Zone State Diagrams

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master Thesis

Michael Gerke

micger@react.cs.uni-sb.de

Saarbrücken, November 2010

Supervisor

Prof. Bernd Finkbeiner, Ph.D.

Advisor

Dipl. Inform. Hans-Jörg Peter

Reviewers

Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm

Einverständniserklärung

Ich bin damit einverstanden, dass diese Arbeit in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, 29. November 2010

"The complexity that we despise
is the complexity that leads to difficulty."
(Ward Cunningham,
"The Simplest Thing that Could Possibly Work",
http://www.artima.com/intv/simplest.html)

Danksagung

Ich möchte meiner Familie für ihre Unterstützung und Geduld danken.

Mein besonderer Dank gilt Prof. Finkbeiner, dessen unschätzbare Ratschläge das Entstehen dieser Arbeit erst ermöglicht haben.

Ausserdem möchte ich meinem Betreuer Hans-Jörg Peter danken, der immer da war, um mir mit seiner Meinung oder guten Tipps weiterzuhelfen. Vielen Dank auch an Rüdiger Ehlers, der mir seine boolean function abstraction library zur Verfügung gestellt hat.

Abstract. This thesis presents a model of FlexRay's physical layer protocol and a suitable data structure for model checking such models.

The FlexRay standard, developed by a cooperation of leading companies in the automotive industry, is a robust communication protocol for distributed components in modern vehicles. The key challenge in the analysis is that the correctness of FlexRay's physical layer protocol relies on the interplay of the bit-clock alignment mechanism with the precise timing behavior of the underlying asynchronous hardware. This thesis presents a timed automata model of the physical layer protocol and the underlying hardware.

Model checking such data-intensive systems using a semi-symbolic state space representation, which represents timing information symbolically but discrete information explicitly, turns out to be infeasible. To overcome this problem, this thesis presents Zone State Diagrams (ZSDs), a novel data structure combining difference bound matrices with reduced ordered binary decision diagrams. In terms of the size of the instances that can be handled, the performance of a model checker using ZSDs is better than the standard semi-symbolic approach on several benchmarks and the FlexRay model, and sometimes even better than Uppaal, e.g., in the case of the FlexRay model. While Uppaal is the fastest on all benchmarks, ZSDs are often a bit slower than the standard semi-symbolic approach.

ZSDs represent the discrete part of the state space symbolically as well as the timed part. Thus data-intensive state spaces can be more efficiently represented with ZSDs than with semi-symbolic state space representations, as used, e.g., by Uppaal.

Contents

List of Figures

Chapter 1

Introduction

This thesis explores a novel approach for storing the state space during timed reachability model checking of systems with a small number of clocks but a huge number of discrete states. The example for such a system used in this thesis is the physical layer protocol of the FlexRay vehicle bus protocol. The model of this protocol incorporates asynchronous behavior and a complicated communication procedure. Thus, manual analysis of the model is difficult due to the complexity of the interactions in the model. However, a model checking approach can handle this complexity, given the appropriate algorithms and data structures.

First, a short introduction to timed model checking and a description of the problem considered in this thesis and the proposed solution is given in the remainder of this Chapter. Chapter 2 reviews the concepts needed for the understanding of timed model checking, like timed automata, and explains the standard approach for representing the timed state space. The notion of data-intensive real-time systems is introduced in Chapter 3. Chapter 4 proposes an alternative timed state space representation, zone state diagrams, designed specifically for data-intensive real-time systems. The FlexRay protocol is described in Chapter 5, and a model of a data-intensive real-time system, FlexRay's physical layer protocol, is described in Chapter 6. Chapter 7 compares the performance of timed state sets and zone state diagrams on the task of model checking the model of FlexRay's physical layer protocol and on the Fischer, Gear Production Stack and Leader Election benchmarks. Chapter 8 sums up the obtained results and discusses possibilities for further investigation of the idea behind zone state diagrams and of further improvement of methods for model checking data-intensive real-time systems. Finally, Appendix A briefly describes a variant of model of FlexRay's physical layer protocol which is of interest for investigation of FlexRay's physical layer protocol.

1.1 Discrete Model Checking

Model checking is a formal method based on a model of a system and is used to find errors and to prove properties of the modeled system. According to Baier and Katoen [BK08, p 8],

> "Model checking is a verification technique that explores all possible system states in a brute force manner."

or, as they define it more precisely [BK08, p 11],

> "*Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*"

The model of the system consists of the discrete *states* the system can be in, and the *transition relation*, i.e., a collection of rules describing how the systems goes from one state to an other state. Variables over a discrete domain as well as the control location of the system are all encoded in the discrete state. This thesis will only consider reachability model checking, e.g., properties of the form "The system does never reach state X" will be proven.

1.2 Timed Model Checking

In timed model checking, not only the order in which discrete states are visited is important, but also the exact point in time. The model of the system is enriched with variables that measure time, so called *clocks*. A state of a timed system is a pair containing the discrete state, called *location*, and the valuation of the clocks. The transition relation can now also refer to clocks, i.e., it contains rules how to get from one location to the other at what time. It is also possible to restrict the time that the system can stay in a certain location.

1.3 Complex Systems

Model checking turns out to be quite useful for systems of low complexity, as it does find errors humans would easily overlook even in simple systems. Its real strength is the ability to also cover the cases nobody would have thought of when testing, as model checking exhaustively explores every possible behavior of the system. However, when model checking is used to investigate very complex systems or systems with many discrete variables that cannot be easily abstracted from, the technique eventually runs into trouble if the number of possibilities is too high and cannot be handled anymore in reasonable time using a reasonable amount of memory [LCL88].

1.3.1 Timed State Space Explosion

Models of very complex systems can often not be exhaustively explored in an efficient way. Techniques like, e.g., symmetry reduction and symbolic representation are used to overcome this problem, but nevertheless many safety-relevant systems are too complex to be efficiently model checked. In pure discrete model checking, the use of reduced ordered binary decision diagrams (BDDs), which are described in Section 2.1, enabled a huge increase in the size and complexity of models that can be checked [BCM$^+$92].

However, if the system can only be modeled in a way that leads to an exponential blowup of the reachable state space, this so-called *state space explosion* still limits the practical use of model checking.

For timed model checking the situation is even worse. The use of symbolic representations for the timing aspects of the system, e.g., *clock zones* [Alu99], is imperative, but still the complexity of systems that can be handled is very limited. Early models of the FlexRay physical layer protocol described in Chapter 5 could not be verified at all using the state-of-the-art timed model checker UPPAAL due to lack of memory. After some refinement of the model using abstraction to reduce the state space, UPPAAL was able to verify safety properties for simple instances of the model described in Chapter 6.

1.3.2 Idea: Reverse Hierarchy

A thorough discussion of the difficulties of UPPAAL to check the FlexRay model highlighted inefficient space usage by UPPAAL. A model like the one from Chapter 6 has a very large discrete part and just a very limited real time related behavior. If the timed state space is stored in a way similar to *timed state sets* as described in Section 2.6, every location is extended with timing information, i.e., a collection of clock zones. In a system with many discrete locations and only a limited number of different clock zones, these zones are thus stored multiple times. In the naive approach, having 1,000,000 locations and just 1000 different clock zones means that at least 999,000 stored clock zones are copies of already stored clock zones.

For systems expected to have a larger number of locations than clock zones, this redundant storage leads to unnecessary memory consumption. However, expanding locations with timing information is a natural approach, as a systems behavior is described in terms of what it does if it is in a certain location and certain restrictions on the timing aspects hold. A location extended with timing information thus represents the common view on the behavior of systems.

To exploit the characteristics of such a system, i.e., the large discrete part and the small timing part, this thesis presents a novel approach that significantly reduces space consumption in the timed state space representation. The timed state space is represented the other way round: The timing information is extended with the locations, as realized by *zone state diagrams*, introduced in Chapter 4. This can significantly reduce redundant storage if many different locations have the same timing requirements, as it is the case in the model described in Chapter 6. The reversed paradigm does of course not guarantee that redundancy is reduced in all cases, as the number of location / clock zone pairs is still the same. But it does guarantee that every zone is stored at most once, and only locations are stored multiple times. Huge sets of locations, in turn, can be efficiently stored using BDDs. The use of BDDs to store sets of locations will usually reduce the amount of memory needed dramatically. However, off the shelf BDD libraries cannot be used in a mapping from locations to sets of clock zones, they can only be used if the direction of the mapping is reversed, i.e., if clock zones are mapped to sets of locations.

As the exploration of the model is still based on the locations extended by timing

information, a certain computational overhead is introduced by reversing the order, as the information about the timed state of the model has to be converted to the new format in each step. However, a comparison of the results from Table 7.3 and Table 7.2 in Chapter 7 indicates that this overhead is small while the efficiency gain is huge when model checking the FlexRay model.

Chapter 2

Preliminaries

Contents

2.1 Binary Decision Diagrams

For representing sets of bit vectors, *reduced ordered binary decision diagrams* (BDDs) [Bry86] are very efficient[BCM+92]. Basically, a BDD is a directed acyclic graph similar to a binary decision tree, with a total order on the variables occurring on the paths from the root to a leaf. The nodes are labeled with variable names, while the corresponding two outgoing edges are labeled with 1 and 0. There are two leaves, one labeled with **true** and one labeled with **false**. A valuation of variables is contained in a BDD that uses no other variables, if and only if there is a path from the root to the **true** leaf such that at every node, the edge corresponding to the value of the variable this node

is labeled with is taken. If the valuation is not contained in the BDD, then there is such a path starting at the root and ending in the **false** leaf.

The graph is organized such that valuations that are identical for the first i variables and valuations that are identical from the j^{th} variable on have paths that share the corresponding edges.

Moreover, all path in a BDD either end in the **false** leaf or in the **true** leaf, so edges leading to a subgraph where the **true** leaf is no longer reachable can be redirected to the **false** leaf immediately, and the subgraph can be removed. Vice versa, all edges leading into a subgraph were the **false** leaf is no longer reachable can be redirected to the **true** leaf, which also allows to remove now unreachable parts of the graph, thus reducing its size. If both edges that leave a node go to the same node, the value of the variable the node is labeled with does not influence the rest of the path, so the value does not have to be checked, and the respective node can just be removed, redirecting all incoming edges to the node that was the target of the two outgoing edges.

The size of a BDD strongly depends on the variable ordering.

Formally, BDDs represent binary functions $f : 2^V \to \mathbb{B}$ for some finite set of boolean variables V. Given two binary functions f and f', their disjunction is defined as $(f \vee f')(x) = f(x) \vee f'(x)$ for all $x \subseteq V$.

Taking the individual bits of the vector as assignments for the variables in V, a BDD representing function f contains a bit vector x if $f(x) = true$ and does not contain this vector if $f(x) = false$. So if a bit vector x' is added to a BDD bdd representing f, the resulting BDD $bdd' = bdd \vee x'$ represents the binary function f' that is defined by

$$f'(x) = \begin{cases} true & : x = x' \\ f(x) & : x \neq x' \end{cases}$$

.

2.2 Timed Automata

A popular formalism for modeling components of timed systems are timed automata [AD90, AD94]. Clarke et al. start their detailed description of timed automata in [CGP99, p 265] with the short characterization:

> "A *timed automaton* is a finite automaton augmented with a finite set of real-valued clocks."

In a location of the automaton, time can elapse, i.e., if time elapses, all clocks increase at the same rate.

When the automaton transitions from one location to another, some of the clocks can be reset to to zero. The *transitions* between two locations can also be equipped with a *guard*, i.e., a constraint on the clock valuations that determines whether the transition is *enabled* and can be taken or not.

The locations can be equipped with *invariants* that restrict the clock valuations that may be reached in this location. The location has to be left before this invariant

is violated and cannot be entered if the valuations of the clocks do not satisfy the invariant, thus disabling the respective incoming transition.

2.2.1 Syntax

With X being a finite set of *clock variables*, ranging over the nonnegative reals ($\mathbb{R}_{\geq 0}$), the set of *clock constraints* over X, $\mathcal{C}(X)$, is defined as follows: Let $x \in X$ be a clock variable and let $c \in \mathbb{N}_0$ be a constant integer, a clock constraint $\varphi \in \mathcal{C}(X)$ is of the form

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

where $\varphi_1, \varphi_2 \in \mathcal{C}(X)$.

Formally, a *timed automaton* is a 6-tuple $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ where

- L is a finite set of control *locations*,
- $l_0 \in L$ is the unique *initial location* of the automaton,
- $I : L \to \mathcal{C}(X)$ maps each location to an *invariant*, i.e., a clock constraint that has to hold while the automaton is in the corresponding location,
- Σ is a finite set of *actions*,
- $\Delta \subseteq L \times \Sigma \times \mathcal{C}(X) \times 2^X \times L$ is a set of *edges*, The 5 tuple $\delta = \langle l, a, \varphi, \lambda, l' \rangle \in \Delta$ is an edge from location l to location l' with action a, *guard* φ, i.e., a clock constraint that has to be satisfied to enable the edge, and a set λ of clocks to be *reset* to zero when the edge is taken, and
- X is a finite set of nonnegative real-valued *clocks*.

2.2.2 Semantics

A *clock valuation* $\vec{v} : X \to \mathbb{R}_{\geq 0}$ maps each clock to a nonnegative real value. The set of all clock valuations over X is denoted by $\mathcal{V}(X)$. If the clocks are numbered from 1 to $|X|$, $\vec{v} \in \mathcal{V}(X)$ can be represented as a vector with $|X|$ dimensions by assigning the value of the i^{th} clock to the i^{th} dimension. Resetting clock $x \in X$ to zero in $\vec{v} \in \mathcal{V}(X)$ results in the new valuation $\vec{v}[x := 0]$. $\vec{v}[\lambda := 0]$ is obtained by resetting all clocks in $\lambda \subseteq X$ to zero in $\vec{v}$. For $d \in \mathbb{R}_{\geq 0}$, the valuation $\vec{v} + d$ assigns to each clock $x \in X$ the value $\big(\vec{v} + d\big)(x) := \vec{v}(x) + d$.

A (timed) *state* of a timed automaton is a pair $(l, \vec{v})$ of control location $l \in L$ and clock valuation $\vec{v} \in \mathcal{V}(X)$. The initial valuation is the zero vector $\vec{0}$. Thus, the initial state of the automaton is $(l_0, \vec{0})$. The automaton can change its state using two types of transitions.

The first type of transition, the so called *discrete transition*, uses an enabled edge to get to a different control location, not changing the clock valuation. An edge $\langle l, a, \varphi, \lambda, l' \rangle \in \Delta$ is *enabled* in state $(l'', \vec{v})$ if and only if the automaton is in the right location ($l'' = l$), the valuation satisfies the guard ($\vec{v} \vDash \varphi$), and the valuation after

execution of the discrete transition using this edge satisfies the invariant of the new location ($\vec{v}[\lambda := 0] \vDash I(l')$). If the enabled discrete transition using edge $\langle l, a, \varphi, \lambda, l' \rangle$ is executed in state $(l, \vec{v})$, the resulting state is $(l', \vec{v}[\lambda := 0])$.

There is a second type of transitions, the so called *delay transitions* corresponding to the elapsing of time in a location. A delay transition with delay d is enabled in state $(l, \vec{v})$ if and only if the invariant of location l is not violated during the elapsing of time, i.e.,$\forall 0 \leq d' \leq d \; . \; \vec{v} + d' \vDash I(l)$. As the invariants are convex, this requirement is equivalent with $\vec{v} \vDash I(l) \wedge \vec{v} + d \vDash I(l)$. If an enabled delay transition with delay d is executed in state $(l, \vec{v})$, the resulting state is $(l, \vec{v} + d)$. Note that for model checking purposes, infinite *zeno*, i.e., time converging, sequences of transitions are not allowed.

2.3 Networks of Timed Automata

To describe timed systems, several timed automata can be composed to a network.

The composition of the timed automaton $\mathcal{A}_1 = (L_1, l_0^1, I_1, \Sigma_1, \Delta_1, X_1)$ and the timed automaton $\mathcal{A}_2 = (L_2, l_0^2, I_2, \Sigma_2, \Delta_2, X_2)$ with a disjoint set of clocks ($X_1 \cap X_2 = \emptyset$) is defined as the timed automaton $\mathcal{A}_1 || \mathcal{A}_2 = (L_1 \times L_2, (l_0^1, l_0^2), I, \Sigma_1 \cup \Sigma_2, \Delta, X_1 \cup X_2)$ where $I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ and the transition relation Δ contains all transitions formed according to the following rules:

- If $a \in \Sigma_1 \cap \Sigma_2$ then for all $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$ and for all $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$, $\langle (l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2') \rangle \in \Delta$.
- If $a \in \Sigma_1 \setminus \Sigma_2$ then for all $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$ and for all $l_2 \in L_2$, $\langle (l_1, l_2), a, \varphi_1, \lambda_1, (l_1', l_2) \rangle \in \Delta$.
- If $a \in \Sigma_2 \setminus \Sigma_1$ then for all $l_1 \in L_1$ and for all $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$, $\langle (l_1, l_2), a, \varphi_2, \lambda_2, (l_1, l_2') \rangle \in \Delta$.

The *global timed automaton* representing the whole system can be formed by incrementally composing the timed automata that represent the individual components. Note that for model checking purposes, the global timed automaton does not necessarily need to be constructed completely. In practice, model checkers use on-the-fly techniques to explore the reachable product states until the property can be proven or disproven.

2.3.1 Extended Automata Syntax

An extension to the classical definition of timed automata are *extended timed automata* [BDL04]. The syntax used to describe the variant of extended timed automata used in the model described in Section 6 is similar to the syntax used by the model checking tool UPPAAL as described in [BDL04] with some minor changes.

For the extended automata used in Section 6, three concepts are added to the timed automata concept:

1. Integer and boolean *variables* that have a specified range and thus just a finite number of values, are part of the state space. The values of these variables can be referred to in guards and can be updated when edges are taken.

2. Actions are replaced by so called *synchronization channels* using *broadcast synchronization*. If an edge has no *synchronization* label, then it is treated like an edge with an action that no other edge in a different automaton uses, i.e., it does not need to synchronize with any other edge. A synchronization label specifies the identifier of the synchronization channel and whether the edge is *sending* or *receiving* on that channel. A sending edge can be taken whenever it is enabled, independent of any synchronization. A receiving edge will be taken if and only if it is enabled and a sending edge using the same synchronization channel is taken at the same time, i.e., receiving edges have to synchronize to sending ones.

3. A location that is declared *committed* has to be left before time can pass or some edge not starting in a committed location can be taken.

All these three concepts can be expressed in terms of classical timed automata.

1. The values of integer and boolean *variables* can be encoded into extra control locations. The updates of variables can be encoded by directing the edge to the respective location encoding the new value. Likewise, if a constraint on a variable is used in a guard the respective edges do only start at locations that encode values satisfying the constraint.

2. *Synchronization channels* using *broadcast synchronization* can be encoded into the locations, edges, and actions of a timed automaton by building the full product automaton. The broadcast synchronization on a channel is simulated by individual edges for each combination of a guard from a sending edge with negated or not negated guards from all receiving edges. Checking all combinations of satisfied or unsatisfied guards allows to transition to the appropriate product location that represents the behavior that all enabled receiving edges were taken.

3. To simulate the behavior of *committed* locations, all incoming edges of committed locations reset some otherwise unused clock and the location gets the additional invariant that this clock may not become bigger than 0, thus preventing the passage of time. If an extra boolean variable is added indicating whether some automaton is in a committed location, this variable can be used to disable all edges not starting in a committed location.

Figure 2.1 shows the syntax used to describe locations and transitions.

- Synchronization is the optional broadcast channel on which this transition synchronizes. A "!" indicates that taking this transition forces all other enabled transitions that synchronize with this channel to be taken as well, making this transition broadcasting on this channel. A "?" indicates that this transition will be taken if and only if it is enabled and a transition which is broadcasting on the channel is taken.

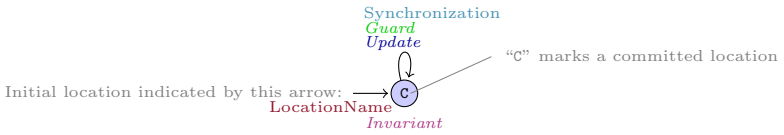

Figure 2.1: Syntax of locations and transitions

- *Guard* is a formula evaluating to a boolean value that indicates whether the transition is enabled.
- *Update* is a comma-separated list of updates of variables or resets of clocks that are executed when the transition is taken.
- *Invariant* is a logical formula that has to be true while the automaton stays in this location. This forces the automaton to leave this location before the *Invariant* is violated and disables any transition leading into a location with a violated *Invariant*.
- A *committed* location has to be left before any transition not leaving a committed location can be taken.

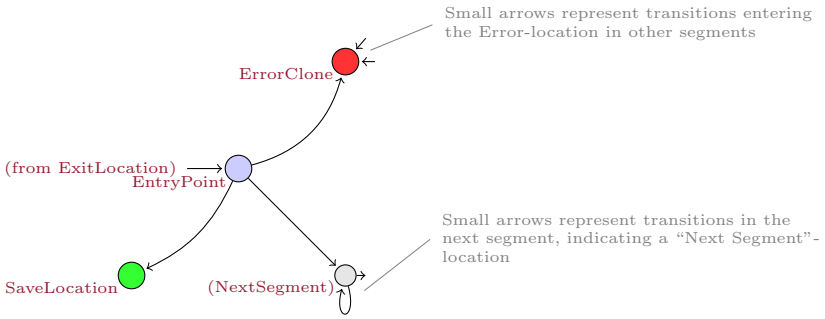

Figure 2.2: Syntax used to describe split automata

In order to make bigger automata easier to understand, they can be split into segments using the auxiliary syntax shown in Figure 2.2.

- NextSegment is a location representing the parts of the automaton reachable by the incoming transitions that are described in the next segment. It is indicated 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 "Next Segment"-location of the previous segment.

- ErrorClone is a location indicating an error state. It is reachable from various segments of the automaton, incoming transitions in other segments are indicated by small arrows. The location is marked in red.

- SaveLocation is a location from which no error state can be reached. The location is marked in green.

In the example shown in Figure 2.2, the location "EntryPoint" is also the "ExitLocation" of this segment, as there is a transition from EntryPoint to NextSegment.

2.4 Finite Representation

Recall that a state of a timed automaton is a pair consisting of a control location l and a clock valuation $\vec{v}$. As a valuation $\vec{v}$ maps clocks to $\mathbb{R}_{\geq 0}$, there can be uncountably many states of a timed automaton. In order to perform an exhaustive search on the reachable states of such an automaton, an equivalent symbolic representation that has a finite amount of reachable states is needed.

2.4.1 Clock Regions

If a valuation assigns a value to a clock that is bigger than the biggest constant used for comparison in some guard or invariant, the behavior of the automaton will not be influenced by the concrete value of that clock. A value of a clock is *maximal* if it is bigger than the biggest constant any clock is compared to in any guard or invariant. If a valuation assigns a maximal value to a clock, the actual value can be abstracted from.

Alur et al. [ACD90] proposed to partition the set of all clock valuations into a finite number of equivalence classes, the so called *clock regions*. Two valuations $\vec{v}_1, \vec{v}_2 \in \mathcal{V}(X)$ are in the same clock region if and only if all of the following conditions are fulfilled:

- Both valuations consider the values of the same clocks to be maximal, i.e., $\vec{v}_1(x)$ maximal if and only if $\vec{v}_2(x)$ maximal.

- Both valuations agree on the integer parts of the non-maximal clock values, i.e., $\lfloor \vec{v}_1(x) \rfloor = \lfloor \vec{v}_2(x) \rfloor$.

- Both valuations consider the non-integer parts of the same non-maximal clock values to be zero, i.e., $\lfloor \vec{v}_1(x) \rfloor = \vec{v}_1(x)$ if and only if $\lfloor \vec{v}_2(x) \rfloor = \vec{v}_2(x)$.

- Both valuations agree on the order in which the non-maximal clock values will change their integer part, i.e., $\vec{v}_1(x) - \lfloor \vec{v}_1(x) \rfloor \leq \vec{v}_1(x') - \lfloor \vec{v}_1(x') \rfloor$ if and only if $\vec{v}_2(x) - \lfloor \vec{v}_2(x) \rfloor \leq \vec{v}_2(x') - \lfloor \vec{v}_2(x') \rfloor$.

Two states $(l, \vec{v})$ and $(l', \vec{v}')$ are *region equivalent* if and only if (1) $l = l'$ and (2) $\vec{v}$ and $\vec{v}'$ are in the same clock region.

Regions are a useful finite abstraction as two region equivalent states $(l, \vec{v}_1)$ and $(l, \vec{v}_2)$ have two interesting properties that guarantee that every path in the original automaton is simulated in the region equivalence abstraction:

1. If there is an edge that allows to take a discrete transition from $(l, \vec{v}_1)$ to $(l', \vec{v}_1')$, then this edge also allows to take a discrete transition from $(l, \vec{v}_2)$ to $(l', \vec{v}_2')$ where $(l', \vec{v}_1')$ and $(l', \vec{v}_2')$ are region equivalent.

2. If there is a delay transition from $(l, \vec{v}_1)$ to $(l, \vec{v}_1')$, then there is also a delay transition from $(l, \vec{v}_2)$ to $(l, \vec{v}_2')$ such that $(l, \vec{v}_1')$ and $(l, \vec{v}_2)'$ are region equivalent.

Thus, using region equivalence, the infinite state space of a timed automaton can be abstracted to a symbolic state space with a finite number of states.

2.4.2 Clock Zones

Alur [Alu99] proposes *clock zones*, convex sets of clock valuations characterized by conjunctions of *clock difference constraints*. With $x_1, x_2 \in X$, $c \in \mathbb{Z}$, and $\prec \in \{<, \leq\}$, the constraints defining a clock zone are of the form (1) $x_1 \prec c$, (2) $c \prec x_1$, or (3) $x_1 - x_2 \prec c$. Since a clock never has a negative value and all values above the biggest constant are represented by the same value, the value of a clock is always bounded from below and from above. Constraints of the form (1) or (2) can, thus, always be written in the form $-c_{0,i} \prec x_i \prec c_{i,0}$. A special *zero clock* x_0 always having the value 0 is used to write all such constraints involving only one clock as the conjunction of two constraints of the form (3):

$$x_0 - x_i \prec c_{0,i} \wedge x_i - x_0 \prec c_{i,0}$$

Thus, the general form of a clock zone is

$$0 \leq x_0 \leq 0 \wedge \bigwedge_{0 \leq i \neq j \leq |X|} x_i - x_j \prec c_{i,j}.$$

As the zero clock is always 0, the constraint on the zero clock $0 \leq x_0 \leq 0$ is implicit and does not have to be mentioned in the description of a clock zone.

2.4.3 Difference Bound Matrices

Dill [Dil90] proposed *difference bound matrices* (DBM) to represent clock zones. Each entry $z_{i,j}$ of a matrix corresponds to the constraint $x_i - x_j \prec c_{i,j}$. The entry indicates which constant $c_{i,j} \in \mathbb{Z}$ is to be used in the constraint and whether the $\prec$ is $<$ or $\leq$. For example, the entry $z_{0,3} = (-2, <)$ represents the constraint $x_0 - x_3 < -2$. As the zero clock x_0 is always 0, this DBM entry expresses $x_3 > 2$.

Recall, if the biggest constant used in the automaton is exceeded by the value of a clock, the value of that clock is abstracted by ∞. If no upper bound for a clock difference is given, the entry in the DBM for that clock difference will be $(\infty, <)$, which is a safe over-approximation.

As an example, consider the clock zone

$$x_1 - x_2 \leq 1 \wedge 2 \leq x_1 \leq 3 \wedge x_2 < 3$$

which can be represented by the following DBM:

$+\backslash^-$	x_0	x_1	x_2
x_0	$(0, \leq)$	$(-2, \leq)$	$(\infty, <)$
x_1	$(3, \leq)$	$(0, \leq)$	$(1, \leq)$
x_2	$(3, <)$	$(\infty, <)$	$(0, \leq)$

However, this DBM representation of the zone is not unique. There are implicit constraints that are not mentioned. From $2 \leq x_1$ and $x_1 - x_2 \leq 1$ follows $1 \leq x_2$, a fact not mentioned by the DBM above. Thus, this information can be added to the DBM and the resulting DBM still describes the same zone:

$+\backslash^-$	x_0	x_1	x_2
x_0	$(0, \leq)$	$(-2, \leq)$	$(-1, \leq)$
x_1	$(3, \leq)$	$(0, \leq)$	$(1, \leq)$
x_2	$(3, <)$	$(\infty, <)$	$(0, \leq)$

In a similar manner, $x_2 - x_1 < 1$ can be deduced from $x_2 < 3$ and $2 \leq x_1$. Generally, the clock difference $x_i - x_j$ is bounded from above by the sum of the upper bounds of $x_i - x_k$ and $x_k - x_j$.

Formally, the rule is: If $x_i - x_k \prec_{i,k} c_{i,k}$ and $x_k - x_j \prec_{k,j} c_{k,j}$ then $x_i - x_j \prec_{i,j} c_{i,k} + c_{k,j}$ with $\prec_{i,j} = \leq$ if $(\prec_{i,k} = \leq) \wedge (\prec_{k,j} = \leq)$, and $\prec_{i,j} = <$ otherwise. If this reasoning comes up with a tighter bound than the one listed in the respective DBM entry, the DBM entry can be replaced by an entry describing the tighter bound. This process is called *tightening*. If all entries in the DBM are tightened until further tightening will not change the DBM any more, a canonical DBM representation of the zone is obtained. Let ζ denote the set of all canonical DBMs. The canonical representation of the example zone is:

$+\backslash^-$	x_0	x_1	x_2
x_0	$(0, \leq)$	$(-2, \leq)$	$(-1, \leq)$
x_1	$(3, \leq)$	$(0, \leq)$	$(1, \leq)$
x_2	$(3, <)$	$(1, <)$	$(0, \leq)$

Note that in canonical form, it is cheap to check if the zone represented by the DBM is empty, as a empty zone has some unsatisfiable constraints that lead to a tighter bound than $x_i - x_i \leq 0$ for some i in the canonical form of the DBM representing that zone. Thus, a canonical DBM can be checked for emptiness by examining the entries on the diagonal of the matrix. Moreover, it is easy to check for semantical equality of two canonical DBMs.

A valuation satisfies a DBM if the valuation satisfies all constraints encoded in the DBM. Let $z \in \zeta$, then $[\![z]\!] := \{\vec{v} \in \mathcal{V}(X) \mid \vec{v} \vDash z\}$.

In the following, canonical DBMs are used to represent clock zones on a technical level.

Intersection

As guards or invariants can be interpreted as zones, they can as well be described as DBMs. Thus intersection of DBMs is needed to analyze timed automata using DBMs.

The intersection of two zones has to satisfy the conjunction of the constraints of these two zones. Let the DBM z be the intersection of $z' \in \zeta$ and $z'' \in \zeta$. For every entry $z_{i,j}$, the tighter one of the two bounds represented by $z'_{i,j}$ and $z''_{i,j}$ is chosen. The resulting DBM has then to be tightened to its canonical form.

Clock Reset

For analyzing transitions of timed automata using DBMs, a method to reset clocks in DBMs is needed.

Let the DBM $z = z'[\lambda := 0]$ be the result of resetting all clocks in $\lambda \subseteq X$ to zero in $z' \in \zeta$. Then for all $0 \leq i, j \leq |X|$,

$$z_{i,j} = \begin{cases} (0, \leq) & x_i \in \lambda \wedge x_j \in \lambda \\ z'_{0,j} & x_i \in \lambda \wedge x_j \notin \lambda \\ z'_{i,0} & x_i \notin \lambda \wedge x_j \in \lambda \\ z'_{i,j} & \text{otherwise} \end{cases}$$

In case z and z' are empty, the entries $(0, \leq)$ for $z_{i,j}$ with $x_i, x_j \in \lambda$ might not be tight, thus the resulting DBM also has to be tightened to its canonical form.

Time Elapsing

For analyzing timed automata when time can elapse in a location, an operation to let time elapse in a DBM is needed.

Let the DBM $z = z'^{\Uparrow}$ be the result of letting time elapse in $z' \in \zeta$. Then for all $0 \leq i, j \leq |X|$,

$$z_{i,j} = \begin{cases} (\infty, <) & i \neq 0 \wedge j = 0 \\ z'_{i,j} & \text{otherwise} \end{cases}$$

Federations

A set of DBMs is called a *clock federation*, representing a set of zones and, thus, a set of clock valuations. A valuation satisfies a federation if it satisfies at least one of the DBMs contained in the federation. A clock federation can represent non convex sets of clock valuations: The set can be represented as a union of (convex) clock zones that, in turn, are represented as DBMs.

Formally, let ξ be a federation:

$$[\![\xi]\!] = \{\vec{v} \mid \exists z \in \xi . \vec{v} \in [\![z]\!]\}$$

The binary set operator union does extend to federations straightforward. Let ξ, ξ' be federations:

$$\xi \cup \xi' = \{z \mid z \in \xi \vee z \in \xi'\}$$

However, the intersection of two federations is defined as the set of all nonempty intersections of DBMs from the two federations, where only one of the two intersected DBMs is taken from each federation. Let ξ, ξ' be federations:

$$\xi \cap \xi' := \{z'' \mid \exists z \in \xi, z' \in \xi'.z'' = z \cap z' \wedge \neg empty(z'')\}$$

Finally, subtracting federation ξ' from federationξ results in removing all valuations of ξ' from ξ, which is defined as a federation ξ'' representing the remaining valuations. Let ξ, ξ', ξ'' be federations with $\xi'' = \xi \setminus \xi'$:

$$[\![\xi'']\!] := \{\vec{v} \mid \exists z \in \xi. \forall z' \in \xi'. \vec{v} \in [\![z]\!] \wedge \vec{v} \notin [\![z']\!]\}$$

2.4.4 Zone Graph

States of a timed automaton can be grouped into so called *zones*. A zone is a pair of a location l and a clock zone (represented by a DBM) z.

For an edge $\delta = \langle l, a, \varphi, \lambda, l' \rangle \in \Delta$ of a timed automaton and a zone (l, z), the clock zone that is obtained by executing the transition using edge δ and then letting time elapse is referred to as $succ(z, \delta)$. For the zone (l, z), $succ(z, \delta)$ is obtained by executing the following steps:

1. To find the clock valuations that enable δ, the clock zone is intersected with the guard φ: $z \wedge \varphi$.

2. The clocks in λ are reset in the result: $(z \wedge \varphi)[\lambda := 0]$

3. To exclude clock valuations that violate the invariant, the result is intersected with the invariant of the new location: $(z \wedge \varphi)[\lambda := 0] \wedge I(l')$.

4. Time is elapsed in the result: $\big((z \wedge \varphi)[\lambda := 0] \wedge I(l')\big)^{\Uparrow}$.

5. To exclude the new valuations that violate the invariant, the result is intersected with the invariant of the new location again.

Thus, $succ(z, \delta) = \big((z \wedge \varphi)[\lambda := 0] \wedge I(l')\big)^{\Uparrow} \wedge I(l')$ is obtained.

The *zone graph* is a transition system for the automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ with zones of $\mathcal{A}$ being states of the zone graph. The initial state of the zone graph is the zone formed by the initial location of $\mathcal{A}$ and a clock zone where all clocks have value 0, time has elapsed, and the invariant is fulfilled, $(l_0, [X := 0]^{\Uparrow} \wedge I(l_0))$. Note that $[X := 0] \vDash I(l_0)$ is required. For each edge $\delta = \langle l, a, \varphi, \lambda, l' \rangle \in \Delta$, the zone graph contains an edge from state (l, z) to state $(l', succ(z, \delta))$.

2.5 Reachability Model Checking

In order to answer the question if a system modeled by a timed automaton does ever reach a certain state, the zone graph of the automaton is explored. In so called *forward* reachability model checking, states that are reachable from the initial state are explored to check if a certain state is contained in a reachable state.

A state is reachable from another state, if there is a sequence of transitions leading from the latter to the former. With R being a set of states in a zone graph, let $post(R)$ be all states reachable from some state in R by taking exactly one edge in the zone graph. Formally, $post(R) = \{(l', z') \mid \exists (l, z) \in R : \exists \langle l, a, \varphi, \lambda, l' \rangle \in \Delta : succ(z, \langle l, a, \varphi, \lambda, l' \rangle) = z'\}$.

To find the set of reachable states, the *post* operator is iterated in order to find its least fixed point. Algorithm 1 gives a definition in pseudo code.

Algorithm 1 Basic algorithm for calculating the least fixed point of reachable states.

```
R_0 := initial states
i := 0
repeat
  i := i + 1
  R_i := R_{i-1} ∪ post(R_{i-1})
until R_i = R_{i-1}
```

Algorithm 2 shows a basic breadth first reachability check for a state with location l_{error} in pseudo code. It uses the fact that it is not necessary to apply *post* to all reachable states in every iteration, but only to the states that are newly found to be reachable.

2.6 Timed State Sets

Traditional timed model checkers like UPPAAL[BDL04] and KRONOS[DOTY95] use an explicit mapping from locations to timing information, e.g., as provided by timed state sets (TSSs). In this Section, a formal definition of the usual timed state space representation, TSSs, is given.

2.6.1 Definition

A TSS maps a location to a set of DBMs: $TSS : (L \to 2^{\zeta})$. A set of DBMs is called a *federation*. Let Θ denote the set of all TSSs.

The empty TSS, θ_0, maps every location to the empty set of DBMs:

$$\forall l \in L : \theta_0(l) = \emptyset$$

A new TSS can be formed by applying the mapping operation $[l \mapsto z]$ to a TSS, which adds the DBM z to the set of DBMs to which the TSS maps the location l. Consider

Algorithm 2 Basic breadth first exploration of the reachable state space checking the reachability of error location l_{error}.

```
R := initial states
FIFOqueue := empty queue
for all (l, z) ∈ R do
  enqueue (l, z) in FIFOqueue
end for
repeat
  dequeue (l, z) from FIFOqueue
  for all (l', z') ∈ post({(l, z)}) do
    if l' ≠ l_error then
      if (l', z') ∉ R then
        enqueue (l', z') in FIFOqueue
        R := R ∪ {(l', z')}
      end if
    else
      return l_error reachable
    end if
  end for
until FIFOqueue = empty queue
return l_error unreachable
```

a TSS $\theta' = \theta[l \mapsto z]$, with $z \in \zeta$ and $l, l' \in L$:

$$\theta'(l') := \begin{cases} \{z\} \cup \theta(l') & \text{if } l' = l \\ \theta(l') & \text{else} \end{cases}$$

Finally, the set of location-valuation pairs represented by a TSS is defined as

$$[\![\theta]\!] := \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \theta(l).\vec{v} \vDash z\}.$$

2.6.2 Operations

In this Section, the binary set operators are defined for TSSs in a straightforward manner, using the definitions for federations from Section 2.4.3. Let $l \in L$, and $\theta, \theta' \in \Theta$:

$$(\theta \cup \theta')(l) := \theta(l) \cup \theta'(l)$$
$$(\theta \cap \theta')(l) := \theta(l) \cap \theta'(l)$$
$$(\theta \setminus \theta')(l) := \theta(l) \setminus \theta'(l)$$

Part I

Zone State Diagrams

Chapter 3

Data-Intensive Real-Time Systems

Models describing real world systems tend to consist of a large number of locations, depending on the level of abstraction. If these models cannot describe the system with an adequate accuracy in a purely discrete manner, formalisms like hybrid automata or timed automata can provide the desired expressivity.

Model checking large networks of timed automata is hard. This thesis is going to present an approach that allows to model check a certain class of large network of timed automata, namely *data-intensive real-time systems*[EGP10]. A data-intensive real-time system is a network of timed automata that has many locations (recall that "location" includes integer variables), but only few clocks.

Controllers of real world systems are implemented in hardware or software that basically is a state based version of some algorithm used to control the system. The environment the controller of the system has to deal with can also often be abstracted to a state based model to a certain extent. Thus, data-intensive real-time systems are common real world examples, as a large part of many real world systems can be modeled as a discrete automaton, and only small parts cannot be modeled without modeling real time. The model of the physical layer protocol used by the FlexRay automotive bus protocol shown in Chapter 6 is a witness of a data-intensive real-time system: The model uses just two clocks, but many discrete variables and lots of locations. This is because the operation of the protocol can be completely modeled in a discrete fashion, and only the model of the underlying hardware introduces the need to use timed automata, as the hardware contains oscillators that cannot be modeled in a purely discrete way.

Initial attempts to model check early variants of the FlexRay model with Uppaal showed that the state space arising from a exploration of the model was huge and difficult to handle for Uppaal. As described in Section 2.6, Uppaal uses an explicit mapping from locations to sets of DBMs to represent the state space. This has a drawback when representing data-intensive real-time systems: If the few clocks just give rise to a number of clock zones that is smaller than the number of locations (the

value of discrete variables is also encoded into locations), many locations will map to sets containing the same clock zones. As the clock zones are not shared in the state space representation, many clock zones will be represented several times, wasting memory.

Two ideas to exploit the characteristics of data-intensive real-time systems in order to design a more space efficient state space representation are combined in this thesis:

- Reversing the direction of the mapping between the locations and the timing information.
- Using a BDD library to represent the sets of locations space-efficiently.

The use of BDDs is helpful for systems with many locations, as large sets of locations can be represented concisely using BDDs. The use of some of-the-shelf BDD library for representing sets of locations is only efficient if clock zones are mapped to sets of locations, and not the other way round, because for efficient semi-symbolic exploration of the state space, it is necessary to have a quick implementation for looking up whether a particular zone (i.e., a pair of a location and a clock zone) is contained in a state space representation or not. A clock zone, represented as a canonical DBM, can be hashed using UPPAALs DBM library, thus looking up the corresponding BDD is possible in constant time, if the state space is organized as a hash map that maps DBMs to BDDs. If BDDs would be mapped to sets of DBMs, it would not be possible to find the BDD that contains the location in constant time, as hashing a location to find the right BDD would caricature the use of BDDs in the first place. These ideas are concretized in Chapter 4.

Chapter 4

Zone State Diagrams

Contents

In this Chapter, an alternative to the standard state space representation is presented. A TSS as presented in Section 2.6 is a mapping from locations to clock valuations. The approach presented in this Chapter reverses the direction of this mapping. By mapping DBMs to location information, a specialized data structure is created, providing a space-efficient storage for the state space of data-intensive real-time systems.

4.1 Definition

A *zone state diagram* (ZSD) maps a DBM to a set of locations: $ZSD : (\zeta \to 2^L)$. Let $\mathcal{D}$ denote the set of all ZSDs.

The empty ZSD, $\mathcal{d}_0$, maps every DBM to the empty set:

$$\forall z \in \zeta : \mathcal{d}_0(z) = \emptyset$$

A new ZSD can be formed by applying the mapping operation $[z \mapsto l]$ to a ZSD, which adds the location l to the set of locations to which the ZSD maps the DBM z. Consider a ZSD $\mathcal{d}' = \mathcal{d}[z \mapsto l]$, with $z, z' \in \zeta$, $l, l' \in L$, and $\mathcal{d} \in \mathcal{D}$:

$$\mathcal{d}'(z') := \begin{cases} \{l\} \cup \mathcal{d}(z') & \text{if } [\![z]\!] = [\![z']\!] \\ \mathcal{d}(z') & \text{otherwise} \end{cases}$$

Finally, the set of location-valuation pairs represented by a ZSD is defined as

$$[\![\mathcal{d}]\!] := \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge l \in \mathcal{d}(z)\}.$$

Note that this definition can also be written as:

$$[\![\mathcal{d}]\!] := \bigcup_{z \in \zeta} \mathcal{d}(z) \times [\![z]\!].$$

A ZSD is *equivalent* to a TSS iff they represent the same set of location-valuation pairs. Formally, let $\mathcal{d} \in \mathcal{D}, \theta \in \Theta$:

$$\mathcal{d} \equiv \theta \Leftrightarrow [\![\mathcal{d}]\!] = [\![\theta]\!]$$

4.2 Operations

In this Section, the binary set operators are defined for ZSDs. The union operator, needed for reachability analysis, is straightforward. However, intersection and subtraction are slightly more subtle.

A state of a timed automaton is a pair consisting of a location and a clock valuation. States are an obvious choice for a foundation for formal considerations on properties of ZSDs.

The $[\![x]\!]$-operator for $x \in \{\mathcal{D}, \Theta\}$ preserves the binary set operations union, intersection and subtraction, defined in the following. The proofs for preservation under $[\![x]\!]$ are provided along the way.

4.2.1 Union

Conveniently, the union of two ZSDs is straightforward: Let $z \in \zeta$, let $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$:

$$(\mathcal{d} \cup \mathcal{d}')(z) := \mathcal{d}(z) \cup \mathcal{d}'(z)$$

The following lemma states the preservation of the union operation under the $[\![x]\!]$-operator:

Lemma 4.2.1. *For $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$, the following holds:*

$$[\![\mathcal{d}]\!] \cup [\![\mathcal{d}']\!] = [\![\mathcal{d} \cup \mathcal{d}']\!]$$

Proof.

$$\begin{aligned}
[\![\mathcal{d}]\!] \cup [\![\mathcal{d}']\!] &= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge l \in \mathcal{d}(z)\} \\
&\quad \cup \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge l \in \mathcal{d}'(z)\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge \big(l \in \mathcal{d}(z) \vee l \in \mathcal{d}'(z)\big)\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge l \in \big(\mathcal{d}(z) \cup \mathcal{d}'(z)\big)\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta . \vec{v} \vDash z \wedge l \in \big(\mathcal{d} \cup \mathcal{d}'\big)(z)\} \\
&= [\![\mathcal{d} \cup \mathcal{d}']\!]
\end{aligned}$$

□

4.2.2 Intersection

The intersection of two ZSDs maps a DBM z to the union of all intersections of the two sets of locations mapped at by two DBMs respectively, the intersection of those DBMs being equivalent to z. Formally, let $z \in \zeta$, let $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$:

$$(\mathcal{d} \cap \mathcal{d}')(z) := \{l \in L \mid \exists z', z'' \in \zeta.l \in \big(\mathcal{d}(z') \cap \mathcal{d}'(z'')\big) \wedge [\![z]\!] = [\![z']\!] \cap [\![z'']\!]\}$$

The following lemma states the preservation of the intersection operation under the $[\![x]\!]$-operator:

Lemma 4.2.2. *For $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$, the following holds:*

$$[\![\mathcal{d}]\!] \cap [\![\mathcal{d}']\!] = [\![\mathcal{d} \cap \mathcal{d}']\!]$$

Proof.

$$\begin{aligned}
[\![\mathcal{d}]\!] \cap [\![\mathcal{d}']\!] &= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z' \in \zeta.\vec{v} \vDash z' \wedge l \in \mathcal{d}(z')\} \\
&\quad \cap \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z'' \in \zeta.\vec{v} \vDash z'' \wedge l \in \mathcal{d}'(z'')\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z', z'' \in \zeta.\vec{v} \vDash z' \\
&\qquad \wedge l \in \mathcal{d}(z') \wedge \vec{v} \vDash z'' \wedge l \in \mathcal{d}'(z'')\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z, z', z'' \in \zeta.\vec{v} \vDash z \wedge [\![z]\!] = [\![z']\!] \cap [\![z'']\!] \\
&\qquad \wedge l \in \mathcal{d}(z') \wedge l \in \mathcal{d}'(z'')\} \\
&= \big\{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z, z', z'' \in \zeta.\vec{v} \vDash z \wedge [\![z]\!] = [\![z']\!] \cap [\![z'']\!] \\
&\qquad \wedge l \in \{l' \in L \mid l' \in \mathcal{d}(z') \wedge l' \in \mathcal{d}'(z'')\}\big\} \\
&= \big\{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta.\vec{v} \vDash z \\
&\qquad \wedge l \in \{l' \in L \mid \exists z', z'' \in \zeta.l' \in \mathcal{d}(z') \wedge l' \in \mathcal{d}'(z'') \\
&\qquad\qquad \wedge [\![z]\!] = [\![z']\!] \cap [\![z'']\!]\}\big\} \\
&= \{(l, \vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta.\vec{v} \vDash z \wedge l \in \big(\mathcal{d} \cap \mathcal{d}'\big)(z)\} \\
&= [\![\mathcal{d} \cap \mathcal{d}']\!]
\end{aligned}$$

□

4.2.3 Subtraction

The ZSD resulting from subtracting a ZSD $\mathcal{d}'$ from a ZSD $\mathcal{d}$, maps a DBM z to all locations l that are mapped at by $\mathcal{d}$ from some DBM z' where all clock valuations contained in z are contained in z' but are not contained in any DBM z'' that is mapped to l by $\mathcal{d}'$. Formally, let $z \in \zeta$, let $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$:

$$(\mathcal{d} \setminus \mathcal{d}')(z) := \big\{l \in L \mid \exists z' \in \zeta.l \in \mathcal{d}(z') \wedge z \in \big(\{z'\} \setminus \{z'' \in \zeta \mid l \in \mathcal{d}'(z'')\}\big)\big\}$$

The following lemma states the preservation of the subtraction operation under the $[\![x]\!]$-operator:

Lemma 4.2.3. *For $\mathcal{d}, \mathcal{d}' \in \mathcal{D}$, the following holds:*

$$[\![\mathcal{d}]\!] \setminus [\![\mathcal{d}']\!] = [\![\mathcal{d} \setminus \mathcal{d}']\!]$$

Proof.

$$
\begin{aligned}
[\![\mathcal{d}]\!] \setminus [\![\mathcal{d}']\!] &= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z' \in \zeta.\vec{v} \vDash z' \wedge l \in \mathcal{d}(z')\} \\
&\quad \setminus \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z'' \in \zeta.\vec{v} \vDash z'' \wedge l \in \mathcal{d}'(z'')\} \\
&= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z' \in \zeta.\vec{v} \vDash z' \wedge l \in \mathcal{d}(z')\} \\
&\quad \cap \overline{\{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z'' \in \zeta.\vec{v} \vDash z'' \wedge l \in \mathcal{d}'(z'')\}} \\
&= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z' \in \zeta.\vec{v} \vDash z' \wedge l \in \mathcal{d}(z') \\
&\qquad \wedge \nexists z'' \in \zeta.\big(\vec{v} \vDash z'' \wedge l \in \mathcal{d}'(z'')\big)\} \\
&= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z' \in \zeta.\vec{v} \vDash z' \wedge l \in \mathcal{d}(z') \\
&\qquad \wedge \forall z'' \in \zeta.\big(\vec{v} \nvDash z'' \vee l \notin \mathcal{d}'(z'')\big)\} \\
&= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z, z' \in \zeta.\vec{v} \vDash z \wedge l \in \mathcal{d}(z') \\
&\qquad \wedge \forall \vec{v}' \vDash z.\Big(\vec{v}' \vDash z' \wedge \forall z'' \in \zeta.\big(l \in \mathcal{d}'(z'') \Rightarrow \vec{v}' \nvDash z''\big)\Big)\} \\
&= \Big\{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta.\vec{v} \vDash z \\
&\qquad \wedge l \in \big\{l' \in L \mid \exists z' \in \zeta.l' \in \mathcal{d}(z') \\
&\qquad\qquad \wedge z \in \big(\{z'\} \setminus \{z'' \in \zeta \mid l' \in \mathcal{d}'(z'')\}\big)\big\}\Big\} \\
&= \{(l,\vec{v}) \in L \times \mathcal{V}(X) \mid \exists z \in \zeta.\vec{v} \vDash z \wedge l \in \big(\mathcal{d} \setminus \mathcal{d}'\big)(z)\} \\
&= [\![\mathcal{d} \setminus \mathcal{d}']\!]
\end{aligned}
$$

□

4.3 Implementing Zone State Diagrams

In order to use ZSDs for timed model checking, the implementation of the ZSD concept was integrated into a prototype timed model checker that checks reachability of states in an *extended timed automaton*[1]. As only forward reachability model checking is used in this prototype, just the union operator `unify` is implemented.

The prototype model checker uses bit vectors to code a location of the automaton. A set of *locations* is encoded using a BDD that represents all bit vectors in the set. The BDDs are implemented using the CUDD library [Som09]. Note that the CUDD library will try to find a more efficient variable ordering from time to time, a procedure that dominates the runtime of the prototype in the long run, thus leading to a space efficient, but slow representation of the locations. The UPPAAL-DBM library [Ben02] is used for representing DBMs. A state of the automaton is encoded as a pair of a location vector and a clock federation.

ZSDs are basically a mapping from DBMs to sets of locations. Thus, a ZSD is implemented as a hash map that maps DBMs to location BDDs, using the hashing function provided by the UPPAAL-DBM library [Ben02]. A ZSD has a function `unify` that can be used to add states or sets of states to the ZSD.

First, consider the case of adding a state, i.e., a pair of a location and a federation, to a ZSD as shown in Algorithm 3. For every DBM in the federation, the location

[1] Behrmann et al. describe extended timed automata in [BDL04].

Algorithm 3 The function `unify(loc, F)` takes a location vector `loc` and a federation `F` and returns whether this pair was already contained in the ZSD, adding the state described by the pair to the ZSD. For DBM d, $ZSD[d]$ represents the BDD mapped to by d in ZSD.

```
changed := false
for all DBM d ∈ F do
  if d ∈ ZSD then
    if loc ∉ ZSD[d] then
      ZSD[d] := ZSD[d] ∨ loc
      changed := true
    end if
  else
    ZSD[d] := loc
    changed := true
  end if
end for
return changed
```

is added to the BDD the DBM maps to in the ZSD. The union of two ZSDs can be implemented similar, as shown in Algorithm 4: A mapping can always be described as a set of pairs, and ZSDs map DBMs to BDDs. Every pair of a DBM `d` and location set `LOC` from the one ZSD is added to the other ZSD.

Algorithm 4 ZSD Y has the function `union(Z)` taking a ZSD `Z` and transforming Y into the union of Y and `Z`. For DBM d, $Y[d]$ represents the BDD mapped to by d in ZSD Y.

```
for all (DBM, BDD) (d, LOC) ∈ Z do
  Y[d] := Y[d] ∨ LOC
end for
```

Forward reachability model checking tries to find a fixed-point of the set of reachable states using the *post* operator defined in Section 2.5 to find all immediate successors of reachable states of the automaton. As shown in Algorithm 2 from Section 2.5, it is not necessary to apply *post* to all reachable states in every iteration, but only to the states that are newly found to be reachable. To exploit this source of efficiency, the set of newly found states is over-approximated by adopting Algorithm 3 to collect all changes it makes to a ZSD.

Finally, the function `unifyMany` adds several states to a ZSD and returns a set of the states that were newly added and have not been in the ZSD before, as shown in Algorithm 5: For all states, if the location of the state is not mapped to by all DBMs of the state's federation, collect the appropriate DBMs in the federation `new` and add the pair consisting of the location and `new` to the set of newly found states.

A specific location is contained in a ZSD, if it is in one of the BDDs mapped at by one

Algorithm 5 The function `unifyMany(results, delta)` takes a set of states `results`, which was produced by the application of *post* to the ZSD, and a set of states `delta`. It adds all states from `results` not contained in the ZSD to the ZSD and to `delta`. Finally, `unifyMany` returns whether the ZSD has been changed. For DBM d, $ZSD[d]$ represents the BDD mapped to by d in the ZSD.

```
added := false
for all (location, federation) (loc, F) ∈ results do
  new := empty federation
  for all DBM d ∈ F do
    if d ∈ ZSD then
      if loc ∉ ZSD[d] then
        ZSD[d] := ZSD[d] ∨ loc
        new := new ∪ {d}
        added := true
      end if
    else
      ZSD[d] := loc
      new := new ∪ {d}
      added := true
    end if
  end for
  delta := delta ∪ {(loc, new)}
end for
return added
```

of the DBMs in the ZSD. So, checking reachability is relatively straightforward. Note that in the implementation of the algorithms, all empty states containing no valuations or no locations are of course immediately removed and no longer considered.

Part II

FlexRay

Chapter 5

Protocol Overview

Contents

The growing use of embedded devices in cars has given rise to the development of communication protocols specifically tailored to an automotive context. The communication has to be fast and reliable. Slow communication or uncorrected errors can have catastrophic consequences in safety-critical applications such as *drive by wire*.

FlexRay is a communication protocol developed by the FlexRay consortium which consists of major industrial companies such as BMW, Bosch, Daimler, Freescale, General Motors, NXP Semiconductors, and Volkswagen. FlexRay was first used in BMW's X5 car for the pneumatic damping system. The fifth generation of BMW's 7 Series uses FlexRay for safety critical systems such as brake control. The development of FlexRay finished in 2009, the newest available specification version is 2.1 revision A [Fle05].

Verification of FlexRay is an active field of research. Beyer et al. [BBG+05] presented a manual deductive correctness proof for FlexRay's physical layer protocol under the assumption of a fault free underlying hardware layer. Schmalz [Sch06, Sch07] gave a semi automatic correctness proof, using Isabelle/HOL and the NuSMV model checker for the proof obligations. Knapp and Paul [KP07] integrated this correctness proof into a deductive correctness proof for an implementation of a programming model for

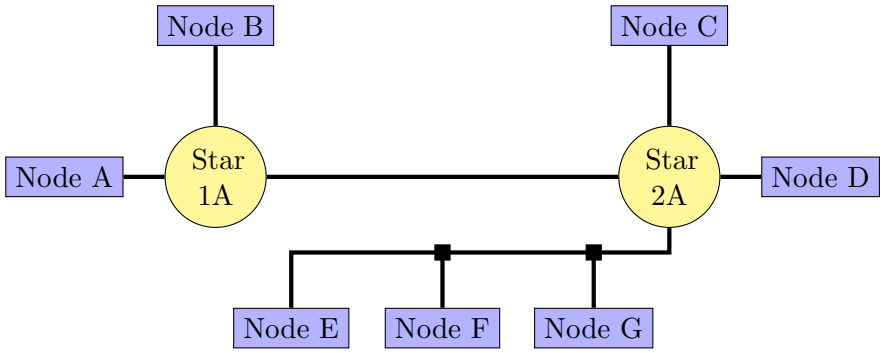

Figure 5.1: FlexRay single channel hybrid network

a distributed time-triggered real-time system at instruction set architecture level, using worst-case execution time analysis, while Alkassar et al. [ABK08b, ABK08a] extended the correctness proof for FlexRay's physical layer protocol to formalize the correctness of a real-time scheduler for an automotive bus inspired by FlexRay. All these approaches assume a fault free underlying hardware layer.

5.1 Architecture

The FlexRay protocol is used to organize communication in a network. The network, which is also called *cluster*, is formed of so called *nodes*, each of which has a FlexRay *controller*, which is described in Section 5.1.2.

Each FlexRay controller can be attached to embedded devices or computers. The cluster can be configured in a bus or star network architecture or a hybrid combination, as shown in Figure 5.1, and can support a maximum of 64 controllers. Each controller can support up to two independent communication `channels`.

5.1.1 Time Organization

A FlexRay cluster organizes the access to the channel separately for each channel by trying to maintain a channel-wide schedule, thus a *time division multiple access* (TDMA) scheme is used. The correctness of critical parts of the scheme is demonstrated by Alkassar et al. in [ABK08b, ABK08a]. The timing hierarchy is shown in Figure 5.2 and described in the following.

In one cycle of communication, one *static slot* is reserved for usage by each node, thus, every node connected to the channel gets the chance to use the bus. This *static segment* of the cycle is followed by a *dynamic segment* where every node is allowed to try to send a message outside the normal schedule. Thus, time critical communication is faster. The end of the cycle consists of a small *symbol window* for protocol related communication using so called *symbols* and finally a *network idle time* during which the local view of the global time is adjusted to achieve some degree of synchronization with the rest of the cluster.

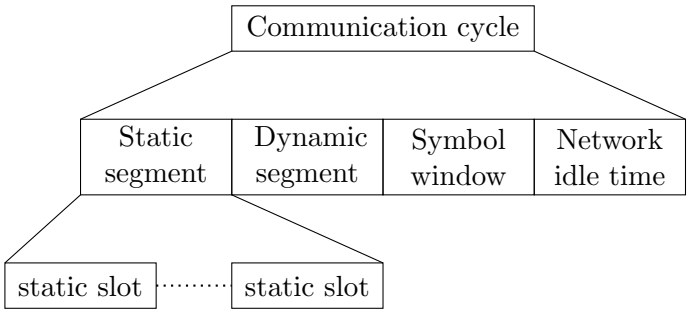

Figure 5.2: FlexRay communication cycle schedule

5.1.2 Controller Architecture

Each node in a FlexRay network has a FlexRay controller, which is connected to the electronic device that is to be connected using the FlexRay protocol and to the FlexRay network, which will be called *bus* in the following, regardless of its architecture. The controller provides a standardized interface to the electronic device and controls the communication over the FlexRay network.

As shown in Figure 5.3, a FlexRay controller consists of a distribution-and-control layer, one communication layer per channel and one bit-level layer per channel. The controller consists of six processes each of which can be attributed to one of the layers. The processes belonging to the communication or bit-level layer are instantiated once for each channel.

Distribution and control

The FlexRay controller communicates with the host using the *controller host interface*, while the *protocol operation control* controls the protocol's operation. The *clock synchronization processing* and the *macrotick generation* try to establish a shared view of time throughout the whole cluster.

Controller Host Interface. The *controller host interface* (CHI) provides a well defined interface for all communication between the host and the FlexRay controller. The host can use the interface to control the configuration of the controller or check the controller's status. The CHI distributes the commands of the host to the relevant processes of the controller.

The CHI buffers all incoming and outgoing communication via the FlexRay bus.

Protocol Operation Control. The *protocol operation control* controls the overall behavior of the protocol by controlling the status of the other processes of the controller.

Clock Synchronization Processing. The *clock synchronization processing* (CSP) tries to establish a shared view of the time for the whole cluster by keeping the local

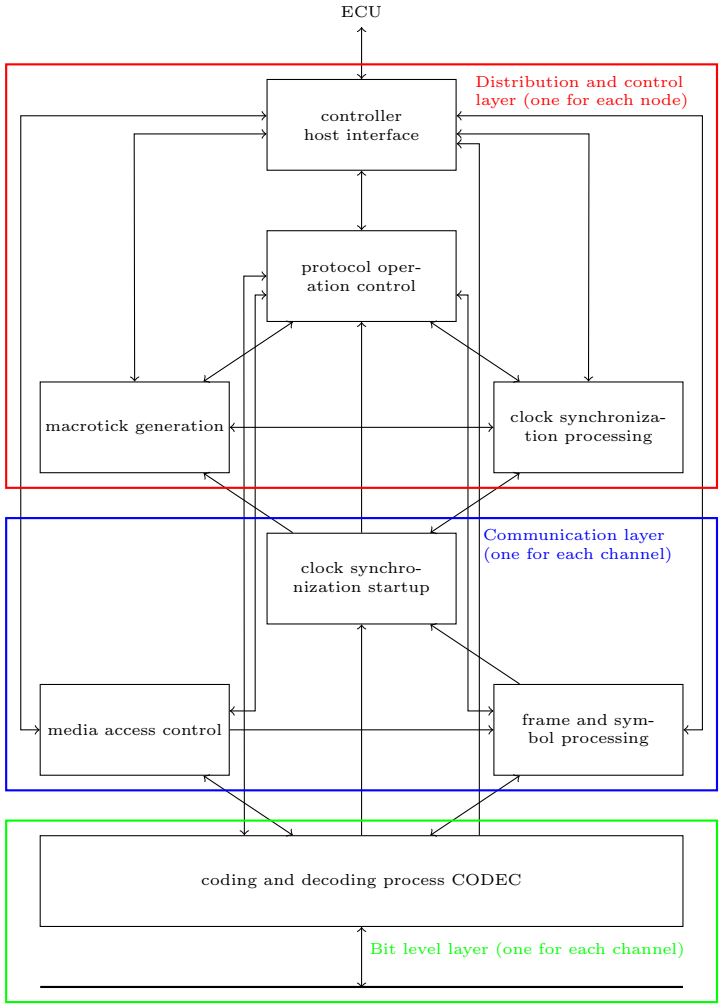

Figure 5.3: FlexRay node architecture

view of the time close to the views of other nodes. To achieve this, it analyses the synchronization information provided by the *clock synchronization startup* and the *frame and symbol processing* and calculates values for the macrotick generation that are then used to reduce the difference between the local view of time and the view of time of the synchronization nodes in the rest of the cluster.

Macrotick Generation. A *macrotick* is a time unit used to measure the lengths of slots, segments and communication cycles that are described in Section 5.1.1.

The *macrotick generation* adjusts the length of the macroticks and adjusts the length of the communication cycle during the network idle time, according to the values calculated by the CSP.

Communication

The communication layers of each channel operate independent of each other. The *clock synchronization startup* triggers the *clock synchronization*. The *media access control* guards the access to the communication channels. The *frame and symbol processing* handles the meta data attached to received messages.

Clock Synchronization Startup. The *clock synchronization startup* process waits for startup signals on the bus and records their arrival time. This data is sent to the CSP, which synchronizes the local view of the time with the rest of the cluster as described in Section 5.1.2. A detailed implementation and mathematical description of this process can be found in [Böh06].

Media Access Control. The *media access control* (MAC) controls the access to the bus by enforcing the compliance with the schedule based on the local view of the global time as described in Section 5.1.1. The MAC decides when to send a symbol or when to send a message, which will be called message *frame* in the following. The MAC furthermore attaches meta data to the outgoing communication by assembling the *frame header* which is described in [Fle05, Chapter 4]. Figure 5.4 shows the format of a frame.

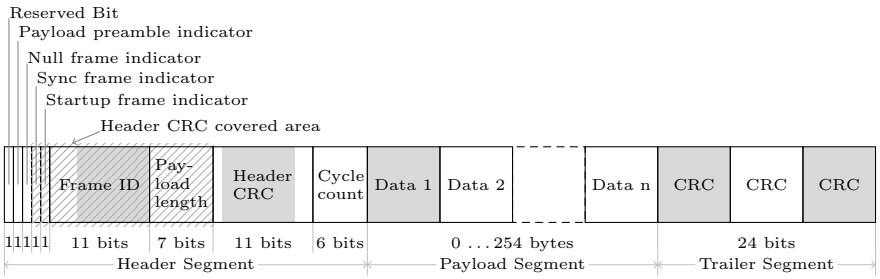

Figure 5.4: Format of a frame

Frame and Symbol Processing. The *frame and symbol processing* (FSP) controls the integrity of the communication. It analyzes the meta data of incoming communication to check if the message has the right format and is consistent with the local assumptions about the time (see 5.1.1). The FSP then either reports an error or, if there is none, removes the meta data before passing the message on. Furthermore, the FSP forwards meta data that can be used for synchronization to the CSP, which is described in Section 5.1.2.

Bit Level

The bit level layer consists of the *coding and decoding* process (CODEC), which handles all communication via the bus, i.e., symbols or messages. In the following, the focus

is on messages. CODEC sends messages via the bus using FlexRay's *physical layer protocol* adding a *cyclic redundancy code* (CRC) that enables the receiver to recognize errors. CODEC also receives messages from the bus using the physical layer protocol, and checks for consistency with the attached CRC and the header CRC from Figure 5.4.

A detailed implementation and mathematical description of a CODEC can be found in [Ger07].

The correct operation of the physical layer protocol is at the heart of the problem of the correctness of FlexRay. Thus, proving the correctness of the physical layer protocol is the prerequisite for arguing about FlexRay's overall correctness.

5.2 Coding and Decoding

While the higher levels of the FlexRay protocol manage the communication process, i.e., decide when to send and what to send, and when to listen for messages, CODEC is responsible for actually transmitting information from one node to the other nodes of the cluster. The rest of the protocol operates under the assumption that CODEC works as intended. So, proving the correctness of the physical layer protocol used by CODEC is vital.

FlexRay uses two types of communication, message frames, and symbols.

5.2.1 Frames

Messages are enriched with meta data before they are given to the encoding process (ENC). A frame consists of a 5 byte *header*, containing the meta data, and a payload section, which contains up to 254 message bytes, and finally a 3 byte CRC covering the header and the payload section.

5.2.2 Symbols

Symbols are used for the wake up procedure and for testing the availability of the bus, they do not transport more information than the fact that the symbol itself is sent. They have a predefined format that is relatively simple and reliable compared to frames. Thus, if the CODEC can handle frames correctly, it is easy to check if it can also handle symbols correctly.

5.3 Physical Layer Protocol

A *physical layer protocol* is a protocol used to transmit data using some physical layer, the receiver and the sender being only connected through this physical layer, e.g., a wire. In the following, the setting in which the physical layer protocol will be operated is presented. Afterwards, the format used to structure the transmitted bit stream is presented and finally the mechanism adding redundancy before transmission and removing this redundancy after reception of a message is discussed.

5.3.1 Setting

The simple task of sending messages using a bus becomes difficult in a decentralized setting. For embedded devices in cars, a decentralized communication architecture as provided by FlexRay is needed for reasons of reliability and efficiency. Several problems have to be tackled in such a setting.

Asynchronous Clocks

Every node in a FlexRay cluster has its own oscillator. The frequency of these oscillators may deviate as much as 0.15% from the standard. Furthermore, as these oscillators are not started at the same time, their frequencies can be arbitrarily drifted against each other right from the start. So, the clock edges derived from these oscillators will almost never occur at the same time in all nodes of the cluster.

Even if the delay that the signals acquire while traversing wires is ignored, a change of the value on the bus due to actions triggered by some rising clock edge of the sender will arrive at the position on the bus where the receiver is listening at a point in time that is not connected to the receivers rising clock edges. While the functionality of most of the FlexRay protocol can be described and analyzed using two valued logic with a simple cycle-based model, the transfer of messages over the bus can only be described and analyzed when the continuous nature of time is taken into account. Thus, the standard binary model with cycle-based discrete time is not sufficient.

Register Semantics

Cycle-Based Digital Model. In the cycle-based digital model, in every clock cycle the register will contain some well defined value. In the next cycle, it will still contain this value if the *enable input* in the cycle before was 0. If the enable input in the cycle before was 1, in the next cycle r will contain the value of the *input* of register r in the cycle before.

Formally, let r^i_{en} be the value of the enable input of register r in cycle i, let r^i be the value of the register r in cycle i and let r^i_{in} be the value of the input of register r in cycle i:

$$r^{i+1} = \begin{cases} r^i_{in} & \text{iff } (r^i_{en} = 1) \\ r^i & \text{else} \end{cases}$$

However, for describing a setting with more than one clock, a more elaborated model taking into account the continuous nature of time and more values than logical 1 and logical 0 is needed. In the following, a real-time model combined with a three valued logic (*high*, *low* or *undefined*) is used to describe the behavior of a register with an input connected to a circuit with a different clock.

Continuous-Time-Based Model. Figure 5.5 shows the setting of a sending and a receiving register connected via a bus. The hardware on the sender's side can be described using the cycle-based time abstraction from Section 5.3.1, as can be the

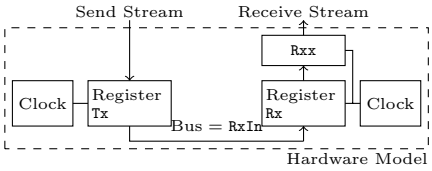

Figure 5.5: Overview of the hardware sub-architecture.

hardware on the receiver's side, as both receiver and sender each have a clock and the cycles of that clock can be used for abstraction. The setting depicted in Figure 5.5 cannot be fully described using the cycle-based time abstraction from standard hardware models, every signal on the bus is relative to a certain point in time. As the communication is strictly going from the sender to the receiver, this affects only the description of the receiver's `Rx` register, as it's input is connected to the output of the sender's `Tx` register using the bus, and the sender uses a different clock, making cycle based abstraction impossible.

To describe registers like `Rx`, let $r(t)$ describe the value of register r at time t.
Several parameters have to be considered:

- τ is the cycle time of the connected clock, i.e., the time between two consecutive rising edges of the clock
- t_s is the *setup time*, i.e., the time that the value on the input of an enabled register is required to be stable *before* the occasion of a rising clock edge
- t_h is the *hold time*, i.e., the time that the value on the input of an enabled register is required to be stable *after* the occasion of a rising clock edge
- t_{pmax} is the maximal propagation delay, i.e., the maximal time that an enabled register needs to change its value to the new value after the occasion of a rising clock edge and while the stable value on the input was different from the value of the register
- t_{pmin} is the minimal propagation delay, i.e., the minimal time that an enabled register needs to change its value after the occasion of a rising clock edge and while the stable value on the input was different from the value of the register

It is evident that $t_{pmin} \leq t_{pmax}$. Note that the register represents the logical values using voltage levels. A value below a certain voltage level is considered as logical 0 and a voltage above a certain level is considered as logical 1. Between those levels, there is a certain range of voltage levels that cannot be interpreted as a logical value. Furthermore, changes in a voltage level take time, which will be bounded by a worst case for simplicity. However, generally it takes, e.g., a different amount of time to pull the voltage on a wire up (from logical 0 to logical 1) than it takes to pull the voltage down (from logical 1 to logical 0).

As long as the register is not enabled, i.e., $r_{en} = 0$, the value of the register stays the same, just as in the cycle-based digital model. As soon as the register is enabled, i.e., $r_{en} = 1$, the semantics vary.

The behavior of the register r is as follows: If it is enabled, assuming a rising clock edge at time T, a stable value on the input of the register in the interval $[T - t_s, T + t_h]$ and value on the input that is different to the value of the register, $r_{in}(T) \neq r(T)$, the value of the register is guaranteed to stay the same until $T + t_{pmin}$ and is guaranteed to be the new value after $T + t_{pmax}$. During the interval $(T + t_{pmin}, T + t_{pmax})$, the value of the register is unknown.

Figure 5.6 shows three values:

- the value of the clock[1] which is connected to register R,
- the bus which is connected to the input R_{in} of the enabled register R,
- the output R_{out} of the enabled register R.

Ω designates an unknown value that is either representing logical 0 or 1 or some voltage level in between.

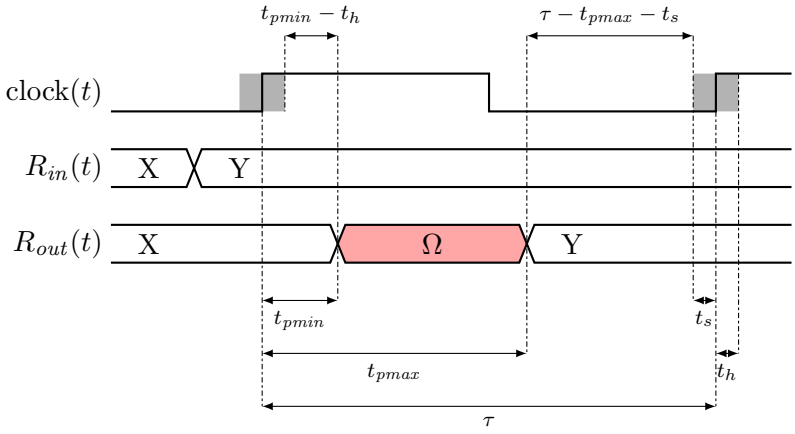

Figure 5.6: Value of enabled register R over time t

Assuming a rising clock edge at time T and a change on the input of the register in the interval $[T - t_s, T + t_h]$ the value of the register is guaranteed to stay the same until $T + t_{pmin}$, but afterwards, its value is unknown. Nevertheless, it is assumed[2] that the unknown value is stable before $T + \tau - t_s$, i.e., before it could violate the setup times of connected registers in the next cycle.

Note that assuming sensible values for the parameters, i.e., changing a register's value does not violate the setup time or hold time of connected registers ($\tau - t_s > t_{pmax}$ and $t_{pmin} > t_h$), most of the functionality of a circuit can be described using the cycle-based digital model from Section 5.3.1.

[1] A rising edge of the clock constitutes a tick of the clock.

[2] See [KP95, Section 5.2] for more details on the issue of metastable registers.

As long as the register is not enabled, the value of the register does not change. As long as the value of the stable input is the same as the value of the register, the register also does not change. The semantics of an enabled register can be formally defined: Assume a rising clock edge at time T, let $t_{old} = T-\tau+t_{pmax}$, let $t_{next} = T+\tau+t_{pmin}$, let $t_{old} \leq t \leq t_{next}$ and assume some change of the value $\exists t'.T-t_s \leq t' \leq T+t_h \wedge (r_{in}(t') \neq r(t_{old}))$.

$$r(t) = \begin{cases} r(t_{old}) & t_{old} \leq t \leq T + t_{pmin} \\ \Omega & T + t_{pmin} < t < T + t_{pmax} \\ X & T + t_{pmax} \leq t \leq t_{next} \end{cases}$$

where

$$X = \begin{cases} r_{in}(T) & \forall t'.(T - t_s \leq t' \leq T + t_h) \Rightarrow (r_{in}(t') = r_{in}(T)) \\ \Omega & \text{else} \end{cases}$$

Note that, generally, Ω will be either logical 1 or logical 0. There is, however, a very small probability that a register with a value of Ω is in a so called metastable state[Män98], i.e., the voltage level oscillates for a while before resolving to either logical 1 or logical 0.

If an enabled register is given a metastable input, there is again a very high probability that the register will not store this metastable value[3] but that it will store either a logical 1 or a logical 0. So, if a register at risk of becoming metastable is only connected to a second register, and only the second register is used by other circuits, the probability of having a metastable second register is negligible.

See [BBG+05] and [KP95, Section 5.2] for further details on the subject of continuous time models for registers. The undesired behavior introduced by sampling Ω due to the drifting clocks is called *jitter*.

Glitches

Typically in a modern car, the wires of the bus are several meters long. These wires can sometimes have characteristics similar to antennas. Thus, electromagnetic interference has to be taken into account. Very strong interference can always disturb electronic communication, but smaller disturbances should be tolerable. The voltage level on the bus might be affected by electromagnetic interference, in consequence, a logical value sent via the bus might be replaced by an arbitrary value from Ω, i.e., it might still be correctly read, it might be negated, or the value might be neither logical 1 nor logical 0. Taking into account the register semantics as described in Section 5.3.1, even a small fluctuation in the voltage level might violate the stability requirement for the input of the register connected to the bus. Simply said, it is possible that something different from the bit that has been sent is received. Such a falsification of a bit is called a *glitch* in [Fle05, Section 3.2.2].

[3] A construction with two consecutive registers is a standard method of forcing a metastable value to either logical 1 or logical 0, e.g., it is used in [Jon06, Figure 15]. If an additional small delay is added, metastability can be excluded according to [Män98].

If too many glitches occur, the message can be compromised. This can be recognized using the CRC code of the message, as described in Section 5.1.2. If a glitch flips a bit and another glitch flips one of the next 4 bits, i.e., if 2 bits in any sequence of 5 consecutive bits can be flipped by a glitch, experiments with UPPAAL have shown that the message can be compromised[GEFP10]. However, infrequent glitches can be compensated for by the physical layer protocol. For the purpose of verification of the FlexRay protocol, it will be assumed that the next 4 bits after a bit affected by a glitch will not be affected by a glitch.

5.3.2 Frames

A message frame as described in Section 5.2.1 can be seen as a bit string. This string is transmitted as a stream of bits, but the stream is structured as described in [Fle05, Section 3.2.1.1]. Figure 5.7 shows the format of a message frame bit stream. This structure is used for the bit clock alignment as described in Section 5.3.3.

The start of the stream is the so called *transmission start sequence* (TrSS[4]), 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 [Fle05, Section B.2.1]. As the bus is high idle, this long low period can be recognized no matter how badly synchronized the nodes are on the bit level. It precedes every transmission.

After the TrSS, the *frame start sequence* (FSS) signals the start of a frame transmission, as opposed to a symbol. The FSS consists of a single high bit. A receiving node will accept a transmission even if the FSS is not received, 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.

The bit string of the message frame is partitioned into bytes. Each of these bytes is prefixed with a *byte start sequence* (BSS) and then sent. 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 of the bit clock alignment.

At the end of the message frame, a *frame end sequence* (FES) is sent. The FES consists of one low bit followed by a high bit.

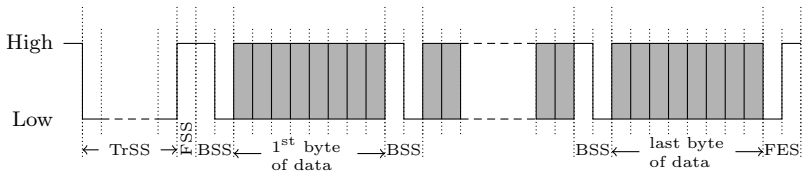

Figure 5.7: Format of a message frame bit stream

[4]In [Fle05], *transmission start sequence* is abbreviated TSS, but this shorthand is already used for TSSs from Section 2.6.

5.3.3 Redundancy

To achieve resilience against errors, FlexRay uses redundancy. Every bit that is to be sent is sent 8 times consecutively, i.e., the length of a *bit cell* is 8 clock cycles. This allows to correct some errors, but reduces the throughput of the protocol. This can be implemented very easily on the sender's side, as a simple cyclical counter counting from 1 to 8 will suffice to control the process.

Voting

The receiver has to reconstruct the information from the samples it receives in the bit cell and eliminate errors if possible. To achieve this, the samples from the bus are stored in a so called *voting window* which contains the sample from this cycle[5] and the samples from the 4 cycles before. The value to be considered by the higher layers of the protocol is the *voted value*, the value of the majority of the 5 samples in the voting window. As described in Section 5.3.1, the values sampled from the bus using two consecutive registers can be assumed to have a logical value of either 1 or 0. Thus, there will always be a majority.

This voting process will add an additional constant delay of 2 cycles until a change in the values sampled from the bus will be noticed by higher levels of the protocol. This delay is called *voting delay*. The number of bits is not affected by this process, from every bit cell several voted values are taken as there are several samples from each bit cell.

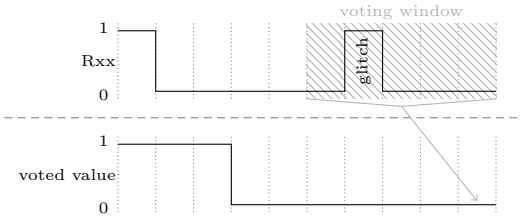

Figure 5.8: Correction of a glitch through majority voting

Glitches as described in Section 5.3.1 can be corrected as shown in Figure 5.8 by this voting procedure, as a glitch in a voting window with five bits that does not change the majority does not affect higher layers of the protocol. However, if the glitch occurs close to a change in the value to be sent, it might add a delay to the change. If the glitch inverts one of the bits of the new value, it will take one more cycle until the new value is the majority in the voting window. On the other hand, if the glitch inverts one bit of the old value, the value will change one cycle to early. This might lead to a situation where the last and the first sample from a bit cell, which should contain the same logical value in every bit, are inverted. This kind of error may also be the result

[5] According to [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], [Böh06] and [Ger07].

of drift between the receiver's and the sender's clock that can lead to violated setup or hold times, as described in Section 5.3.1. Of course, these errors can also combine, as shown in Figure 5.9.

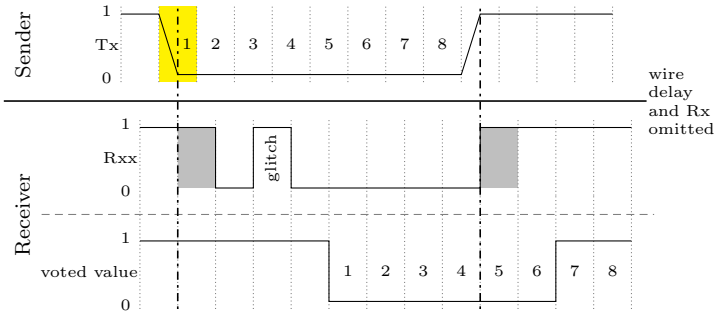

Figure 5.9: Combination of violated setup or hold time with a glitch

Strobing

Out of the voted values from a bit cell, just one is used for the higher levels of the protocol. To avoid choosing voted values that are affected by glitches, a voted value out of the middle of the bit cell, the fifth voted value, is chosen. The chosen voted value is called *strobed value*. To decide which voted value is the fifth out of the voted values from the bit cell, the *strobecounter* is used. The cyclical strobecounter counts from 1 to 8 and can be reset to 2 whenever needed.

Bit Clock Alignment

As the receiver's and the sender's clock drift against each other, it is necessary to repeatedly synchronize the strobecounter to the stream of received voted values, in order to identify the boundaries of the sequence of voted values from one bit cell and thus the voted value which corresponds to the fifth voted value of the bit cell.

The bit clock alignment mechanism makes use of the message frame format as described in Section 5.3.2. 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 the strobecounter is reset to 2 for the next voted value. Thus, the second recognized voted value of the voted values sequence from the low bit cell is considered the second voted value from 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, the strobecounter will be off by more than 1. Thus, parts of the 6th and 7th bit of the bit cell might be strobed instead of the fifth. This situation is shown in Figure 5.9; remember the voting delay of 2 cycles as described in Section 5.3.3. The bit clock alignment can analogously also happen to early.

Chapter 6

The FlexRay Benchmark

Contents

Only a part of a FlexRay controller is modeled, namely the *physical layer protocol* that is an important part of FlexRay's CODEC process. A physical layer protocol is used to transmit data over some physical device, like a wire, from a sender to a receiver, both running asynchronously to each other.

Physical layer protocols are investigated by several groups, e.g., Vaandrager and Groot [Vd06] use simulation with Uppaal to derive invariants of the *biphase mark* protocol. These invariants are then proven using the proof assistant PVS. Brown and Pike [BP06] use the verification tool SAL to automate parts of the correctness proofs for biphase mark and also the *8N1* physical layer protocol. However, unlike FlexRay, 8N1 and biphase mark are not designed for an unreliable physical communication link.

The model can be divided into two parts: the hardware model and the protocol model, as shown in Figure 6.1.

The model considers the transfer of a message frame in the static segment. As the transfer of symbols is less complex, the interesting scenario is the transfer of a message frame.

[BBG+05, Sch06, Sch07, ABK08b, ABK08a] have already shown that the transfer of a message frame will succeed under ideal circumstances. The focus of the model is the area that is affected by the new parts of the error model that are described in Section 5.3.1, i.e., glitches that flip bits on the bus. Just the transfer of a bit

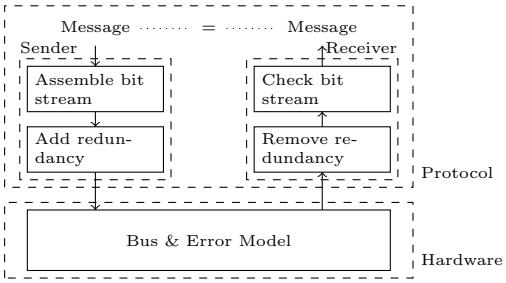

Figure 6.1: The structure of the model.

stream using the physical layer protocol is modeled. The correctness of the higher levels and the ability of FlexRay to deal with errors outside the error model described in Sections 5.3.1 and 5.3.1 naturally cannot be verified using this model.

6.1 Scenario Description

Assume a node is trying to send a message frame during it's static slot in the static segment, and some other node is trying to receive this frame. The number of bytes in the frame can be configured, a frame format as described in Figure 5.4 is not assumed. Nevertheless, the bit stream of the frame is structured by the physical layer protocol as described in Figure 5.7.

The operation of the protocol inside the receiver and the sender is modeled assuming a cycle-based, two-valued-logic-based behavior, as described in Section 5.3.1. However, to model the bus and the single register of the receiver that is connected to the bus, a behavior as described in the continuous-time-based model from Section 5.3.1 is assumed, which will introduce jitter.

Furthermore, the possibility of a bit being flipped by electromagnetic interference, i.e., a glitch as described in Section 5.3.1, is modeled but may only affect a bit of the stream if none of the 4 bits received before this bit have been affected.

The sender transmits the formated bit stream, and the receiver checks if the format of the stream complies with the standard described in [Fle05, Section 3.2.1.1]. The receiver also nondeterministically checks if a bit of the frame is received as it was sent. Model checking will guarantee that every bit is considered, as every bit could potentially be checked.

6.2 Configuration Parameters

The model can be configured by setting the following constants[1]:

- `messagelength`(8) describes the length of the frame in bits. As the length of a frame is defined in bytes, messagelength has to be a nonzero multiple of 8.

[1]The default values are given in brackets

- `TSSlimit`(15) is the number of low bits that will be sent in a TrSS (see Section 5.3.2).
- `TSSmin`(3) is the number of low bits that have to be received to recognize a TrSS.
- `delayHLmin`(1001) is the minimal propagation delay[2] on the bus if the value changes from high to low (see Section 5.3.1).
- `delayLHmin`(1001) is the minimal propagation delay on the bus if the value changes from low to high.
- `delayHLmax`(5001) is the maximal propagation delay[3] on the bus if the value changes from high to low.
- `delayLHmax`(5001) is the maximal propagation delay on the bus if the value changes from low to high.
- `setup`(1000) is the setup time of a register (see Section 5.3.1).
- `hold`(1000) is the hold time of a register (see Section 5.3.1).

The model uses the following shared variables[4]:

- `Tx`:=1(int[0,1]) is the value that the sender puts on the bus.
- `Rx`:=1(int[0,1]) is the value that the receiver samples from the bus. As two consecutive registers are used to sample from the bus, metastability is ignored here (see `Rxx`).
- `Rxx`:=1(int[0,1]) is the register that forwards `Rx` to the rest of the sender. At least the second register will not be metastable but have a logical 0 or logical 1 value as described at the end of Section 5.3.1.
- `VV`:=1(int[0,1]) is the voted value in the current cycle of the receiver's clock (see Section 5.3.3).
- `OldVV`:=1(int[0,1]) is the voted value of the cycle before.
- `EnableSyncEdgeDetect`:=1(int[0,1]) enables or disables the bit clock alignment (see Section 5.3.3).

[2]The minimal propagation delay may not be bigger than the corresponding maximal propagation delay. Furthermore, it should be bigger than `hold`.

[3]The maximal propagation delay may not be smaller than the corresponding minimal propagation delay. Furthermore, it should be smaller than 10000 − `setup`, assuming an ideal cycle length of 10000.

[4]The type of the variable is given in brackets, the range of the variables can be restricted to an interval, which is then given directly after the type. The variables are initialized with the value given behind the ":=" sign. As the bus is high idle, variables connected to the bus or derived from those are initialized assuming a logical 1 value for the idle bus.

- `bstr`:=1(int[0,1]) is the strobed value used by the higher protocol layers(see Section 5.3.3), i.e, the fifth voted value from each bit cell.

- `savedTX`:=2(int[0,2]) saves a message bit to allow the receiver to check if it correctly received this bit.[5]

- `savedindex`:=`messagelength`(int[0,`messagelength`]) saves the index of a message bit that is to be checked by the receiver (see `savedTX`).[6]

Channels

The automata of the model synchronize using broadcast channels, i.e., all automata that try to receive the broadcast will synchronize when the sending automaton sends the signal, but the sending automaton does not wait for other automata, it just sends the broadcast and proceeds no matter how many automata, if any, synchronize. To adjust for this, the automata were designed in such a way that they do not miss their relevant synchronization points.

The following channels are used:

- `SenderCLK` is the clock tick, i.e., the rising flank of the clock signal, of the senders oscillator. It corresponds to one *microtick* in [Fle05].

- `ReceiverCLK` is the clock tick, i.e., the rising flank of the clock signal, of the receivers oscillator. It corresponds to one *microtick* in [Fle05].

- `ValueVoted` signals that the voting mechanism has successfully updated the voted value, `VV`.

- `Strobed` signals that the bit clock alignment has strobed a new strobed value, `bstr`.

6.3 Hardware Model

The hardware model represents the clocks of the sender and the receiver in Section 6.3.1, the output of the `Tx` register used by the sender to put a value on the bus, the model of the bus with the nondeterministic glitches, the receivers `Rx` register used to sample a value from the bus and finally the consecutive `Rxx` register used to resolve an unstable sample to either high or low in Section 6.3.2. Figure 5.5 shows the setting of the hardware model.

[5] `savedTX` should be initialized with 2 to indicate that no bit has been saved yet.

[6] `savedindex` should be initialized with the value of `messagelength` to indicate that no bit has been saved yet.

6.3.1 Clocks

Each FlexRay controller has an oscillator which is used to generate the clock signal. These oscillators may deviate from the standard frequency by at most 0.15%, according to [Fle05, Appendix A.1]. The model assumes that the clock cycle of an ideal clock has a length of 10000. The deviation is modeled by allowing the length of a cycle to vary between 9985 and 10015.

Sender Clock

The model models the system starting exactly from the point in time of a clock tick from the sender, but after that clock tick has been broadcasted. The length of a cycle may only vary due to the 0.15% deviation. Figure 6.2 shows the model of the sender's clock.

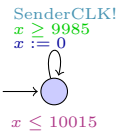

Figure 6.2: Model of the sender's clock

Receiver Clock

The clock of the receiver may be arbitrarily drifted against the clock of the sender at the beginning. As the length of a clock cycle may be at most 10015, this is the maximal length of the the first cycle. After that first clock tick, the length of the cycle may only vary due to the 0.15% deviation. Figure 6.3 shows the model of the receiver's clock.

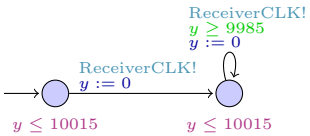

Figure 6.3: Model of the receiver's clock

6.3.2 Bus and Register Semantics

The model of the bus and the registers that sample values from the bus has to represent the behavior described in Section 5.3.1.

Bus

The bus is represented by the variable `RxIn`, which represents the value of the bus at the input of the receiver's `Rx` register. As the bus is high idle, it initially has a stable high value. At every tick of the sender's clock, the variable `Tx` is checked: if the sender is still putting the same value on the bus, nothing changes, but if the sender tries to put a different value on the bus, `RxIn` will change its value. This change will be delayed according to the propagation delay as described by t_{pmax} and t_{pmin} from Section 5.3.1. The delay is modeled using the constants `delayHLmin`, `delayHLmax`, `delayLHmin` and `delayLHmax` from Section 6.2.

A register as described in Section 6.3.2 needs a stable input for `setup` time units before a tick of the receivers clock occurs, otherwise an uncertain value is sampled. The uncertain value of the bus between t_{pmin} and t_{pmax} is modeled using a value of 2 for `RxIn`.

More formally, register `Rx` needs a stable input for `setup` time units before a tick-event of the receiver's clock occurs at time T_r. With a tick of the sender's clock at time T_s, the sender decides to put the new value on the bus. This new value is stable after $T_s + t_{pmax}$. Thus, we want:

$$T_s + t_{pmax} < T_r - \texttt{setup} \Leftrightarrow T_s + t_{pmax} + \texttt{setup} < T_r$$

To model this requirement on the sender's side, the uncertainty period is prolonged by `setup` in the model[7] of the bus as shown in Figure 6.4.

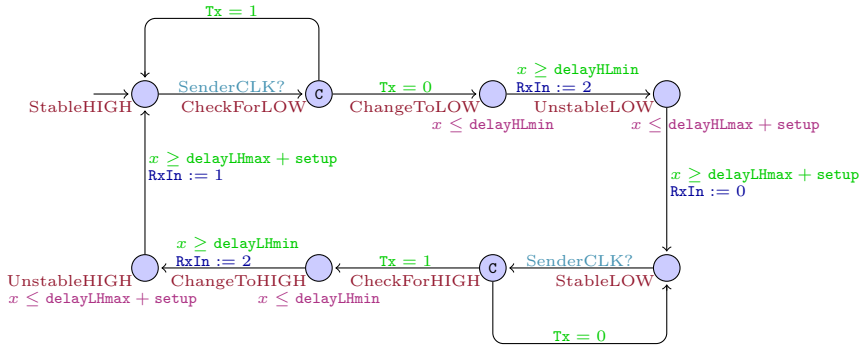

Figure 6.4: Model of the bus

Receiver Bussampler

The problem of metastability, as described in Section 5.3.1, is addressed by adding a second consecutive register, `Rxx`, and is not modeled.

[7] We assume a constant delay for the medium connecting the senders output register with the receivers input register, which, being constant, can be ignored. The model can be adopted to reflect variability in the delay of the connecting medium by adjusting the minimal and maximal delay accordingly.

As a logical value of either 1 or 0 is assumed after the second register, the model nondeterministically resolves an unstable or unknown value, which is represented by a value of 2, either to logical 1 or to logical 0 in the first register, for reasons of simplicity.

The receiver tries to sample a value from the bus using a register. This register needs a stable value during an interval around a tick[8] from the receivers clock to correctly sample a value from the bus as described in Section 5.3.1. After a change on the bus, the value has to be stable for `setup` time units before a clock tick occurs, otherwise either 0 or 1 is nondeterministically sampled, a requirement that is taken care of by the model of the bus described in Section 6.3.2.

After a clock tick, a stable value has to remain stable for another `hold` time units, otherwise a boolean value is nondeterministically sampled. If there is no change in the value of `RxIn` during the hold period, the value has been stable, as completing a change and changing back to the original value takes more than `hold` time units. If the value has changed at the end of the hold period or if `RxIn` has the unstable value 2, a boolean value is nondeterministically sampled.

However, the sampling process may also be affected by a glitch. The local variable `lasterror`:=4(int[0,4]) is used to count the number of receiver clock cycles without a glitch. As described in Section 5.3.1, in any consecutive sequence of five receiver clock cycles, there may be at most one glitch. To realize this error model, the glitches are not modeled as changes on the bus, but may strike nondeterministically and affect a bit, which is modeled by assigning an unstable value to the bit and thus nondeterministically sample a boolean value for this bit. After such an error, the error may not occur again for the next four samples.

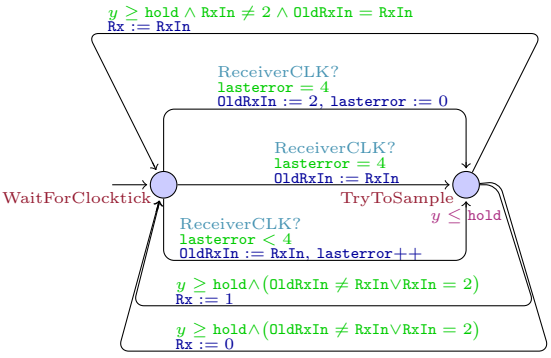

Figure 6.5: Model of the sampling process

Figure 6.5 shows the model of the sampling process.

[8] A *tick* designates a rising flank in the value of the clock.

Consecutive Register Rxx

This register delays the received value by one cycle of the receiver clock. As metastability is already resolved as described in Section 6.3.2, only the delay has to be modeled, which is done assuming the cycle-based boolean behavior as described in Section 5.3.1.

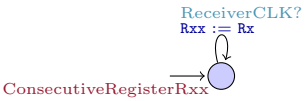

Figure 6.6: Model of the consecutive register Rxx

Figure 6.6 shows the model of the consecutive register Rxx.

6.4 Physical Layer Protocol Model

The removal of redundancy through voting and the bit clock alignment is described in Section 6.4.1. The assembly of the message bit stream and the adding of redundancy as well as the reception and the check of the message are described in Section 6.4.2.

6.4.1 Voting and Strobing

The model of the voting and strobing process assumes a cycle-based boolean behavior as described described in Section 5.3.1.

Voter

The last four sampled bits from the bus and the sample from the current receiver's clock cycle are stored in the local variables `window0`:=1(int[0,1]), `window1`:=1(int[0,1]), `window2`:=1(int[0,1]), `window3`:=1(int[0,1]) and `window4`:=1(int[0,1]). The variable `window0` always holds the newest value. In every cycle, the values of the `window` variables are shifted to the `window` variable with the next higher index, which means that the old value of `window4` is discarded, while `window0` is assigned the value of `Rxx`.

If the majority of the `window` variables contains a 1, `VV` is set to 1. If there is no majority containing a 1, `VV` is set to 0. As the size of the voting window is odd, there is always a clear majority. Other automata may synchronize to the new `VV` using the channel `ValueVoted`.

Figure 6.7 shows the model of the voting process.

Old Voted Value

In every cycle of the receivers clock, `VV` is stored in `OldVV` before a new `VV` is calculated, as shown in Figure 6.8. This value is needed to recognize the synchronization points for the bit clock alignment described in Section 5.3.3, as described in Section 6.4.1.

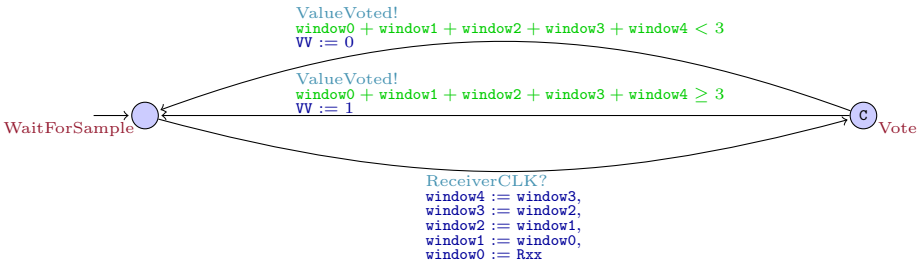

Figure 6.7: Model of the voting process

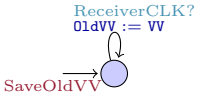

Figure 6.8: The old voted value is saved.

Bitstrobe Control

The strobecounter is used to select one voted value from each bit cell. To model the strobecounter, the local variable `Strobecounter`(int[1,8]) is used. In the beginning, `Strobecounter` has no default value, but is set nondeterministically.

When the new voted value, `VV`, is 0 and the voted value from the cycle before, `OldVV`, is 1 and `EnableSyncEdgeDetect` enables the bit clock alignment mechanism, `Strobecounter` is reset to 2 and the bit clock alignment mechanism is deactivated using `EnableSyncEdgeDetect`.

`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 the value 5 and `ValueVoted` signals that the voted value for this cycle of the receiver's clock has been found, `VV` is chosen as the value for `bstr`, if the bit clock alignment does not interfere. The channel `BitStrobed` allows other automata to synchronize to this event in order to use the new `bstr` value.

Figure 6.9 shows the model of the strobing process.

6.4.2 Protocol Control

The bit string that is transmitted is a frame in the static segment. The format of the bit stream of a message frame has to be generated by the sender and is expected by the receiver. The correct reception of the message has to be checked as well.

Sender Control

The following local variables are used:

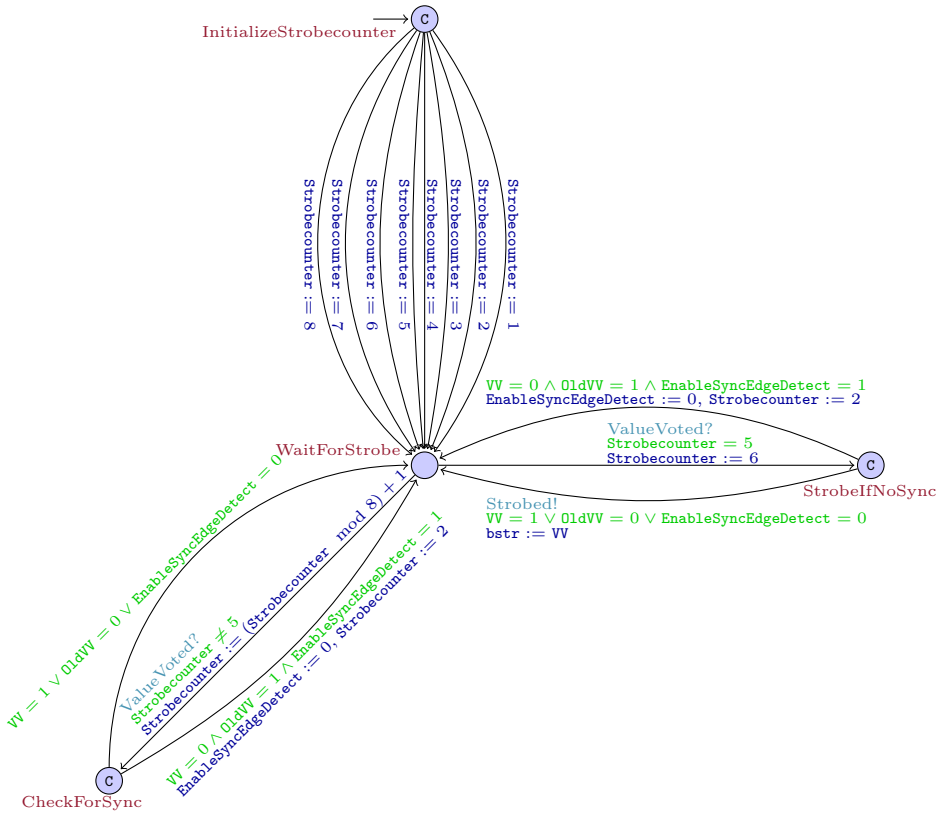

Figure 6.9: Model of the strobing process

- `TSScount`:=0(int[0,`TSSlimit`]) is used to count how many bits of the transmission start sequence have already been sent (see Section 5.3.2).
- `samplecounter`:=8(int[1,8]) is used to count how many times one bit of the bit stream has been sent[9].
- `bitcounter`:=1(int[1,8]) is used to count how many bits of one message byte have been sent[10].
- `bufferindex`:=0(int[0,`messagelength`]) is used to count how many bits of the message have been transmitted so far.

The format described in Section 5.3.2 is generated starting with the first tick of the sender's clock. Upon every tick of the sender's clock, `samplecounter` is incremented. Whenever one bit cell, i.e., a sequence of 8 consecutive identical samples, has been sent, the next bit is assigned to `Tx`.

[9] As described in Section 5.3.3, every bit is sent 8 times.
[10] As described in Section 5.3.2, the message bytes are separated by byte start sequences.

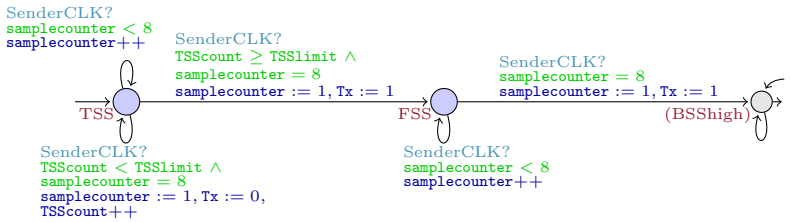

Figure 6.10: Simulation of the start of the transmission

First, a transmission start sequence and a frame start sequence are modeled, as shown in Figure 6.10.

The message is generated nondeterministically, i.e., whenever a message bit is to be sent, it is nondeterministically decided if a logical 1 or a logical 0 will be sent and if this bit is to be verified by the receiver, in which case `savedindex` is assigned a number indicating how many bits have been send before, which is derived from `bufferindex`, and `savedTx` is assigned the value of the bit that is sent, `Tx`.

The model of the message bytes and the byte start sequences that precede each byte is shown in Figure 6.11.

In the end, a frame end sequence is modeled, as shown in Figure 6.12.

Receiver Control

Several local variables are used:

- `TSScount`:=0(int[0,`TSSlimit` + 1]) is used to count how many bits of the transmission start sequence have already been received (see Section 5.3.2).
- `BYTEbitcounter`:=1(int[1,8]) is used to count how many bits of one message byte have been received[10].
- `bufferindex`:=0(int[0,`messagelength` − 1]) is used to count how many bits of the message have been received so far.

When the channel `Strobed` signals that a new bit has been strobed (see Section 5.3.3), the receiver tries to check if this bit is consistent with the expected format of the frame, as described in Section 5.3.2. As soon as a received bit has not the expected value, the error state `DECerr` is entered.

First, the reception of a transmission start sequence followed either by a frame start sequence or the first bit of a byte start sequence is modeled, as shown in Figure 6.13.

The received transmission start sequence is accepted if it contains at least `TSSmin` bits and at most `TSSlimit` + 1 bits, as `TSScount` is the number of bits that have been received in this transmission start sequence: a bit of the transmission start sequence may be received if up to `TSSlimit` bits have been received before.

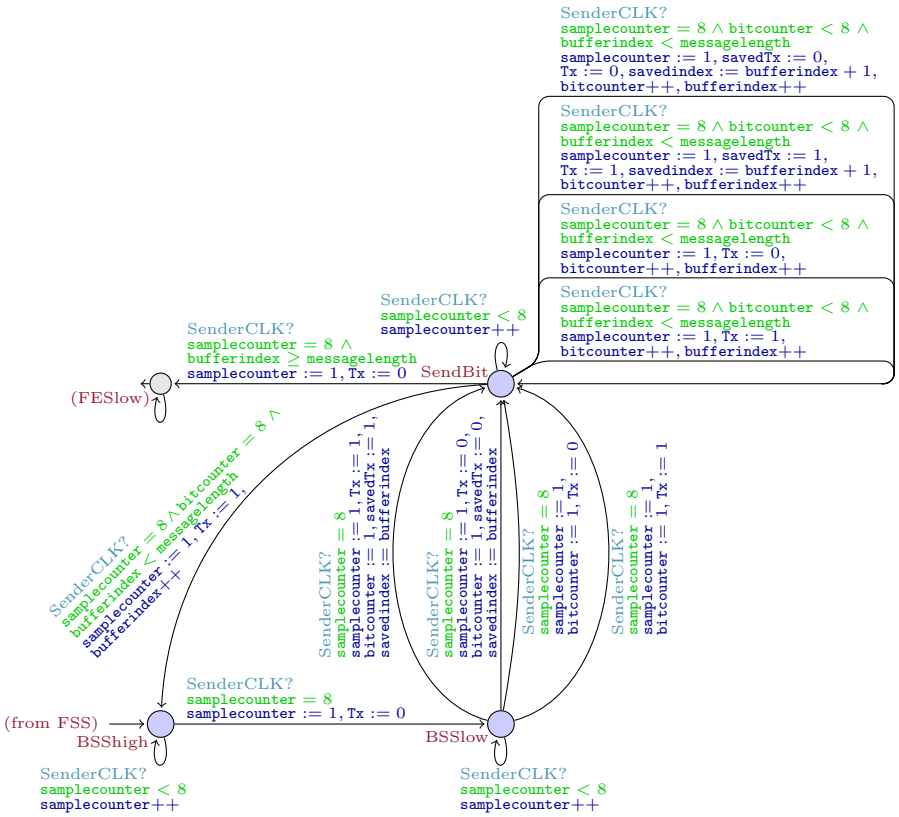

Figure 6.11: Simulation of the transmission of the message bytes

During the reception of the transmission start sequence or of a byte start sequence, the variable `EnableSyncEdgeDetect` is used to enable the bit clock alignment mechanism, as is described in Section 6.4.1.

While the receiver is expecting to receive message bits, the number of bits is counted using the variable `BYTEbitcounter` to find out when the next byte start sequence is to be expected. The overall number of received message bits is counted in `bufferindex`. When `savedindex` indicates that the next message bit is to be verified, the received `bstr` value is compared to `savedTx`. The model of the reception of the message bytes and their preceding byte start sequences is shown in Figure 6.14.

In the end, the reception of a frame end sequence is modeled, as shown in Figure 6.15.

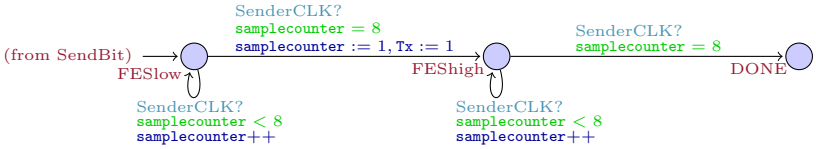

Figure 6.12: Simulation of the end of the transmission

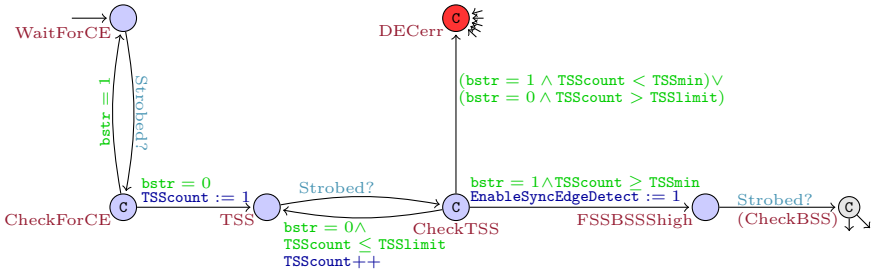

Figure 6.13: Simulation of the start of the reception

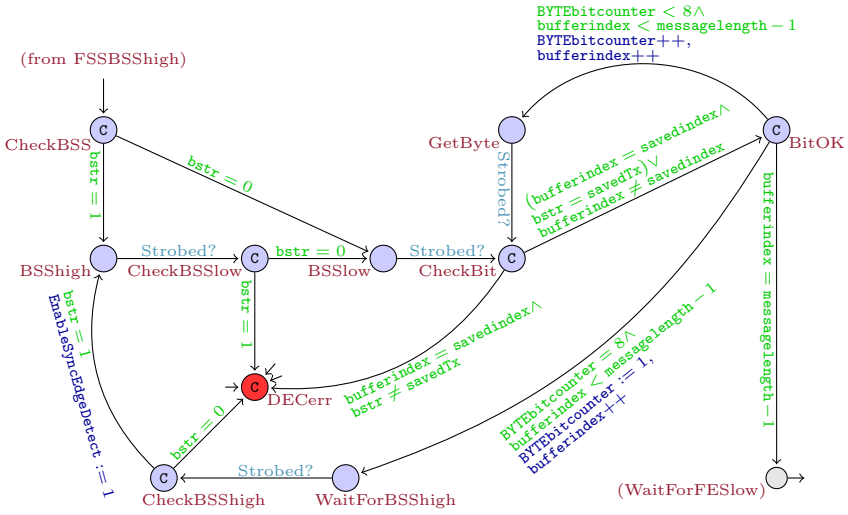

Figure 6.14: Simulation of the reception of the message bytes

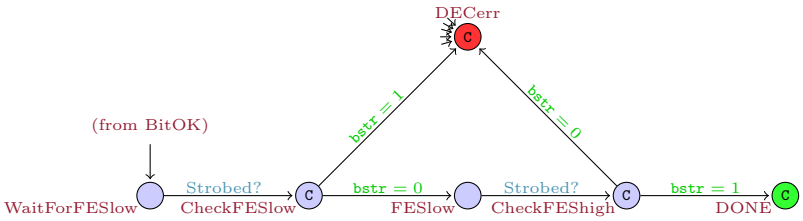

Figure 6.15: Simulation of the end of the reception

Part III

Discussion

Chapter 7

Experimental Results

The effectiveness of ZSDs for timed reachability model checking has been evaluated on several benchmarks, comparing Uppaal[BDL04] against a prototype model checker using ZSDs or TSSs. All Benchmarks were executed on an AMD Opteron processor with 2.6 GHz running Linux, limited to 4 GiB of memory and limited to 3 month of computation time.

7.1 FlexRay

Several experiments on model checking FlexRay's physical layer protocol, modeled as described in Chapter 6, have been made using three different tools:

- Uppaal 4.0.10
- Our prototype using ZSDs
- Our prototype using TSSs

The checked property, which turned out to hold for all checked instances, was

$$\texttt{A[]!Receiver\_Control.DECerr},$$

meaning the error state is never reached.

7.1.1 Results with Uppaal

The ability of Uppaal to check the FlexRay model was tested in different configurations, as shown in Table 7.1. The most effective configuration used the parameters -C -o 0 -S 2 to use DBMs, breadth first search, and aggressive state space reduction. However, all configurations where able to check the correctness of a message transfer if the message contained up to 3 bytes, but none was able to verify the correctness of a message transfer for 4 bytes due to the inability of Uppaal to handle more than 4 GiB of memory. Unsurprisingly, breadth first search was significantly faster than depth first search. Note that for depth first search the number of explored states listed

in Table 7.1 is the output of UPPAAL. The number is not very informative, as the negative numbers indicate that UPPAAL's internal counter overflowed.

7.1.2 Results with our Prototype using TSSs

For fairness of comparison, our prototype was also limited to 4 GiB of memory. In order to be able to evaluate the effectiveness of the ZSDs in comparison to the traditional TSSs, the performance our prototype using TSSs is shown in Table 7.2. The model checker is identical to the one using ZSDs, the only difference is the representation of the timed state space. Using TSSs, only a message containing a single byte could be verified.

7.1.3 Results with our Prototype using ZSDs

Our prototype using ZSDs was able to check the correctness of a FlexRay message transfer of 10 byte, as shown in Table 7.3. Experiments for larger payloads have not been conducted, but probably would have succeeded before eventually reaching very large message lengths that would run into the 3 month time limit or the memory limit of 4 GiB. Note that the prototype uses BDDs, which causes the reordering heuristic used by the BDD library to consume a large amount of time. As the memory consumption increases with growing message length, the reordering is done more often in order to save memory by finding variable orders that result in a more compact representation for the BDDs. This causes the runtime for long messages to vary, as the reordering heuristics introduce a lot of variance in the runtime.

ZSDs thus demonstrate their ability to represent the timed state space more space efficient as done by UPPAAL, however being significantly slower than the highly optimized model checker UPPAAL. These results demonstrate the effectiveness of ZSDs, as compared to TSSs. While the traditional TSSs are just approximately 1.4% faster than ZSDs on the FlexRay example, TSSs cannot handle messages with a length of more than 1 byte. ZSDs, however, can handle far larger messages while being only slightly slower.

7.1.4 Conclusions for FlexRay

Judging from the results shown in Tables 7.1, 7.2, and 7.3, ZSDs have great potential in handling data-intensive systems. The prototype is considerably slower than UPPAAL, but a comparison of two instances of the prototype model checker, one using ZSDs, the other using TSSs, shows that ZSDs are almost as fast as TSSs. Thus, the higher speed of UPPAAL does not stem solely from UPPAAL's use of a TSSs-like data structure, but mostly from other optimizations as described, e.g., by Larsen et al. in [LLPY97].

However, ZSDs prove to be able to handle larger examples, as the prototype using ZSDs can handle the FlexRay model for FlexRay message sizes of up to 10 bytes, and possibly even larger ones, while UPPAAL can only handle message sizes of up to 3 bytes. On the other hand, the prototype using TSSs can just handle a message size of a single byte. In the end, the prototype using ZSDs can handle message sizes of up to

5 bytes in less than one day, messages sizes of up to 7 bytes in less than 4 days, and message sizes of up to 10 bytes in less than 12 days. ZSDs have not been tested on larger message sizes because the required computational resources to find the maximal possible message size were expected to exceed the scope of this work.

7.2 Fischer

The Fischer mutual exclusion protocol[AL91], a standard benchmark, was used to compare the performance of three tools:

- Uppaal 4.0.11
- Our prototype using ZSDs
- Our prototype using TSSs

The verified property was mutual exclusion, i.e., no two processes are in their critical section at the same time. The Fischer implementation for the benchmark is taken from [EFGP10].

7.2.1 Results with Uppaal

The performance of Uppaal on the Fischer benchmark was tested extensively using different configuration parameters, as shown in Table 7.5. The fastest configuration for Uppaal turned out to be "-C -o 0 -S 1", i.e., breath first search using DBMs and conservative state space optimization. Aggressive state space consumption was not helpful, as the state space seems to be relatively small. Unsurprisingly, depth first search was not very effective.

7.2.2 Results with our Prototype using TSSs

A fair comparison of the performance of the data structures can only be based on experiments using the same model checking framework. Surprisingly, as shown in Table 7.6, our prototype is significantly slower when using TSSs instead of ZSDs (see Table 7.7). For the Fischer benchmark with 6 processes, our prototype using TSSs hit the limit of 3 month of computation time, indicating that for our prototype, TSSs are a bad choice when verifying Fischer.

7.2.3 Results with our Prototype using ZSDs

As the results from Table 7.7 show, the Fischer mutual exclusion protocol benchmark turned out to be quite hard for our prototype model checker. The ZSD based state space representation ran out of memory for Fischer with 8 processes. Moreover, our prototype was considerably slower than Uppaal, even compared to Uppaal started with the worst configuration parameters. Using ZSDs, the size of the state space blew

up quite fast. The state space seems to consist of a small discrete part and a very large timed part.

Interestingly, the ZSD based prototype was significantly faster than the TSS based one.

7.3 Gear Production Stack

The Gear Production Stack (GPS) benchmark taken from [EFGP10] models a manufacturing plant with communicating processing stations. The stations process gears in a sequential manner. Three tools were compared on the GPS benchmark.

- Uppaal 4.0.11
- Our prototype using ZSDs
- Our prototype using TSSs

The bounded liveness property that each gear is processed in a certain amount of time is verified.

7.3.1 Results with Uppaal

Table 7.9 shows the result of running Uppaal on the GPS benchmark with various configuration parameters. The configuration "-C -o 0 -S 2", i.e. using DBMs, breadth first search, and aggressive state space optimization, turned out to be the most effective one. Depth first exploration combined with aggressive state space optimization is very slow: checking GPS with 6 stations took 2 days, GPS with 7 stations hit the limit of 3 month of computation time. However, with conservative state space optimization, uppaal can handle GPS with up to 14 stations before running out of memory, while aggressive state space optimization allows Uppaal to handle GPS with up to 15 stations for breadth first search, causing depth first search to quickly hit the time limit.

7.3.2 Results with our Prototype using TSS

Our prototype using TSS can handle GPS with up to 12 stations, before running out of memory, as shown in Table 7.10.

7.3.3 Results with our Prototype using ZSDs

The prototype using ZSDs can handle larger instances than the TSS based one, as shown in Table 7.11. Being able to handle GPS with up to 14 stations, the ZSD based prototype is as effective as Uppaal with conservative state space optimization. However, the ZSD based prototype is considerably slower than Uppaal, and also slower than the prototype using TSSs.

7.4 Leader Election

The Leader Election benchmark taken from [EFGP10] describes a protocol used to elect a leader in a ring. Again, this benchmark is used to compare three tools.

- Uppaal 4.0.11
- Our prototype using ZSDs
- Our prototype using TSSs

It is checked that a leader has been elected after a certain amount of time.

7.4.1 Results using Uppaal

Leader Election is checked very efficiently by Uppaal, as shown in Table 7.13. The fastest configuration for Uppaal was "-C -o 0 -S 1", i.e., using DBMs, breadth first search and conservative state space optimization. Again, depth first search in combination with aggressive state space optimization turned out to be slow.

7.4.2 Results with the Prototype using TSS

As Table 7.14 shows, the prototype using TSS can handle the Leader Election benchmark with up to 7 participants.

7.4.3 Results with the Prototype using ZSDs

The prototype using ZSDs can handle the Leader Election benchmark for all instances we tested, as shown in Table 7.15. While Leader Election with up to 6 participants is checked faster when TSSs are used, for Leader Election with 7 and more participants, the ZSD based prototype is faster than the TSS based one. This indicates that the growing size of the state space is better handled by ZSDs, while for smaller state spaces, TSSs are more efficient.

7.5 Summary

Only the FlexRay benchmark demonstrated a clear superiority of the ZSD based prototype over the TSS based one and even over Uppaal, in terms of the size of instance of the benchmark that can be handled. For the Gear Production Stack, ZSDs still are able to handle larger instances than TSSs, and Uppaal needs an aggressive state space optimization scheme to be able to handle also a slightly larger instance than the instances the ZSD based prototype can handle, without this scheme Uppaal can handle the benchmark for the same number of participants that the ZSD based prototype. For the Leader Election benchmark, ZSDs can handle larger instances than than TSSs, but Uppaal seems to have no problem with large instances as well. For the Fischer benchmark, ZSDs can handle only small instances, whereas Uppaal can also handle

larger ones. However, the TSS based prototype can handle only very small instances in a reasonable amount of time.

Unsurprisingly, UPPAAL was generally the fastest tool on all benchmarks. For FlexRay and Gear Production Stack, the TSS based prototype was faster than the ZSD based one, while for Fischer, the ZSD based prototype was faster than the TSS based one. Interestingly, for Leader Election, the TSS based prototype was faster for small instances, while the ZSD based prototype was faster for larger instances.

parameters	message length	time (seconds)	states explored	states stored
-o 0 -S 1	1 byte	76.33	8,800,916	7,253,864
-o 0 -S 1	2 bytes	268.64	30,059,181	25,999,231
-o 0 -S 1	3 bytes	594.22	65,706,463	56,519,926
-o 0 -S 1	4 bytes	823.39	memout	
-C -o 0 -S 1	1 byte	68.44	8,800,916	7,253,864
-C -o 0 -S 1	2 bytes	250.33	30,059,181	25,999,231
-C -o 0 -S 1	3 bytes	545.53	65,706,463	56,519,926
-C -o 0 -S 1	4 bytes	601.72	memout	
-o 0 -S 2	1 byte	75.59	8,800,916	6,979,613
-o 0 -S 2	2 bytes	263.50	30,059,181	24,999,598
-o 0 -S 2	3 bytes	591.82	65,706,463	54,341,183
-o 0 -S 2	4 bytes	700.50	memout	
-C -o 0 -S 2	1 byte	69.29	8,800,916	6,979,613
-C -o 0 -S 2	2 bytes	245.59	30,059,181	24,999,598
-C -o 0 -S 2	3 bytes	536.52	65,706,463	54,341,183
-C -o 0 -S 2	4 bytes	655.64	memout	
-o 1 -S 1	1 byte	5,548.36	608,028,007	7,272,142
-o 1 -S 1	2 bytes	20,533.01	-1,990,507,322	26,083,605
-o 1 -S 1	3 bytes	45,606.86	749,784,845	56,712,076
-o 1 -S 1	4 bytes	56,139.61	memout	
-C -o 1 -S 1	1 byte	4,811.09	608,028,007	7,272,142
-C -o 1 -S 1	2 bytes	18,679.38	-1,990,507,322	26,083,605
-C -o 1 -S 1	3 bytes	41,867.40	749,784,845	56,712,076
-C -o 1 -S 1	4 bytes	42,433.53	memout	
-o 1 -S 2	1 byte	5,193.38	609,506,149	6,979,451
-o 1 -S 2	2 bytes	21,204.23	-1,984,775,312	25,015,924
-o 1 -S 2	3 bytes	44,896.59	762,434,867	54,384,365
-o 1 -S 2	4 bytes	56,533.55	memout	
-C -o 1 -S 2	1 byte	4,776.98	609,506,149	6,979,451
-C -o 1 -S 2	2 bytes	19,124.62	-1,984,775,312	25,015,924
-C -o 1 -S 2	3 bytes	42,987.07	762,434,867	54,384,365
-C -o 1 -S 2	4 bytes	45,165.31	memout	

Table 7.1: Checking the FlexRay Model with UPPAAL 4.0.10.

message length	time (seconds)	steps	size
1 byte	278.61	4,692,982	2,273,253
2 bytes	1,427.08	memout	

Table 7.2: Checking the FlexRay model with our prototype using TSSs.

message length	time (seconds)	steps	size
1 byte	282.40	539,079	15,330
2 bytes	4,470.75	6,652,400	58,459
3 bytes	14,341.00	16,314,890	88,911
4 bytes	45,376.26	30,323,660	111,903
5 bytes	50,465.01	49,632,336	141,540
6 bytes	298,377.91	73,249,127	174,703
7 bytes	223,513.48	102,333,790	208,172
8 bytes	910,793.86	99,623,479	271,653
9 bytes	795,268.45	129,175,628	302,717
10 bytes	970,099.11	161,907,581	335,153

Table 7.3: Checking the FlexRay model with our prototype using ZSDs.

parameters	# processes	time (seconds)	states explored	states stored
-o 0 -S 1	3	0.00	71	65
-o 0 -S 1	4	0.01	268	220
-o 0 -S 1	5	0.01	977	727
-o 0 -S 1	6	0.06	3,458	2,378
-o 0 -S 1	7	0.26	11,951	7,737
-o 0 -S 1	8	1.18	40,536	25,080
-C -o 0 -S 1	3	0.00	71	65
-C -o 0 -S 1	4	0.00	268	220
-C -o 0 -S 1	5	0.02	977	727
-C -o 0 -S 1	6	0.04	3,458	2,378
-C -o 0 -S 1	7	0.20	11,951	7,737
-C -o 0 -S 1	8	0.91	40,536	25,080
-o 0 -S 2	3	0.00	162	43
-o 0 -S 2	4	0.01	638	169
-o 0 -S 2	5	0.03	2360	611
-o 0 -S 2	6	0.15	8,394	2,117
-o 0 -S 2	7	0.70	29,044	7,155
-o 0 -S 2	8	3.30	98,494	23,793
-C -o 0 -S 2	3	0.00	162	43
-C -o 0 -S 2	4	0.00	638	169
-C -o 0 -S 2	5	0.02	2,360	611
-C -o 0 -S 2	6	0.11	8,394	2,117
-C -o 0 -S 2	7	0.54	29,044	7,155
-C -o 0 -S 2	8	2.55	98,494	23,793

Table 7.4: Checking Fischer with UPPAAL 4.0.11, breadth first search

parameters	# processes	time (seconds)	states explored	states stored
-o 1 -S 1	3	0.00	65	65
-o 1 -S 1	4	0.00	229	220
-o 1 -S 1	5	0.01	867	727
-o 1 -S 1	6	0.06	3,568	2,378
-o 1 -S 1	7	0.34	16,371	7,737
-o 1 -S 1	8	2.14	75,184	25,080
-C -o 1 -S 1	3	0.00	65	65
-C -o 1 -S 1	4	0.00	229	220
-C -o 1 -S 1	5	0.01	867	727
-C -o 1 -S 1	6	0.04	3,568	2,378
-C -o 1 -S 1	7	0.27	16,371	7,737
-C -o 1 -S 1	8	1.60	75,184	25,080
-o 1 -S 2	3	0.00	192	37
-o 1 -S 2	4	0.01	1,482	144
-o 1 -S 2	5	0.14	13,332	528
-o 1 -S 2	6	1.42	104,651	1,865
-o 1 -S 2	7	22.19	1,296,540	6,499
-o 1 -S 2	8	411.19	19,297,757	22,099
-C -o 1 -S 2	3	0.00	192	37
-C -o 1 -S 2	4	0.01	1,482	144
-C -o 1 -S 2	5	0.11	13,332	528
-C -o 1 -S 2	6	1.19	104,651	1,865
-C -o 1 -S 2	7	17.80	1,296,540	6,499
-C -o 1 -S 2	8	321.72	19,297,757	22,099

Table 7.5: Checking Fischer with UPPAAL 4.0.11, depth first search

# processes	time (seconds)	steps	size
3	0.02	287	65
4	1.40	2,925	220
5	1,106.39	35,334	727
6	timeout		

Table 7.6: Checking Fischer with our prototype using TSSs.

# processes	time (seconds)	steps	size
3	0.02	434	228
4	0.22	4,401	1,846
5	7.17	88,465	21,950
6	351.10	2,229,875	335,866
7	63,350.95	69,385,200	6,223,519
8	14,670.28	memout	

Table 7.7: Checking Fischer with our prototype using ZSDs.

parameters	# stations	time (seconds)	states explored	states stored
-o 0 -S 1	6	0.02	2,185	2,185
-o 0 -S 1	7	0.06	6,559	6,559
-o 0 -S 1	8	0.24	19,681	19,681
-o 0 -S 1	9	0.90	59,047	59,047
-o 0 -S 1	10	3.25	177,145	177,145
-o 0 -S 1	11	11.52	531,439	531,439
-o 0 -S 1	12	40.70	1,594,321	1,594,321
-o 0 -S 1	13	155.67	4,782,967	4,782,967
-o 0 -S 1	14	491.87	14,348,905	14,348,905
-o 0 -S 1	15	1,265.97	memout	
-C -o 0 -S 1	6	0.01	2,185	2,185
-C -o 0 -S 1	7	0.06	6,559	6,559
-C -o 0 -S 1	8	0.22	19,681	19,681
-C -o 0 -S 1	9	0.83	59,047	59,047
-C -o 0 -S 1	10	3.04	177,145	177,145
-C -o 0 -S 1	11	10.99	531,439	531,439
-C -o 0 -S 1	12	39.46	1,594,321	1,594,321
-C -o 0 -S 1	13	154.89	4,782,967	4,782,967
-C -o 0 -S 1	14	467.34	14,348,905	14,348,905
-C -o 0 -S 1	15	1,266.76	memout	
-o 0 -S 2	6	0.02	2,185	364
-o 0 -S 2	7	0.06	6,559	1,093
-o 0 -S 2	8	0.23	19,681	3,280
-o 0 -S 2	9	0.85	59,047	9,841
-o 0 -S 2	10	3.00	177,145	29,524
-o 0 -S 2	11	10.95	531,439	88,573
-o 0 -S 2	12	38.32	1,594,321	265,720
-o 0 -S 2	13	135.72	4,782,967	797,161
-o 0 -S 2	14	464.61	14,348,905	2,391,484
-o 0 -S 2	15	1,687.84	43,046,719	7,174,453
-o 0 -S 2	16	1689.42	memout	
-C -o 0 -S 2	6	0.02	2,185	364
-C -o 0 -S 2	7	0.06	6,559	1,093
-C -o 0 -S 2	8	0.22	19,681	3,280
-C -o 0 -S 2	9	0.80	59,047	9,841
-C -o 0 -S 2	10	2.84	177,145	29,524
-C -o 0 -S 2	11	10.52	531,439	88,573
-C -o 0 -S 2	12	37.55	1,594,321	265,720
-C -o 0 -S 2	13	135.56	4,782,967	797,161
-C -o 0 -S 2	14	468.68	14,348,905	2,391,484
-C -o 0 -S 2	15	1,648.85	43,046,719	7,174,453
-C -o 0 -S 2	16	1,612.45	memout	

Table 7.8: Checking Gear Production Stack with UPPAAL 4.0.11, breadth first search

parameters	# stations	time (seconds)	states explored	states stored
-o 1 -S 1	6	0.02	2,185	2,185
-o 1 -S 1	7	0.07	6,559	6,559
-o 1 -S 1	8	0.23	19,681	19,681
-o 1 -S 1	9	0.89	59,047	59,047
-o 1 -S 1	10	3.25	177,145	177,145
-o 1 -S 1	11	11.62	531,439	531,439
-o 1 -S 1	12	40.39	1,594,321	1,594,321
-o 1 -S 1	13	158.48	4,782,967	4,782,967
-o 1 -S 1	14	469.45	14,348,905	14,348,905
-o 1 -S 1	15	1,236.39	memout	
-C -o 1 -S 1	6	0.02	2,185	2,185
-C -o 1 -S 1	7	0.06	6,559	6,559
-C -o 1 -S 1	8	0.21	19,681	19,681
-C -o 1 -S 1	9	0.81	59,047	59,047
-C -o 1 -S 1	10	3.03	177,145	177,145
-C -o 1 -S 1	11	11.18	531,439	531,439
-C -o 1 -S 1	12	39.57	1,594,321	1,594,321
-C -o 1 -S 1	13	146.12	4,782,967	4,782,967
-C -o 1 -S 1	14	525.98	14,348,905	14,348,905
-C -o 1 -S 1	15	1,376.56	memout	
-o 1 -S 2	6	216,313.25	55,181,318	364
-o 1 -S 2	7	timeout		
-C -o 1 -S 2	6	172,436.73	55,181,318	364
-C -o 1 -S 2	7	timeout		

Table 7.9: Checking Gear Production Stack with UPPAAL 4.0.11, depth first search

# stations	time (seconds)	steps	size
6	0.19	2,549	2,185
7	0.70	7,652	6,559
8	2.70	22,961	19,681
9	10.50	68,888	59,047
10	39.73	206,669	177,145
11	147.56	620,012	531,439
12	546.20	1,860,041	1,594,321
13	1,105.99	memout	

Table 7.10: Checking Gear Production Stack with our prototype using TSSs.

# stations	time (seconds)	steps	size
6	0.27	2,549	12
7	1.19	7,652	14
8	5.65	22,961	16
9	26.00	68,888	18
10	110.76	206,669	20
11	451.51	620,012	22
12	1778.62	1,860.041	24
13	11,108.42	5,580,128	26
14	26,592.94	16,740,389	28
15	20,462.43	memout	

Table 7.11: Checking Gear Production Stack with our prototype using ZSDs.

parameters	# participants	time (seconds)	states explored	states stored
-o 0 -S 1	4	0.02	1,197	661
-o 0 -S 1	5	0.09	7,398	3,406
-o 0 -S 1	6	0.61	42,482	16,717
-o 0 -S 1	7	4.24	227,253	79,757
-o 0 -S 1	8	27.37	1,185,818	374,786
-o 0 -S 1	9	176.01	5,905,852	1,746,863
-C -o 0 -S 1	4	0.01	1,197	661
-C -o 0 -S 1	5	0.07	7,398	3,406
-C -o 0 -S 1	6	0.51	42,482	16,717
-C -o 0 -S 1	7	3.60	227,253	79,757
-C -o 0 -S 1	8	23.35	1,185,818	374,786
-C -o 0 -S 1	9	149.13	5,905,852	1,746,863
-o 0 -S 2	4	0.02	1,324	172
-o 0 -S 2	5	0.09	8,544	898
-o 0 -S 2	6	0.66	49,837	4,332
-o 0 -S 2	7	4.79	277,031	22,258
-o 0 -S 2	8	32.26	1,475,243	107,658
-o 0 -S 2	9	212.89	7,610,621	542,187
-C -o 0 -S 2	4	0.01	1,324	172
-C -o 0 -S 2	5	0.08	8,544	898
-C -o 0 -S 2	6	0.56	49,837	4,332
-C -o 0 -S 2	7	4.08	277,031	22,258
-C -o 0 -S 2	8	27.83	1,475,243	107,658
-C -o 0 -S 2	9	181.97	7,610,621	542,187

Table 7.12: Checking Leader Election with UPPAAL 4.0.11, breadth first search

parameters	# participants	time (seconds)	states explored	states stored
-o 1 -S 1	4	0.02	1,499	661
-o 1 -S 1	5	0.13	11,262	3,406
-o 1 -S 1	6	1.15	78,463	16,717
-o 1 -S 1	7	9.33	493,808	79,757
-o 1 -S 1	8	71.52	2,986,059	374,786
-o 1 -S 1	9	516.14	17,175,675	1,746,863
-C -o 1 -S 1	4	0.02	1,499	661
-C -o 1 -S 1	5	0.11	11,262	3,406
-C -o 1 -S 1	6	0.96	78,463	16,717
-C -o 1 -S 1	7	7.80	493,808	79,757
-C -o 1 -S 1	8	59.41	2,986,059	374,786
-C -o 1 -S 1	9	436.11	17,175,675	1,746,863
-o 1 -S 2	4	0.06	8,123	229
-o 1 -S 2	5	1.63	202,166	1,333
-o 1 -S 2	6	49.28	4,947,205	7,299
-o 1 -S 2	7	1,695.46	138,249,789	38,752
-o 1 -S 2	8	59,450.31	-230,436,488	194,972
-o 1 -S 2	9	2,361,935.18	1,171,666,415	987,170
-C -o 1 -S 2	4	0.04	8,123	229
-C -o 1 -S 2	5	1.38	202,166	1,333
-C -o 1 -S 2	6	42.07	4,947,205	7,299
-C -o 1 -S 2	7	1,471.18	138,249,789	38,752
-C -o 1 -S 2	8	52,818.09	-230,436,488	194,972
-C -o 1 -S 2	9	2,107,786.25	1,171,666,415	987,170

Table 7.13: Checking Leader Election with UPPAAL 4.0.11, depth first search

# participants	time (seconds)	steps	size
4	0.06	742	661
5	0.60	4,185	3,406
6	13.70	22,746	16,717
7	3,399.51	123,251	79,757
8	113,610.26	memout	

Table 7.14: Checking Leader Election with our prototype using TSSs.

# participants	time	steps	size
4	0.24	777	164
5	2.45	4,805	749
6	21.70	30,360	3,935
7	207.52	195,383	19,514
8	3,239.51	1,379,837	107,029
9	34,006.47	12,415,305	1,065,425

Table 7.15: Checking Leader Election with our prototype using ZSDs.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis formally describes two approaches for representing the timed state space, timed state sets (TSSs), and the novel zone state diagrams (ZSDs). Furthermore, it presents algorithms for implementing ZSDs for use in timed reachability model checking.

As a case study, the thesis describes the state-of-the-art vehicle bus FlexRay and presents a model of FlexRay's physical layer protocol modeled as extended timed automata.

Finally, in order to evaluate ZSDs, the thesis compares the performance of a prototype model checker using ZSDs to Uppaal's performance on the task of checking whether the error state of the FlexRay model is reachable. The measured running times demonstrate the need for further optimization, while the ability of ZSDs to handle large message sizes underlines the potential of the approach. Nevertheless, in an industrial setting for safety critical systems, testing often takes several month. So, verification times of several weeks still are a huge improvement over not being able to verify at all and having to rely on testing or manual verification that also takes a lot of time. Thus, ZSDs allow to improve the current quality-control procedures for industry-sized physical layer protocols through the use of automatic formal verification.

Three other benchmarks, Fischer, Gear Production Stack, and Leader Election, are also used for evaluation of the prototype model checker using ZSDs. Concerning speed, Uppaal is superior to the prototype model checker using ZSDs for all three benchmarks, and can handle instances of similar size. However, when compared to a prototype model checker using TSSs, the ZSD-based variant is always capable of handling larger instances of the benchmarks, sometimes being faster as well. This leads to the conclusion that Uppaal's good performance stems from optimizations and is not due to its use of a TSS-like data structure for representing the timed state space. In a fair comparison, ZSDs consistently outperform TSSs in terms of the size that can be handled.

These results justify further investigation into data structures like ZSDs that reverse

the order of the mapping from locations to zones as well as the use of BDDs for representing large sets of locations for real-time model checking.

8.2 Future Work

Several lines of research and development can be based on this work in the future.

Optimizing ZSDs. ZSDs do store more information as needed: it is possible that location l is stored in a ZSD for two DBMs, z, z'. If a third DBM, z'' such that $[\![z'']\!] \subset [\![z]\!] \cup [\![z']\!]$, is added and should map to l as well, the ZSD stores all three pairs $(z, l), (z', l)$, and (z'', l), while a ZSD storing just (z, l) and (z', l) would actually contain the same location / clock valuation pairs. It could also be the case that if $[\![z]\!] \subset [\![z']\!]$ and the ZSD stores (z', l), (z, l) is added to the ZSD without adding new information about location / clock valuation pairs. If this is avoided, space consumption would be even smaller. However, representing non-convex sets of clock valuations efficiently or even checking for (partial) inclusion of one DBM in another efficiently is hard. There seems to be some potential for optimizations by efficiently reducing the redundancy introduced by such phenomenons.

Greatest fixed points with ZSDs. So far only least fixed point constructions are used by the prototype using ZSDs, the performance of ZSDs when using a greatest fixed point construction has yet to be evaluated.

Symbolic ZSD model checking algorithm. The reachability model checking algorithm in the prototype is still traditional, working through a set of location / DBM pairs (encoded like a TSS), adding newly found states to this set as well as to the ZSD storing the visited states. This algorithm cannot fully exploit the ZSD representation, as ZSDs are just used for the visited states. An algorithm working on the whole set of reachable states or of newly reached states simultaneously, e.g., by using a symbolical *post* operator to execute all enabled transitions in one single step, possibly even working on a ZSD representation directly, would speed up the model checking procedure considerably.

Fully symbolic state space representation. The idea of reversing the hierarchy, i.e., the unconventional mapping from timing information to location data, could be advanced to a flexible order between the individual constraints. This would enable the use of heuristics for efficient orderings over all constraints, time constraints as well as constraints on the boolean variables encoding the locations now stored in BDDs. Such a flexible data structure would not have the strict separation between timing and location data, thus probably being even more space efficient by thoroughly exploiting redundancy.

Investigating FlexRay's physical layer protocol. The FlexRay model from Chapter 6 can be a basis for further investigation of FlexRay. Several parameter configurations of this model could be evaluated to observe the effects on the fault tolerance or on the minimal requirements for other parameters. The model itself could also be furthermore refined, speeding up verification (thus making it possible to check larger payloads of up to the maximal 262 bytes) or making room for modeling additional aspects of the FlexRay protocol.

Appendix A

Model Variant

The model of FlexRay used in this thesis is a little bit less permissive than required by [Fle05]: In Section 6.4.2 a `frame start sequence` combined with the first half of a `byte start sequence` is required to be received as either one or two consecutive high bits, [Fle05, Section 3.2.6.3] allows one to three consecutive high bits. To adjust the model, the automaton shown in Figure 6.14 has to be replaced by the automaton shown in Figure A.1, where two states, CheckFSS and BSS, have been added. Moreover,

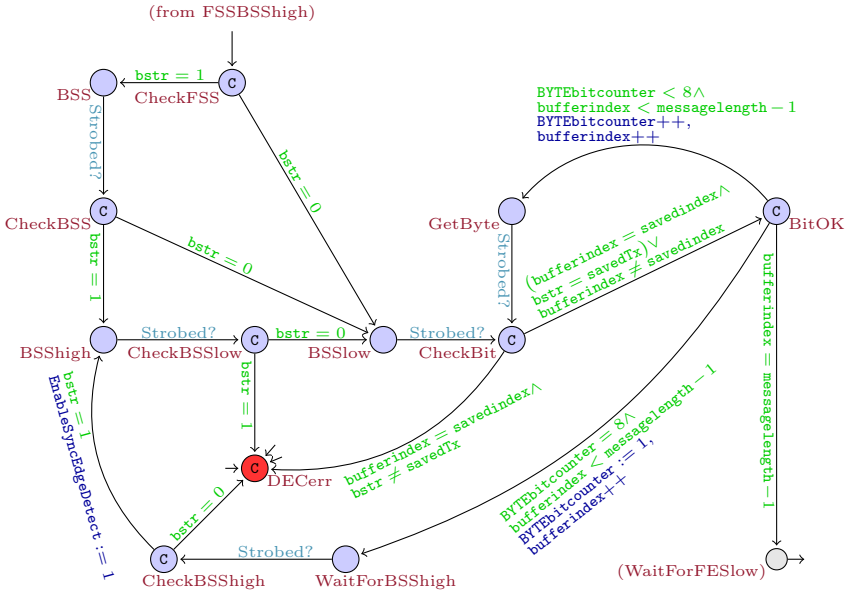

Figure A.1: Simulation of the reception of the message bytes

Figure 6.13 has to be replaced by Figure A.2, as A.1 has a new entry point.

However, the stricter requirements of Section 6.4.2 where met in the evaluation of the model as described in Chapter 7. Thus, either [Fle05, Section 3.2.6.3] is overly

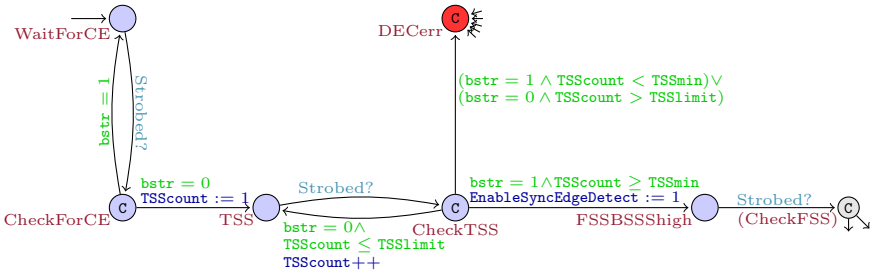

Figure A.2: Simulation of the start of the reception

permissive, or the variant of a three high bit `FSSBSS` is designed to compensate for an additional source of errors that was not considered in this thesis, e.g., truncation as described in [Fle05, Section 3.2.5].

Bibliography

[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] E. Alkassar, P. Böhm, and S. Knapp. Formal Correctness of an Automotive Bus Controller Implementation at Gate-Level. In *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] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking for Real-Time Systems. In *Fifth Annual IEEE Symposium on Logic in Computer Science (LICS '90)*, pages 414–425. IEEE Computer Society, 1990.

[AD90] R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In *17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

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

[AL91] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. In *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *LNCS*, pages 1–27. Springer, 1991.

[Alu99] R. Alur. Timed Automata. In *International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 8–22. Springer, 1999.

[BBG+05] S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W. J. Paul. Towards the Formal Verification of Lower System Layers in Automotive Systems. In *International Conference on Computer Design (ICCD '05)*, pages 317–326. IEEE Computer Society, 2005.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, number 3185 in LNCS, pages 200–236. Springer, 2004.

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

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

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

[BP06] G. M. Brown and L. Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006), held as part of the Joint European Conferences on Theory and Practice of Software, (ETAPS 2006)*, volume 3920 of *LNCS*, pages 58–72. Springer, 2006.

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

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[Dil90] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer, 1990.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop*, volume 1066 of *LNCS*, pages 208–219. Springer, 1995.

[EFGP10] R. Ehlers, D. Fass, M. Gerke, and H.-J. Peter. Fully Symbolic Timed Model Checking using Constraint Matrix Diagrams. In *31st IEEE Real-Time Systems Symposium (RTSS 2010), to appear*, 2010.

[EGP10] R. Ehlers, M. Gerke, and H.-J. Peter. Making the Right Cut in Model Checking Data-Intensive Timed Systems. In *12th International Conference on Formal Engineering Methods (ICFEM 2010)*, volume 6447 of *LNCS*, pages 565–580. Springer, 2010.

[Fle05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1 Revision A*, 2005.

[GEFP10] M. Gerke, R. Ehlers, B. Finkbeiner, and H.-J. Peter. Model Checking the FlexRay Physical Layer Protocol. In *15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2010)*, volume 6371 of *LNCS*, pages 132–147. Springer, 2010.

[Ger07] M. Gerke. *Implementation of Frame and Symbol Transmission in a Time Triggered Serial Bus Architecture*. Bachelorthesis, Universität des Saarlandes, 2007.

[Jon06] A. M. Jones. *Asynchronous communication among hardware object nodes in IC with receive and send ports protocol registers using temporary register bypass select for validity information*. US Patent Number 7409533, 2006.

[KP95] J. Keller and W. J. Paul. *Hardware Design: Formaler Entwurf digitaler Schaltungen*, volume 15 of *Teubner-Texte zur Informatik*. B. G. Teubner Verlagsgesellschaft mbH, 1995.

[KP07] S. Knapp and W. Paul. Realistic Worst-Case Execution Time Analysis in the Context of Pervasive System Verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 53–81. Springer, 2007.

[LCL88] F. J. Lin, P. M. Chu, and M. T. Liu. Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies. In *ACM Workshop on Frontiers in Computer Communications Technology (SIGCOMM '87)*, pages 126–135. ACM, 1988.

[LLPY97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structure and State–Space Reduction. In *18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 14–24. IEEE Computer Society, 1997.

[Män98] R. Männer. Metastable States in Asynchronous Digital Systems: Avoidable or Unavoidable? *Microelectronics Reliability*, 28(2):295–307, 1998.

[Sch06] J. Schmaltz. A Formal Model of Lower System Layers. In *6th International Conference on 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 *7th International Conference on Formal Methods in Computer-Aided Design (FMCAD '07)*, pages 223–230. IEEE Computer Society, 2007.

[Som09] F. Somenzi. CUDD: CU Decision Diagram package release 2.4.2, 2009.

[Vd06] F. W. Vaandrager and A. L. de Groot. Analysis of a biphase mark protocol with UPPAAL and PVS. *Formal Aspects of Computing*, 18(4):433–458, 2006.