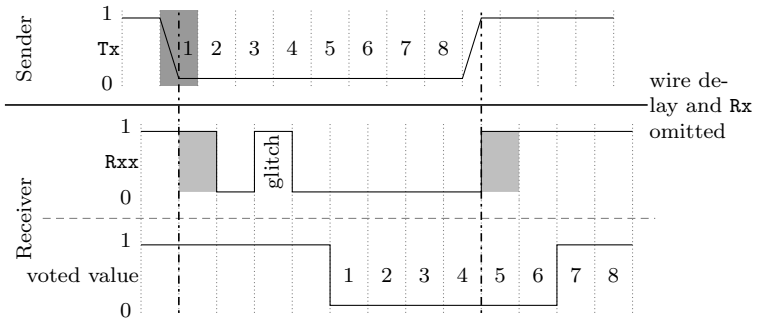


Figure 10.29: Combination of jitter and glitch.[34]

the bus value at the edges of a bit cell such that hold or setup times are violated, as is the case in Figure 10.29.

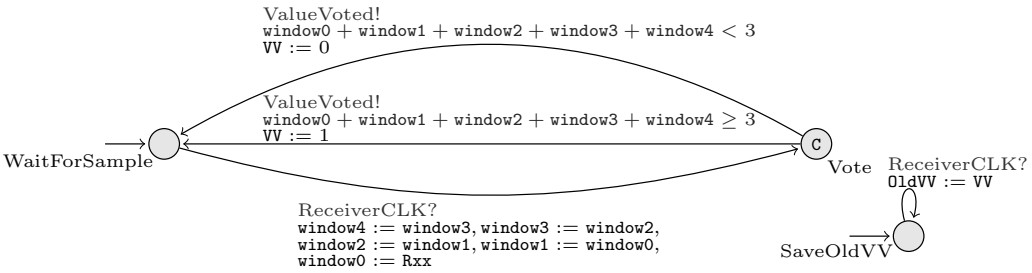


Figure 10.30: Model of the voting process.†[35]

The 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 clock cycle, the values of the `window` variables are shifted accordingly. If the majority of the `window` variables contains a 1, the voted value `VV` is set to 1, and to 0 otherwise. The respective previous value of `VV` is stored in `OldVV`. As shown in Figure 10.30, model † takes the samples from register `Rxx` and uses a construction with a committed location to make sure the old voted value is stored and the voting window is updated before a new majority is checked for and a new voted value is declared. To declare a new voted value, it uses the channel `ValueVoted`.

Its chaining of clock signals from Section 10.7 enables the model for ‡ to be simpler. As shown in Figure 10.31, the voting window is updated together with checking for a new majority in the third step of each receiver clock cycle. This is enabled by using `Rx` directly and also allows to get rid of the variable `window4`, reducing the state-space even further. The old voted value is saved in the first step of each receiver cycle.

[34]This figure is based on a figure already published in [Ger10, Figure 5.9]. It has already been published in [GEFP10, Fig. 7].

[35]This figure is based on figures already published in [Ger10, Figures 6.7 and 6.8]. It has already been published in [GEFP10, Fig. 8].

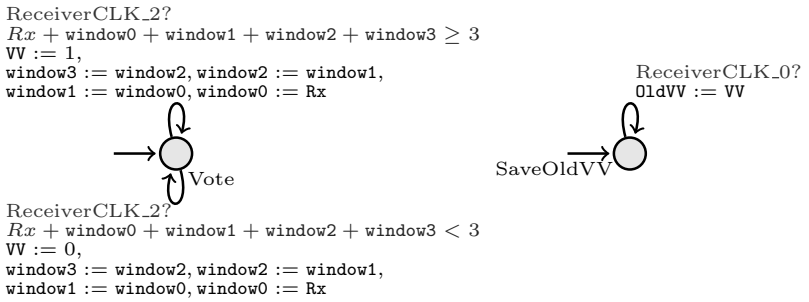


Figure 10.31: Model of the voting process.‡[36]

Strobing and Bit Clock Alignment

From each bit cell, only one voted value is used to reassemble the bit stream, as shown in Figure 2.5 on page 19. 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*.

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.

This *bit clock alignment* mechanism makes use of the bit stream format shown in Figure 2.4. 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 this low bit cell is considered to be the second voted value of the bit 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 Figure 10.29; recall the delay of two cycles introduced by the voting process. The bit clock alignment can analogously also happen too early.

In the model, `strobecounter` has no default value, but is initialized nondeterministically in order to model the absence of bit clock alignment at the start of the verification scenario. 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 was the first bit of the new bit cell, and the bit clock alignment mechanism is deactivated using `EnableSyncEdgeDetect`.

The variable `strobecounter` is incremented whenever a new voted value has been calculated until it reaches 8, then it is set to 1 in the next cycle, if the bit clock alignment does not interfere. When `strobecounter` has a value of 5 and the voted value for this cycle of the receiver's clock is ready, `VV` is chosen as the voted value to be strobed.

[36] This figure is partially based on a figure already published in [Ger10, Figure 6.8]. It has already been published in [GEFP12a, Figure 8].

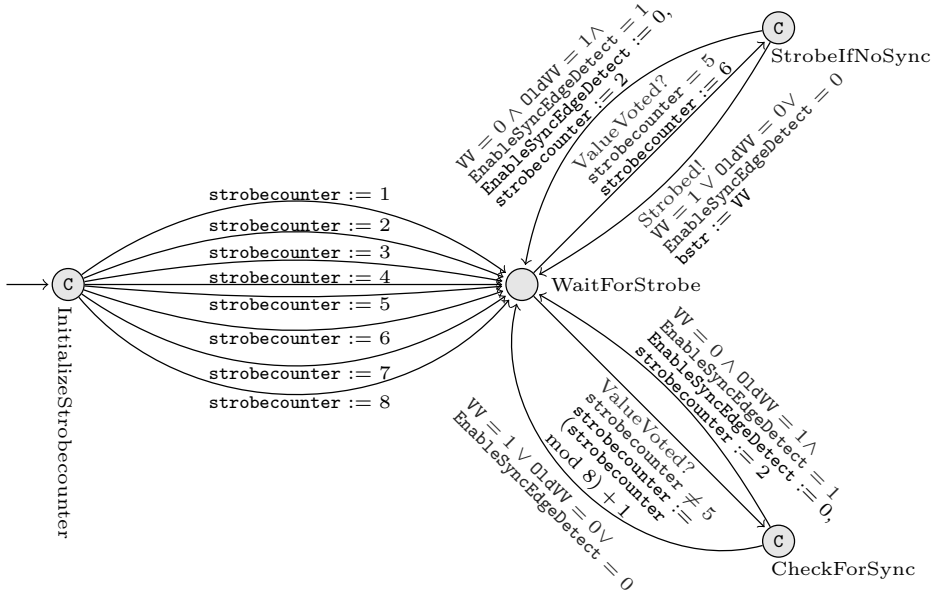


Figure 10.32: Model of the strobing process.†[37]

Figure 10.33 shows how the model for model † uses the variable `bstr` to store the strobed voted value and uses the channel `Strobed` to allow other automata to synchronize on this event in order to use the new `bstr` value. Channel `ValueVoted` is used to synchronize to the event of the current receiver's clock cycle's voted value being chosen, again using a construction with committed locations to chain the steps in the right order without the passing of time.

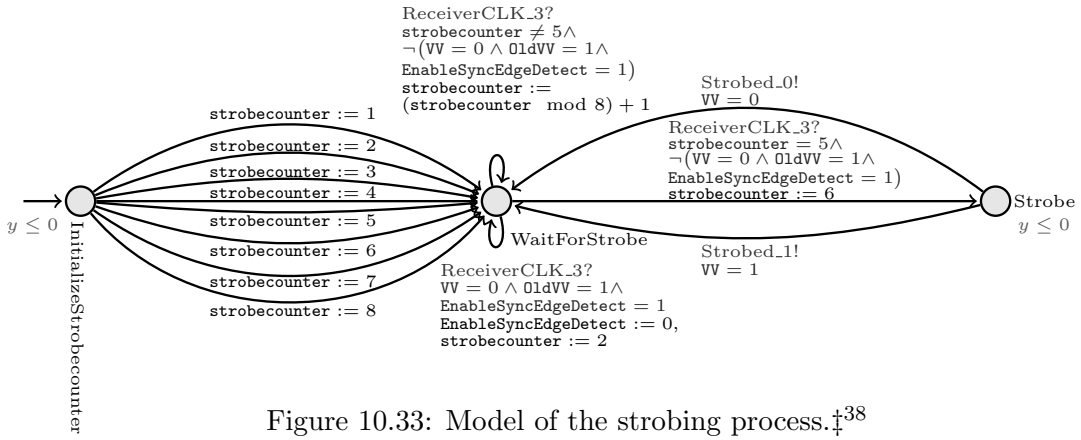


Figure 10.33: Model of the strobing process.‡[38]

[37] This figure is based on a figure already published in [Ger10, Figure 6.9]. It has already been published in [GEFP10, Fig. 9].

[38] This figure has already been published in [GEFP12a, Figure 9].

The model for model ‡, as shown in Figure 10.33, activates the strobing and bit clock alignment process in the 4th step of the receiver's clock cycle. Using a construction with locations that do not allow time to pass—using a constraint on the clock used by the model of the receiver's oscillator, y, which is known to have been reset to 0—a 5th step of the receiver's clock cycle is introduced: The channels `Strobed_1` and `Strobed_0` allow other automata to synchronize to a new strobed value encoded in the choice of which of the two channels is used. This allows to omit the variable `bstr`, making the model smaller.

Communication via Synchronization

If a one bit value has to be communicated at a specific point in time or step in the order, communicating it via a shared variable can often be replaced by encoding it in a synchronization signal if it is consumed directly and does not need to be stored, getting rid of the variable in the process.

Note that the trick of not letting time pass works well in model ‡, because it is done after the *last* step in the chain of clock signals, thus not introducing an alternative order in which the individual chained processes could be triggered.

Checking the Received Bit Stream

Similarly to the model of the sender, the model of the part of the receiver that extracts information from the stream of received strobed values does not need to be modeled in detail. It is enough to basically encode the property that the stream was correctly received into the form of an automaton. However, that entails more than just checking a stored bit as done in Figure 10.24: In order to enable the bit clock alignment mechanism from Section 10.8.2 at the right time, the format of the structured stream needs to be used.

As soon as a new strobed value is signaled, the receiver model checks if it is consistent with the expected format of the bit stream. As soon as a received value is not the expected one, the error state `DECerr` is entered.

Note that as discussed in Section 8.3, the non-reachability of this error state is also used to encode the liveness property that, when the process of checking the received bit stream for the expected stream format has started (by having received the first bit of the TSS), a message is eventually received (in case of finite messages). Thus, most of the correctness property that a message is received as it was sent is encoded in the model, just the reachability of a location needs to be encoded in the specification the model is checked against.

The received TSS is accepted if it contains at least `TSSmin` bits. Further bits of the TSS are accepted if not more than `TSSmax` bits have been received before.[39]

[39] In the FlexRay specification [Fle05], the parameter *gdTSSTransmitter* is used, which corresponds to `TSSlength` in the model. To replicate the behavior described in [Fle05], `TSSmax` should be set to `TSSlength`, and `TSSmin` should be set to 1. In a newer version of the FlexRay specification, [Fle10b],

During the reception of the TSS or after the reception of a message byte, the variable `EnableSyncEdgeDetect` is used 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 the variable `bufferindex`. When `savedindex` indicates that the current message bit is to be verified, the received strobed value is compared to `savedTx`. The variable `End` is checked to prohibit entering the location `Done` too early after (hypothetically) erroneously classifying a BSS as the FES.

Note that the model neither checks the number of received message bytes nor the format of the message as required by [Fle05, Section 3.3.6], as the model abstracts from the content and the length of the message.

Furthermore, 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 model into `DECerr`.

Dedicated End Locations

As soon as the scenario has reached a state that indicates that either the property to check has been violated or can no longer be violated, the exploration of the model does not need to continue. These locations can thus be made to stop time, limiting the state-space reachable from them to make the model checking stop earlier.

The presented models contain exactly two dedicated end locations: The absence of transmission errors can be checked by determining the *reachability* of the *error location*, as advocated in Section 8.1. After the complete reception of a FlexRay message stream, the receiver controller enters the *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 these states that can no longer reach the error location when the safe location has been entered, or cannot undo the error in case the error location was entered.

Furthermore, this allows to check for absence of transmission errors and the absence of other deadlocks in the same model checking run by checking for the reachability of deadlocks outside of the safe location. This then also covers the case of erroneous non-reachability of the error location in a badly designed model that got stuck in a deadlock before a path to the error location was discovered.

For model †, the channel `Strobed` signals that a new value has been stored in `bstr`. Figures 10.34, 10.35 and 10.36 show the use of a construction with committed locations in order to make sure that the check of correct reception indeed happens at the last step of the current receiver's clock cycle.

For model ‡, Figures 10.37 and 10.38 show how using the channels `Strobed_0` and `Strobed_1` instead of a variable like `bstr` allows to make the model smaller and easier

this behavior was modified to accept a longer TSS. The model can, however, easily be adopted to the modified behavior by setting `TSSmax` to `TSSlength` + 1. This demonstrates the robustness of a parameterized model to small changes of the specification.

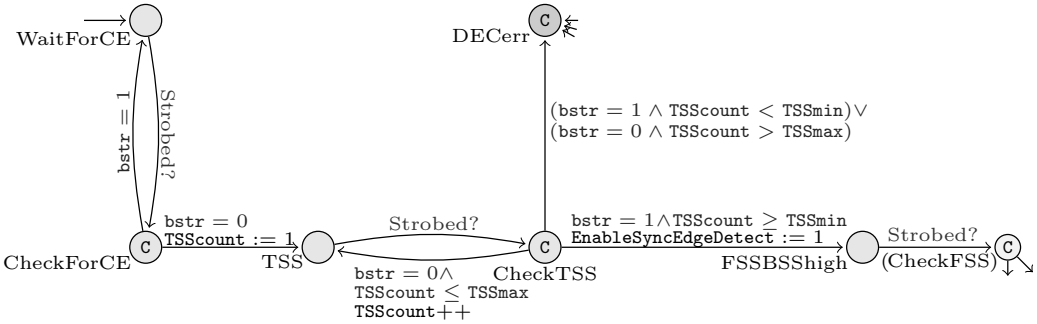


Figure 10.34: Model of the start of the reception.†[40]

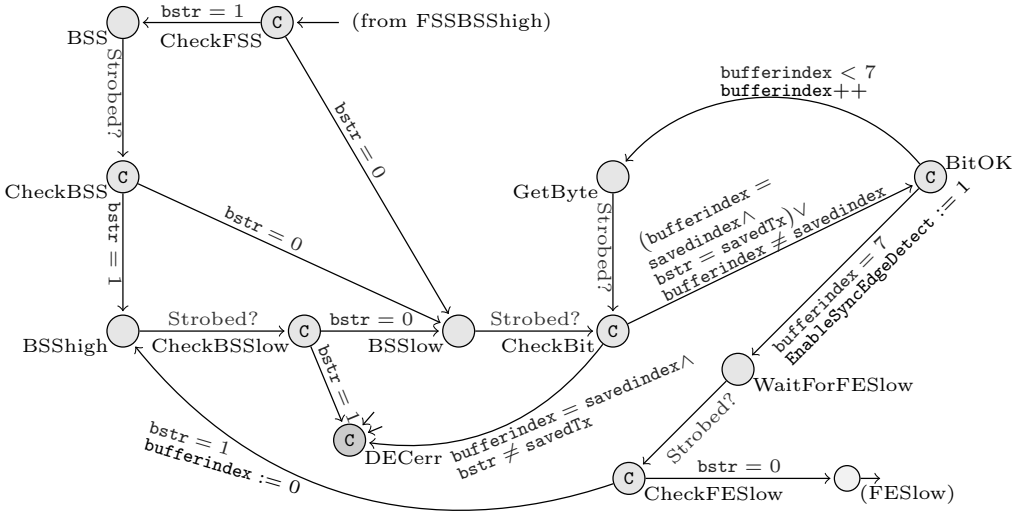


Figure 10.35: Model of the reception of the message bytes.†[41]

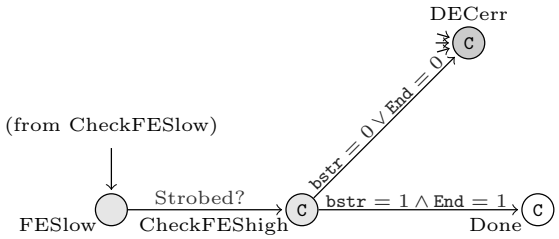


Figure 10.36: Model of the end of the reception.†[42]

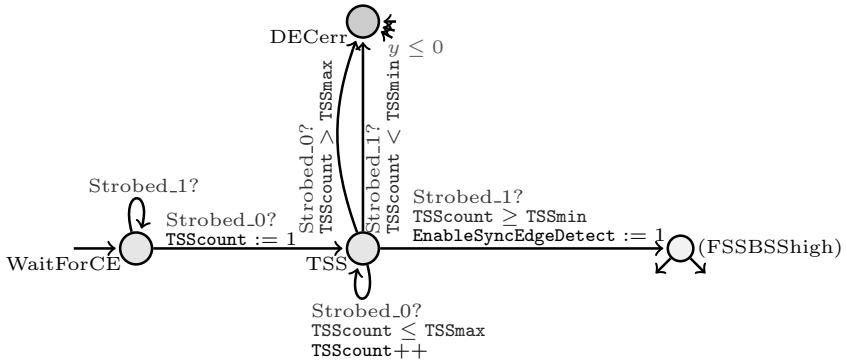


Figure 10.37: Model of the start of the reception.‡[43]

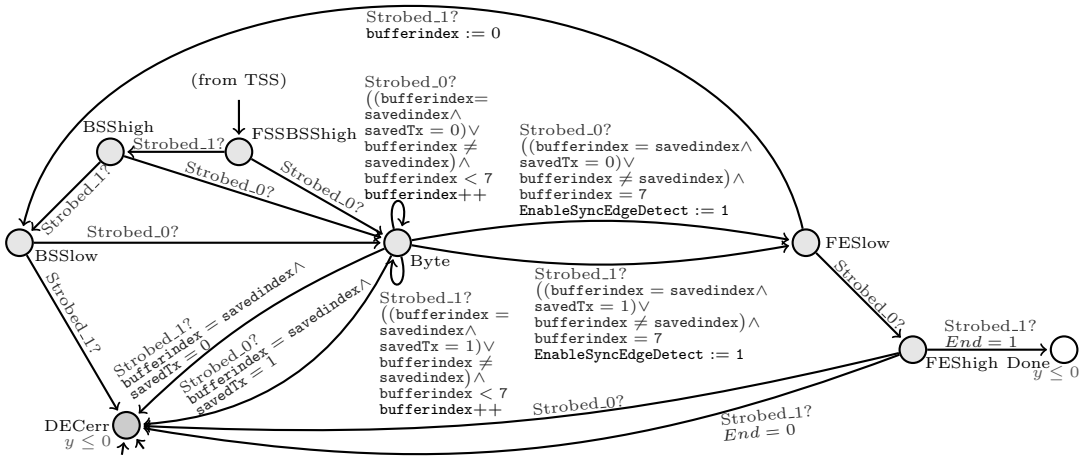


Figure 10.38: Model of the reception of the message bytes and the end of the reception.‡[44]

to understand and how the order imposed by them and the chain of ReceiverCLK synchronizations as described in Section 10.7 gets rid of committed locations.

[40]This figure is based on figures already published in [Ger10, Figure 6.13 and A.2]. It has already been published in [GEFP10, Fig. 10].

[41]This figure is based on figures already published in [Ger10, Figure 6.14 and A.1]. It has already been published in [GEFP10, Fig. 11].

[42]This figure is based on a figure already published in [Ger10, Figure 6.15]. It has already been published in [GEFP10, Fig. 12].

[43]This figure has already been published in [GEFP12a, Figure 10].

[44]This figure has already been published in [GEFP12a, Figure 11].

Chapter 11

Model Checking the FlexRay Physical Layer Protocol[1]

Contents

Abstract. Using the models presented in Chapter 10, Uppaal is used to prove the error resilience of the FlexRay physical layer protocol against various glitch patterns. Exploring the effect of changing assumptions on hardware properties on this error resilience, the robustness of the FlexRay physical layer protocol against lower quality hardware or changes in the hardware is evaluated. A comparison of the analysis results obtained with an early model version and those obtained with an optimized model version shows the importance of finely crafted models for the success of such a model checking endeavor. An analysis of the glitch patterns shown to be tolerable reiterates the need for automatic model checking to support and improve on manual analysis of complex systems.

In Chapter 10, two models for model checking the FlexRay physical layer protocol have been described:

1. the early model †, which uses Uppaal specific syntax and has been presented in [GEFP10], and
2. as an evolution of it the optimized model ‡, which has been presented in [GEFP12a] and avoids using Uppaal specific syntax.

[1] This chapter contains parts already published in [GEFP12a], and [GEFP10].

Both models have been model checked in order to analyze the FlexRay physical layer protocol. While Section 2.3.1 gave a preview of the results of this analysis, a detailed description of those results and an analysis of the findings is given in the following. Section 11.1 discusses the results obtained from model † with a 32-bit version of UPPAAL, which have been presented in [GEFP10]. Section 11.2 describes how model ‡, and the use of a newer 64-bit version of UPPAAL, allowed to improve on the previous results. Finally, Section 11.3 interprets the obtained resilience results against specific glitch patterns.

11.1 A First Verification of the FlexRay Physical Layer Protocol[2]

Consider the following analysis of the model † with fixed values for the model parameters and a check of several correctness properties (shown in Table 11.1) using the real-time model checker UPPAAL [BDL04] in its 32-bit version. This first analysis uses conservative approximations based on [Fle05, Nan09], which are listed in Table 11.2(a).

Table 11.1: Satisfied correctness properties of † and corresponding running times of UPPAAL on a computer with an AMD Opteron 2.6 GHz processor and 4 GiB RAM.[3]

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

Initially, an error distance of four is assumed, which corresponds to one glitch in a voting window (1 out of 5). This intuitive choice is overly pessimistic: in fact, the experiments show that for the standard parameters, the physical layer protocol model can tolerate an error distance of three (1 out of 4) 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 11.1 are still preserved) is shown in Table 11.2(b). To adjust the glitch model, the automaton from

[2] This section contains parts already published in [GEFP10].

[3] This table has already been published in [GEFP10, Table 1].

Table 11.2: Standard values based on conservative approximations of the parameters taken from the FlexRay standard [Fle05] and the Nangate Open Cell Library [Nan09], as well as the impact of changed parameters on the tolerable glitches in †. 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".[4]

(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
$\texttt{PMAX} - \texttt{PMIN} \leq 6086$	1 out of 4
$\texttt{PMAX} - \texttt{PMIN} \leq 6086$	at most 2
$\texttt{PMAX} - \texttt{PMIN} \leq 9616$	at most 1
$\texttt{DEVIATION} \leq 92$	1 out of 4
$\texttt{DEVIATION} \leq 92$	at most 2
$\texttt{DEVIATION} \leq 218$	at most 1
$\texttt{DEVIATION} \leq 348$	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

Figure 10.13 was modified as needed. Interestingly, this further analysis of model † demonstrates the robustness of the FlexRay physical layer protocol even for more pessimistic hardware assumptions: beyond the initial conservative choice of the parameters, there is still a comfortable safety margin when reasonable glitch models are used.

Also explored was the size of the voting window. The last four rows of Table 11.2(b) show what happens to glitch tolerance if the voting windows size is changed. Note that a voting window needs an odd size to have a clear majority, a size of 1 would mean no voting and thus no glitch tolerance, and as a bit cell is only 8 sample ticks long, no more than 9 samples could ever be sampled from one bit cell, even if the receiver's clock is faster than the sender's one, as $9 * \texttt{CYCLE\_MIN} > 8 * \texttt{CYCLE\_MAX}$ for all viable clock deviation values. Here, the tolerable error distance between glitches affecting a single sample increases linearly in the size of the window. These results suggest that the relationship is $\texttt{ERRDIST} = \lceil \frac{voting\ window\ size}{2} \rceil$, i.e., that an error distance between single glitches equal to the number of samples needed for a majority in the voting window is tolerable.

Lastly, some more elaborate adjustments to the automaton from Figure 10.13 allowed to 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 11.2(a), it turns out that two glitches in a sequence of up to 82 samples lead to a violation of the correctness properties. This partial result will be completed to a more useful one in the analysis of model ‡ in Section 11.2. The limitation to 4 GiB

[4] This table has already been published in [GEFP10, Table 2].

here and the absence of real-time glitches in model † only allow to look at 2 arbitrary glitches, which may occur closer together than an `ERRDIST` of 3 allows, just once in the context of the whole scenario (at most 2)—as opposed to 2 arbitrary glitches in every consecutive sequence of a length of more then 82.

11.2 A Thorough Verification of the FlexRay Physical Layer Protocol[5]

To pick up where the analysis presented in the previous section fell flat, a non-reachability check of the error location in model ‡, which will be used in this section together with Uppaal version 4.1.4-64bit, verified that a voting window of size five allows for up to two glitch-affected samples at arbitrary positions in every sequence of 88 consecutive samples (2 out of 88). This was achieved using the sample glitch model shown in Figure 10.16. Here and in the following, the non-reachability of location `DECerr` will be used as the model checked property. However, the counting associated with the sample glitch model introduced a considerable amount of discrete complexity: Uppaal needed 210 minutes using 83 GiB of memory to verify this property. This check needed an advanced machine and required the employment of a newly available Uppaal version that supported 64 bit and could thus address more than 4 GiB of memory. If the glitch-affected samples were next to each other, it took only 262 seconds and 2.1 GiB of memory to verify this property in the improved model (‡).

The property that up to one glitch-affected sample in every sequence of four consecutive samples is allowed for (1 out of 4), which was already shown in [GEFP10] and is here also presented in Section 11.1, took 21 seconds to verify with the sample glitch model shown in Figure 10.14, using 207 MiB of memory.

Introducing real-time glitches helped to improve on those results. While tools specializing on handling discrete complexity should be able to handle the verification task more space-efficiently, introducing a third (and fourth) clock in the real-time glitch model instead of counting unaffected samples, as done in the sample glitch model, turns out to be better suited to Uppaal. The real-time glitch model decouples the occurrence of a glitch from the number of samples affected, but allows to analyze the relationship between maximal glitch duration and the minimal glitch-free time between two glitches. The model ‡ with the real-time glitch model shown in Figure 10.17 could be analyzed with Uppaal version 4.1.4-64bit in just 30 seconds using only 204 MiB for the short glitch (affects at most one sample in the sequence) version, and in just 43 seconds using only 311 MiB for the long glitch (affects at most consecutive 2 samples in the sequence) version.

The results from this real-time glitch model confirmed the 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

[5]This section contains parts already published in [GEFP12a].

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, allows to model two independent glitches, represented by the automata locations `GlitchA` and `GlitchB`, as shown in Figure 10.18. Clock a is associated to `GlitchA` and clock b to `GlitchB`. The clocks are used to measure the duration of a glitch modeled by their associated location, and the period after the glitch that is free of glitches modeled by this location. For example, if a glitch is 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 is 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 1.3 GiB of memory.

Again, these results confirmed the 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.

11.2.1 Analyzing the Parameters

The analysis of the hardware parameters is based on using Uppaal version 4.1.4-64bit to check the non-reachability of location `DECerr`, i.e., a check that no transmission error occurs. Additional properties were checked in [GEFP10] in the basic model †, but, being the essential property, only absence of transmission errors was checked in the model ‡ with varied parameters.

The analysis uses conservative approximations for the hardware parameters, based on [Fle05, Nan09], which are listed in Table 11.3. The results of the initial analysis using all the standard parameter values from Table 11.3 are summarized in Table 11.4. In a further analysis, the impact of changing *either* the hardware parameters `PMIN` and `PMAX` *or* `DEVIATION` on the amount of tolerable glitches is shown in Table 11.5.

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

Table 11.3: Standard parameter values for ‡ based on conservative approximations of the parameters taken from the FlexRay standard [Fle05] and the Nangate Open Cell Library [Nan09].[6]

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 11.4: Tolerable glitch patterns in ‡ 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".[7]

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	$\texttt{CYCLE\_MIN} - \texttt{SETUP} - \texttt{HOLD}$	$10.57125\,ns$
	$\texttt{ERRDIST}_t$	$87 * \texttt{CYCLE\_MAX} + \texttt{HOLD} + \texttt{SETUP}$	$1091.04125\,ns$

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 [MP11] 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 the result from the analysis presented in this chapter, namely 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 11.4, this demonstrates the resilience of the FlexRay physical layer protocol as well as its clock synchronization and time-division-multiple-access scheme against less precise oscillators.

Furthermore, the presented results demonstrate a high resilience of the FlexRay physical layer protocol against changes in the variance of the propagation delay. From [Nan09], the initial assumption of variance in the propagation delay was $1435\,ps$, but this could be increased 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

[6] This table has already been published in [GEFP12a, Table 1] and [GEFP12b, Table 1].

[7] This table has already been published in [GEFP12a, Table 2] and the first two lines of data have also already been published in [GEFP12b, Table 2].

Table 11.5: Impact of changes to the parameter values in ‡ 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 $\texttt{ERRDIST}_t$ correspond to are the changed results yielded by the formulas for their value.[8]

Changed parameter	Tolerable glitch patterns
$\texttt{PMAX} - \texttt{PMIN} \leq 6086$	1 out of 4
$\texttt{PMAX} - \texttt{PMIN} \leq 6086$	at most 2
$\texttt{PMAX} - \texttt{PMIN} \leq 6056$	2 adj. out of 88
$(\texttt{PMAX} - \texttt{PMIN} \leq 6056)^a$	2 out of 88
$\texttt{PMAX} - \texttt{PMIN} \leq 6086$	short real-time glitch
$\texttt{PMAX} - \texttt{PMIN} \leq 6056$	long real-time glitch
$\texttt{PMAX} - \texttt{PMIN} \leq 6056$	2 ind. real-time glitches
$\texttt{DEVIATION} \leq 92$	1 out of 4
$(\texttt{DEVIATION} \leq 80)^a$	2 out of 88
$\texttt{DEVIATION} \leq 90$	2 adj. out of 88
$\texttt{DEVIATION} \leq 92$	at most 2
$\texttt{DEVIATION} \leq 218$	at most 1
$\texttt{DEVIATION} \leq 348$	none
$\texttt{DEVIATION} \leq 92$	short real-time glitch
$\texttt{DEVIATION} \leq 90$	long real-time glitch
$\texttt{DEVIATION} \leq 80$	2 ind. real-time glitches

[a]Largest value which cannot be shown to be unsafe with 2 arbitrary glitch affected samples out of 88 consecutive samples when UPPAAL is limited to 127 GiB of memory, the maximum available during the experiment. However, substituting the sample glitch model with the real-time glitch model shows that 2 independent real-time glitches yield the same results, confirming the correctness of this value.

longer harness will be greater than that of a short one [HR09].

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.

11.3 An Analysis of the Tolerable Glitch Patterns[9]

Consider the glitch patterns that where shown to be tolerable. In hindsight, the resilience of the protocol against these glitch patterns can be partially explained using insight into the protocol.

More than two glitch affected samples in a voting window of size 5 can change the voted value, so this case has to be excluded up front. The cases where two or fewer glitches happen in one voting window are the interesting ones. For these, two patterns have been identified as tolerable:

[8]This table has already been published in [GEFP12a, Table 3] and some of the data has also already been published in [GEFP12b, Table 3] .

[9]This section contains parts already published in [GEFP12a].

1. not more than 1 glitch affected sample in every sequence of 4 consecutive samples (1 out of 4), and

2. not more than 2 glitch affected samples in every sequence of 88 consecutive samples (2 out of 88).

If at least one of these patterns applies to a given scenario with glitches on a stream of samples, the scenario will not lead to a transmission error. In the case of real-time glitches, this is analogous: If at least one of the tolerable patterns of maximal glitch duration and minimal glitch free period between glitches applies to a scenario with glitches in some time periods during the transmission, the scenario will not lead to a transmission error.

To understand what problems can still be caused if the glitch-affected samples do not constitute a majority in some voting window, keep in mind that the FlexRay physical layer protocol uses the strobing process to pick the 5^{th} voted value out of the 8 voted values corresponding to the 8 samples a bit cell is composed of in the ideal case. This strobing mechanism uses the counter variable `strobecounter` which is synchronized by the bit clock alignment to the sample stream at sync edges, which occur during the Byte Start Sequences (BSSs). The sync edges are thus separated by 10 bit cells, which consist of 8 samples each in the ideal case, yielding 88 samples. It is the interference of glitches with this bit clock alignment process and consequently with strobing that allows the glitches to wreck havoc, even if they do not constitute a majority in a voting window.

11.3.1 Tolerable Glitch Pattern 1 out of 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 $1 < i < 4$, 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 4^{th} or 6^{th} voted value may be strobed instead of the 5^{th}, while only the 1^{st} or 8^{th} 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 3^{rd} or 7^{th} 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.

11.3.2 Tolerable Glitch Pattern 2 out of 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 3^{rd} or the 7^{th} voted value to be strobed. Accounting for drift, the 2^{nd} or 8^{th} voted value could be strobed. However, the 1^{st}, 2^{nd}, 7^{th} or 8^{th} 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—this has to be excluded 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 3^{rd} voted value of each bit cell. Assume drift moves this to strobing the 2^{nd} 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 not only be 2 samples late after resynchronization: for the bit-cell composed of the samples send after the sync edge, the 2^{nd} voted value, which would be strobed immediately before the late resynchronization, would be flipped. As this scenario assumes 4 glitch-affected samples in a sequence of $1+3+80+1+4=89$ samples, it can be excluded as each sequence of 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 in 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. 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 87^{th} sample, the requirement that it is not affected by a glitch could be dropped. This thus cannot explain 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. As the example assumed that the receiver's clock is so much faster (and thus not using `CYCLE_MAX` as the duration of many of its clock cycles) that it overtakes the slower sender's clock, there must be another scenario that is behind this result.

The discrepancy between the requirements on the glitch patterns in the 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.

Chapter 12

Conclusion

This thesis presented detailed models of FlexRay's Physical Layer Protocol, a protocol that is used in industry in safety critical contexts. The models did not only help in gaining a better understanding of the inner workings of the protocol, they also were combined with a parameterized model of the underlying hardware and several error models. This combination allowed to verify error resilience guarantees for the protocol that were more precise than the relatively vague error resilience claims the designers of the protocol made in the specification [Fle05, Section 3.2.7]. The process of creating such models and optimizing them for verification was also documented in this thesis, providing a valuable case study that can serve as a road-map for similar endeavors.

As the verification of the protocol involved a lot of complexity in the discrete aspects of the model, parts of the verification effort turned out to be quite memory intensive. To conquer this challenge, not only less memory intensive models were presented, but also the fully symbolic data-structure CZMs and several algorithms using them were presented, which are tailored to handle a full state space exploration of a timed automaton model with little timing complexity but a lot of discrete complexity. This does not only offer an alternative to semi-symbolic approaches like Uppaal, but an alternative that is strong in exactly the area were semi-symbolic approaches are weak, namely when a lot of discrete complexity meets a limited timing complexity. The CZM approach was the basis for CMDs, which, in turn, were met with interest in the community. CZMs were also the basis for the novel approach that was presented in Chapter 7, which, being tailored to bug finding, might in some cases offer a faster exploration of the discrete state space of timed automata.

This thesis thus documents a verification effort that, being sparked by investigation of a new industrially used protocol, developed both modeling methods and advanced the state of the art in fully symbolic model checking for timed automata, while succeeding in providing proof of greater error resilience of the protocol under investigation than assumed in the protocols specification, and giving a precise descriptions of the errors against which the protocol has guaranteed resilience.

12.1 Contributions

This thesis presents work that makes three main contributions:

- The verification of the error resilience of the FlexRay physical layer protocol.
- The development of a model of the FlexRay physical layer protocol (including modeling principles).
- Fully symbolic algorithms and data-structures for reachability checking of timed automata.

The verification results presented in Chapter 11 and previewed in Section 2.3 provide trust into the error resilience and correct operation of the FlexRay physical layer protocol and inform researchers and engineers about the worst-case requirements on hardware timing behavior in relation to the desired resilience against disturbances on the transmission medium.

The modeling effort presented in Part III provides not only a family of models, but also the methodology used to derive it and the lessons learned, of which not the least is that it is possible to model an industrially used protocol in such a way that it can be verified using model checking with only moderate computing resources.

The fully symbolic algorithms and data-structures presented in Part II provide novel approaches to the problem of reachability checking for timed automata, which already led to a much noted enhancement, presented in work together with Rüdiger Ehlers, Daniel Fass and Hans-Jörg Peter [EFGP10], of the approach presented in this thesis in Chapter 6.

12.2 Advancing the State of the Art

Recalling the challenges of overcoming the limitations of previous efforts to verify complex real-time systems as described in Section 1.4, the presented modeling methodology and the algorithm and data-structure for fully symbolic model checking of timed automata successfully addressed the two standing in the way of using model checking on timed automata models of complex systems: Even though FlexRay's physical layer protocol is a rather complex scenario, a realistic model that keeps timing and discrete complexity on a manageable level was arrived at. Also, the fully symbolic approach to model checking timed automata keeps the memory requirement of storing the discrete parts of the statespace down, albeit there is the trade-off of often having a higher running time compared to semi-symbolic approaches.

The presented work thus paves the way to automatic verification of more complex real-time systems, breaking out of the realm of overly simplified or toy examples by showing that and how industrial protocols can be model checked in a real-time setting. The growing ubiquity of embedded and reactive systems even in safety critical areas like automotive or aerospace environments makes automatic verification of such systems imperative, and model checking optimized timed automata models provides one

solution to this ever more pressing need. It will, however, prove a challenge to keep pace with the rising complexity of the systems to be verified, reiterating the need to explore further approaches to keep pushing the capabilities of automatic verification to verify new generations of reactive and embedded systems as well as the protocols employed to connect them.

12.3 Impact

The parts of the work presented in this thesis that have already been published ([GEFP10], [EGP10], [GEFP12b], [GEFP12a]) have influenced and enabled work by others in the same or related fields. This section discusses this impact on published works.

Guo et al. [GLYA13] develop a framework for verifying properties of FlexRay with varying configurations using UPPAAL. They assume the startup of FlexRay to work as expected, which is verified by Malinský and Novák [MN10] using UPPAAL, and they assume the correct transmission of messages in the physical layer protocol with fault tolerance as verified in [GEFP10]. Even though built on these assumptions, which allow their model to ignore the physical layer protocol and the startup phase, the model, which focuses heavily on the dynamic segment and is presented in more detail by Guo [Guo12], needed a lot of abstraction and further simplifying assumptions such as perfect synchronization and absence of transmission errors (to avoid delays) to reduce its complexity and counteract state space explosion. Such an approach is only viable on the basis of assumptions about the correctness and resilience of the physical layer protocol [GEFP10], the startup mechanism [MN10, Tra16], and also the clock synchronization as shown by Böhm [Böh07]. Demonstrating the correctness of such assumptions thus enables work focusing on aspects of FlexRay which rely on these underlying mechanisms. However, manual verification efforts assisted by theorem provers like [MP11, Mül11] have so far been unable to take the unreliability of the physical transmission medium as described in Section 10.6 (glitches) into account. The verification effort described in Chapter 11 provides such works with the assurance that FlexRay is indeed provably resilient against certain patterns of glitches, and informs such approaches about the maximal resilience against jitter, which Müller an Paul [MP11, Mül11] identify as a target for future theorem proving approaches.

Neumann [Neu13] points out that real-time behavior such as drift and jitter need to be taken into account for the verification of distributed systems, and has designated an integration of verification work based on timed automata models as presented in [GEFP10] into his framework of *Language Progressive behavior*, that allows for modular analysis of real-time behavior in component based development contexts such as *AUTomotive Open System ARchitecture* (AUTOSAR) [FBH+06], as a topic that needs investigation.

The FlexRay model (†) developed in [GEFP10] and presented in Chapter 10 was used as a basis for a simplified FlexRay model that Miller, Gitina and Becker [MGB11] use to evaluate their bounded model checking approach with quantified SMT formulas

using *And Inverter Graphs* (AIGs), where they blackbox several components of the protocol. Tran [Tra16] uses the model (†) from [GEFP10] (see Chapter 10) to evaluate various exploration orders for timed automata model checking and verifies the startup mechanism of a FlexRay network in several configurations.

The modeling methodology as described in Part III can inform and inspire people trying to model comparable scenarios. A very similar problem to the verification of FlexRay's physical layer protocol's resilience against transmission faults is explored by Tóth, Vörös and Majzik [TVM15], who also use a timed automaton model. They verify master election and identifier assignment in a communication protocol for a distributed *Supervisory Control and Data Acquisition* (SCADA) system and its resilience against transient faults using Uppaal, applying the same data abstraction used in [GEFP10] and described in Chapter 10, combining it with a cone of influence abstraction and decomposition of the temporal specification.

Feo-Arenis et al. [FAWD+16] took up the challenge formulated in [GEFP10] to apply such a verification approach not *a posteriori* to an already standardized protocol, but *a priori* during the development of a protocol. Together with SeCa GmbH, a small radio technology company, Feo-Arenis et al. developed a timed automata model of a wireless fire alarm system that was under development and formalized the requirements of the certification authority. Employing a component based model with error model components, an approach advocated in Part III for providing flexibility in respect to changes in the design and clarity with a neat separation of concerns, they where able to apply Uppaal after a quasi-equal clocks abstraction [HWFA+12]. The results led to the early discovery of design flaws and provided information about time behavior to the designers before building of prototypes made such data available through measurements. This process of development aided by the verification effort significantly reduced the number of problems discovered at late stages of the development process and during testing of the protoypes, and enabled the developed wireless fire alarm system to pass all tests of the certification agency flawlessly, proving the value of informing the design of a protocol through a model based verification effort.

The sheer scale of the undertaking to model and verify a protocol like FlexRay or a meaningful part of it can be daunting. Thus, demonstrating the feasibility of model based verification of a bus architecture used by industry as done by [GEFP10], which is now one of the success stories for the application of Uppaal [GLLN18], encourages people to tackle similarly complicated protocols with model based approaches. For example, Graf-Brill, Hermanns and Garavel [GBHG14] use formal model based conformance testing with the aim of integrating it into the certification process of the *EnergyBus* standard.

In the field of conformance testing, there is also interest in fully symbolic methods for real-time systems, e.g., as voiced by Andrade and Machado [AM13], who bemoan the lack of approaches such as *constraint matrix diagrams* (CMDs) [EFGP10], which further evolve the approach described in Chapter 6 [EGP10], in the context of conformance testing for real-time systems. Bouyer et al. [BFGL+18] see promise in CMDs, and they could fill a real need, as Cordy [Cor14] voices interest in a more organic combination of decision-diagram based boolean encoding and DBMs such as CMDs. Cordy investigates

model checking of product lines by extending timed automata with *features*, which configure the model's behaviors to represent the products of the product line. These features are treated by BDDs while the real-time aspects are treated by DBMs, so the combination of BDDs with DBMs does work for more purposes than those described in Chapter 6 [EGP10]. The evolution of the approach from Chapter 6 [EGP10] into CMDs) [EFGP10] has thus created a data-structure that has been noted by a wider audience, e.g., Morbé, Pigorsch and Scholl [MPS11], Morbé and Scholl [MS12, MS13, MS14, MS15], Lu et al. [LMM+12], Wang and Jiao [WJ14], and Aicher, Rehberger and Vogel-Heuser [ARVH15].

Bibliography

[AB19] Omar Abbosh and Kelly Bissell. Securing the Digital Economy: Reinventing the Internet for Trust. Technical report, Accenture, 2019.

[ABK+97] Eugene Asarin, Marius Bozga, Alain Kerbrat, Oded Maler, Amir Pnueli, and Anne Rasse. Data-structures for the verification of timed automata. In *Hybrid and Real-Time Systems*, pages 346–360. Springer, 1997.

[ABK08a] E. Alkassar, P. Böhm, and S. Knapp. Correctness of a Fault-Tolerant Real-Time Scheduler and its Hardware Implementation. In *Sixth ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'08)*, pages 175–186. IEEE Computer Society, 2008.

[ABK08b] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Formal Correctness of an Automotive Bus Controller Implementation at Gate-Level. In Bernd Kleinjohann, Lisa Kleinjohann, and Wayne Wolf, editors, *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008)*, volume 271 of *International Federation for Information Processing*, pages 57–67. Springer, 2008.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *LICS*, pages 414–425. IEEE Computer Society, 1990.

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AGU72] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[Alu98] Rajeev Alur. Timed automata. *NATO ASI Summer School on Verification of Digital and Hybrid Systems*, 1998.

[AM13] Wilkerson L. Andrade and Patricia Machado. Generating Test Cases for Real-Time Systems Based on Symbolic Models. *IEEE Trans. Softw. Eng.*, 39(9):1216–1229, Sep 2013.

[ARVH15] T. Aicher, S. Rehberger, and B. Vogel-Heuser. Towards finding the appropriate level of abstraction to model and verify automated production systems in discrete event simulation. In *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1048–1053, Aug 2015.

[BBG+05] Sven Beyer, Peter Böhm, Michael Gerke, Mark Hillebrand, Tom In der Rieden, Steffen Knapp, Dirk Leinenbach, and Wolfgang J. Paul. Towards the Formal Verification of Lower System Layers in Automotive Systems. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 317–326. IEEE Computer Society, 2005.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[BDL+11] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing Uppaal over 15 years. *Software: Practice and Experience*, 41(2):133–142, 2011.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302. Springer, 1998.

[Ben02] Johan Bengtsson. *Clocks, DBM, and States in Timed Systems.* PhD thesis, Uppsala University, 2002.

[Bey01] Dirk Beyer. Improvements in BDD-Based Reachability Analysis of Timed Automata. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 318–343. Springer, 2001.

[BFGL+18] Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell. Model checking real-time systems. In Edmund M. Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1001–1046. Springer, 2018.

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, Compile, Run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BLP+99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 1999.

[BMPY97] Marius Bozga, Oded Maler, Amir Pnueli, and Sergio Yovine. Some Progress in the Symbolic Verification of Timed Automata. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 1997.

[Böh06] Peter Böhm. Implementation of the High-Level Components of a Bus Controller for a Time Triggered Serial Bus. Bachelor thesis, Universität des Saarlandes, 2006.

[Böh07] Peter Böhm. Formal Verification of a Clock Synchronization Method in a Distributed Automotive System. Master thesis, Universität des Saarlandes, 2007.

[BP06] Geoffrey M. Brown and Lee Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *TACAS*, volume 3920 of *LNCS*, pages 58–72. Springer, 2006.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Checker. *STTT*, 2(4):410–425, 2000.

[CE82] Edmund M. Clarke and E. Allen Emerson. *Logics of Programs: Workshop, Yorktown Heights, New York, May 1981*, chapter Design and synthesis of synchronization skeletons using branching time temporal logic, pages 52–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2001.

[Con00] Condor Engineering, Inc. *MIL-STD-1553 Tutorial*, 2000.

[Cor14] Maxime Cordy. *Model checking for the masses*. PhD thesis, University of Namur, 2014.

[Dil89] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1989.

[dW08] Gerit de Wagt. EMI-Hardened Operational Amplifiers for Robust Circuit Design. Application Note AN-1874, National Semiconductor Corporation, 2008.

[DWT95] David L. Dill and Howard Wong-Toi. Verification of Real-Time Systems by Successive Over and Under Approximation. In Pierre Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 1995.

[EFGP10] Rüdiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jörg Peter. Fully Symbolic Timed Model Checking Using Constraint Matrix Diagrams. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 360–371. IEEE Computer Society, 2010.

[EGP10] Rüdiger Ehlers, Michael Gerke, and Hans-Jörg Peter. Making the Right Cut in Model Checking Data-Intensive Timed Systems. In J.S. Dong and H. Zhu, editors, *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM 2010)*, volume 6447 of *Lecture Notes in Computer Science*, pages 565–580, Berlin Heidelberg, 2010. Springer-Verlag.

[EMP10] Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter. *Combining symbolic representations for solving timed games*. Springer, 2010.

[EMST10] Erik Endres, Christian Müller, Andrey Shadrin, and Sergey Tverdyshev. Towards the Formal Verification of a Distributed Real-Time Automotive System. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 212–216, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[FAWD+16] Sergio Feo-Arenis, Bernd Westphal, Daniel Dietsch, Marco Muñiz, Siyar Andisha, and Andreas Podelski. Ready for Testing: Ensuring Conformance to Industrial Standards Through Formal Verification. *Form. Asp. Comput.*, 28(3):499–527, may 2016.

[FBH+06] Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian Wohlgemuth, et al. Achievements and exploitation of the AUTOSAR development partnership. *Convergence*, 2006:10, 2006.

[Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1 Revision A*, 2005.

[Fle06a] FlexRay Consortium. *FlexRay Communications System Electrical Physical Layer Application Notes Version 2.1 Revision B*, 2006.

[Fle06b] FlexRay Consortium. *FlexRay Communications System Electrical Physical Layer Specification Version 2.1 Revision B*, 2006.

[Fle10a] Nic Fleming. Toyota car recall sparks 'drive by wire' concerns. *New Scientist*, February 2, 2010. https://www.newscientist.com/article/dn18485-toyota-car-recall-sparks-drive-by-wire-concerns/ accessed April 21, 2019.

[Fle10b] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 3.0.1*, 2010.

[Fur17] Steve Furber. Microprocessors: the engines of the digital age. *Proc Math Phys Eng Sci*, 473(2199):20160893–20160893, Mar 2017. PMC5378251[pmcid].

[GBHG14] Alexander Graf-Brill, Holger Hermanns, and Hubert Garavel. A Model-Based Certification Framework for the EnergyBus Standard. In Erika Ábrahám and Catuscia Palamidessi, editors, *34th Formal Techniques for Networked and Distributed Systems (FORTE)*, volume LNCS-8461 of *Formal Techniques for Distributed Objects, Components, and Systems*, pages 84–99, Berlin, Germany, June 2014. Springer. Part 2: Monitoring and Testing.

[GEFP10] Michael Gerke, Rüdiger Ehlers, Bernd Finkbeiner, and Hans-Jörg Peter. Model Checking the FlexRay Physical Layer Protocol. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 6371 of *Lecture Notes in Computer Science*, pages 132–147. Springer-Verlag, 2010.

[GEFP12a] Michael Gerke, Rüdiger Ehlers, Bernd Finkbeiner, and Hans-Jörg Peter. Automatic Protocol Verification with Parametric Physical Layers. Reports of SFB/TR 14 AVACS 86, SFB/TR 14 AVACS, 2012. ISSN: 1860–9821, http://www.avacs.org.

[GEFP12b] Michael Gerke, Rüdiger Ehlers, Bernd Finkbeiner, and Hans-Jörg Peter. FlexRay for Avionics: Automatic Verification with Parametric Physical Layers. In *AIAA Infotech@Aerospace (I@A 2012)*. American Institute of Aeronautics and Astronautics, 2012.

[Ger05] Michael Gerke. Flex Ray: Coding and Decoding, Media Access Control, Frame and Symbol Processing and Serial Interface. Seminar report, Institut für Rechnerarchitektur und Parallelrechner, Universität des Saarlandes, 2005. URL: http://www-wjp.cs.uni-sb.de/lehre/seminar/ss05/reports/gerke-report.pdf.

[Ger07] Michael Gerke. Implementation of Frame and Symbol Transmission in a Time Triggered Serial Bus Architecture. Bachelor thesis, Universität des Saarlandes, 2007.

[Ger10] Michael Gerke. Zone State Diagrams. Master thesis, Universität des Saarlandes, 2010.

[Gil18] Martin Giles. At Least Three Billion Computer Chips Have the Spectre Security Hole. *MIT Technology Review*, January 5, 2018. https://www.technologyreview.com/s/609891/at-least-3-billion-computer-chips-have-the-spectre-security-hole/ accessed April 20, 2019.

[GLLN18] Kim Guldstrand Larsen, Florian Lorber, and Brian Nielsen. 20 years of real real time model validation. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 22–36. Springer, 2018.

[GLYA13] Xiaoyun Guo, Hsin-Hung Lin, Kenro Yatake, and Toshiaki Aoki. An UPPAAL Framework for Model Checking Automotive Systems with FlexRay Protocol. In *Second International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2013)*, page 67, 2013.

[Guo12] Xiaoyun Guo. Model Checking of FlexRay Communication Protocol. Master thesis, Japan Advanced Institute of Science and Technology, 2012.

[HNSY94] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HR09] Christoph Heller and Reinhard Reichel. Enabling FlexRay for Avionic Data Buses. In *IEEE/AIAA 28th Digital Avionics Systems Conference (DASC)*, 2009.

[HWFA+12] Christian Herrera, Bernd Westphal, Sergio Feo-Arenis, Marco Muñiz, and Andreas Podelski. Reducing Quasi-equal Clocks in Networks of Timed Automata. In *Proceedings of the 10th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'12, pages 155–170, Berlin, Heidelberg, 2012. Springer-Verlag.

[KDL86] Uwe Kiencke, Siegfried Dais, and Martin Litschel. Automotive serial controller area network. In *SAE Technical Paper*. SAE International, 02 1986.

[KG93] H. Kopetz and G. Grunsteidl. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533, June 1993.

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardware design: Formaler Entwurf digitaler Schaltungen*, volume 15 of *Teubner-Texte zur Informatik*. Teubner, 1995.

[KP07] Steffen Knapp and Wolfgang Paul. Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *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*, pages 53–81. Springer, 2007.

[Kri12] Michael Krigsman. Worldwide cost of IT failure (revisited): $3 trillion. *ZDNet*, April 10, 2012. https://www.zdnet.com/article/worldwide-cost-of-it-failure-revisited-3-trillion/ accessed April 21, 2019.

[LG14] François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM.

[LLPY97] Kim G Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.

[LMM+12] Qi Lu, Michael Madsen, Martin Milata, Sren Ravn, Uli Fahrenberg, and Kim G. Larsen. Reachability analysis for timed automata using max-plus algebra. *The Journal of Logic and Algebraic Programming*, 81(3):298 – 313, 2012.

[LWYP98] Kim G Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock difference diagrams. *BRICS Report Series*, 5(46), 1998.

[Män98] R. Männer. Metastable states in asynchronous digital systems: Avoidable or unavoidable? *Microelectronics Reliability*, 28(2):295–307, 1998.

[MGB11] Christian Miller, Karina Gitina, and Bernd Becker. Bounded Model Checking of Incomplete Real-time Systems Using Quantified SMT Formulas. In *Proceedings of the 2011 12th International Workshop on Micro-*

processor Test and Verification, MTV '11, pages 22–27, Washington, DC, USA, 2011. IEEE Computer Society.

[MLAH99] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams. *Electr. Notes Theor. Comput. Sci.*, 23(2), 1999.

[MN10] J. Malinský and J. Novák. Verification of FlexRay start-up mechanism by timed automata. *Metrology and Measurement Systems*, Vol. 17, nr 3:461–480, 2010.

[MOR+04] Leonardo Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, chapter SAL 2, pages 496–500. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[MP11] Christian Müller and Wolfgang Paul. Complete Formal Hardware Verification of Interfaces for a FlexRay-Like Bus. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 633–648. Springer, 2011.

[MPS11] Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully Symbolic Model Checking for Timed Automata. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 616–632, Berlin, Heidelberg, 2011. Springer-Verlag.

[MS12] G. Morbé and C. Scholl. Guaranteeing Termination of Fully Symbolic Timed Forward Model Checking. In *13th International Workshop on Microprocessor Test and Verification (MTV)*, pages 35–40. IEEE Computer Society, December 2012.

[MS13] G. Morbé and C. Scholl. Fully Symbolic TCTL Model Checking for Incomplete Timed Systems. In H. Treharne and S. Schneider, editors, *Automated Verification of Critical Systems 2013 (AVoCS)*, volume 66, Guildford, Surrey, United Kingdom, 2013. EASST.

[MS14] Georges Morbé and Christoph Scholl. Fully Symbolic TCTL Model Checking for Complete and Incomplete Real-Time Systems. Reports of SFB/TR 14 AVACS 104, SFB/TR 14 AVACS, September 2014. http://www.avacs.org.

[MS15] Georges Morbé and Christoph Scholl. Fully Symbolic TCTL Model Checking for Complete and Incomplete Real-time Systems. *Sci. Comput. Program.*, 111(P2):248–276, nov 2015.

[Mül11] Christian Müller. *Complete Formal Hardware Verification of Interfaces for a FlexRay-like Bus*. PhD thesis, Universität des Saarlandes, 2011.

[Nan09] Nangate Inc. *Nangate 45nm Open Cell Library Databook*, 2009.

[Neu13] Stefan Neumann. *Modular Timing Analysis of Component-Based Real-Time Embedded Systems*. PhD thesis, Hasso Plattner Institute at the University of Potsdam, 2013.

[Nie81] Paul Nielsen. EMP/EMI Hardening of Electrical Conduit Systems. Technical Report CERL-TR-M-292, Construction Engineering Research Laboratory of the United States Army Corps of Engineers, September 1981.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[OD08] E.R. Olderog and H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 2008. doi:10.1017/CBO9780511619953.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb 1995.

[Pau94] Lawrence C Paulson. Isabelle-A Generic Theorem Prover (with a contribution by T. Nipkow), volume 828 of Lecture Notes in Computer Science, 1994.

[PB61] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, Jan 1961.

[PEM11] Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In *Computer Aided Verification*, pages 649–655. Springer, 2011.

[PH08] Michael Paulitsch and Brendan Hall. FlexRay in Aerospace and Safety-Sensitive Systems. *IEEE Aerospace and Electronic Systems Magazine*, 23(9):4–13, 2008.

[PSD06] Florian Pigorsch, Christoph Scholl, and Stefan Disch. Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling. In *FMCAD*, pages 89–96, 2006.

[SB03] Sanjit A. Seshia and Randal E. Bryant. Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 154–166. Springer, 2003.

[Sch06] Julien Schmaltz. A Formal Model of Lower System Layers. In *Formal Methods in Computer Aided Design (FMCAD'06)*, pages 191–192. IEEE Computer Society, 2006.

[Sch07] J. Schmaltz. A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In J. Baumgartner and M. Sheeran, editors, *7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 223–230. IEEE Press Society, November 11–14 2007.

[Sch11] Mareike Dorothee Schmidt. *Formal verification of a small real-time operating system.* PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2011.

[Sch16] Klaus Schwab. *The fourth industrial revolution.* Crown Publishing, 2016. LCCN 2016032826.

[SL02] J. Srinivasan and K. Lundqvist. Real-time Architecture Analysis: A COTS Perspective. In *Digital Avionics Systems Conference, 2002*, volume 1, pages 5D4–1 – 5D4–9, 2002.

[Som09] Fabio Somenzi. CUDD: CU Decision Diagram Package Release 2.4.2, 2009.

[SSL+92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[TA08] S. Tverdyshev and E. Alkassar. Efficient Bit-Level Model Reductions for Automated Hardware Verification. In *Temporal Representation and Reasoning, 2008. TIME '08. 15th International Symposium on*, pages 164–172, June 2008.

[Tom00] James E Tomayko. *Computers take flight: A history of NASA's pioneering digital Fly-by-wire project.* Number NASA SP-2000-4224 in The NASA history series. NASA, 2000. LCCN 99047421.

[Tra16] Thanh Tung Tran. *Verification of timed automata : reachability, liveness and modelling.* Theses, Université de Bordeaux, Nov 2016.

[Tur99] Jim Turley. Embedded Processors by the Numbers. *Electronic Engineering Times*, January 5, 1999. https://www.eetimes.com/author.asp?doc_id=1287712 accessed April 20, 2019.

[Tve05] Sergey Tverdyshev. *Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005. Proceedings*, chapter Combination of Isabelle/HOL with Automatic Tools, pages 302–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Universität des Saarlandes, 2009.

[TVM15] Tamás Tóth, András Vörös, and István Majzik. A Decomposition Method for the Verification of a Real-Time Safety-Critical Protocol. In *Proceedings of the 7th International Workshop on Software Engineering for Resilient Systems - Volume 9274*, SERENE 2015, pages 31–45, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[VG06] F.W. Vaandrager and A.L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. *Formal Aspects of Computing Journal*, 18(4):433–458, December 2006.

[Wan04] Farn Wang. Efficient verification of timed automata with BDD-like data structures. *STTT*, 6(1):77–97, 2004.

[WJ14] Weifeng Wang and Li Jiao. Trace abstraction refinement for timed automata. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, pages 396–410. Springer International Publishing, 2014.

[WKTZ05] Xu Wang, Marta Z. Kwiatkowska, Georgios K. Theodoropoulos, and Qianyi Zhang. Towards a Unifying CSP approach to Hierarchical Verification of Asynchronous Hardware. *Electr. Notes Theo. Comp. Sci.*, 128(6):231–246, 2005.

[YN04] Satoshi Yamane and Kazuhiro Nakamura. Development and evaluation of symbolic model checker based on approximation for real-time systems. *Systems and Computers in Japan*, 35(10):83–101, 2004.

[Yov97] Sergio Yovine. KRONOS: A Verification Tool for Real-Time Systems. *STTT*, 1(1-2):123–133, 1997.