Complete Formal Hardware Verification of Interfaces for a FlexRay-like Bus

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Christian Müller

Saarbrücken, November 2011

Tag des Kolloquiums:	7. November 2011
Dekan:	Prof. Dr. Holger Hermanns
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Dr. habil. Silvia Melitta Müller IBM Systems & Technology Group IBM Deutschland
Akademischer Mitarbeiter:	Dr. Gordon Fraser

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, November 2011

“Time turns the improbable into the inevitable.”

Author unknown

Acknowledgments

I owe my gratitude to Professor Paul for being a great teacher.
I thank Ulan for all the adventures we have gone through.
I thank Dr. Silvia Melitta Müller who kindly agreed to review this thesis.

Abstract

We report in this thesis the first complete formal verification of a bus interface at the gate and register level. The presented bus interface allows to implement a time-triggered system consisting of several units interconnected by a bus. Time-triggered systems work decentralized, allow some grade of fault-tolerance against a bounded number of single errors and show a predictable recurrent behaviour.

We use a hardware model for multiple clock domains obtained by formalization of data sheets for hardware components, and we review known results and proof techniques about the essential components of such bus interfaces: among others serial interfaces, clock synchronization and bus control. Combining such results into a single proof leads to an amazingly subtle theory about the realization of direct connections between units (as assumed in existing correctness proofs for components of interfaces) by properly controlled time-triggered buses. It also requires an induction arguing simultaneously about bit transmission across clock domains, clock synchronization and bus control.[1]

The design of the bus controller can be automatically translated into Verilog and deployed on FPGAs.

Zusammenfassung

In dieser Arbeit präsentieren wir die erste formale Verifikation einer Bus-Schnittstelle auf der Register- und Gatter-Ebene. Die Bus-Schnittstelle ermöglicht die Implementierung eines zeitgesteuerten Systems, welches aus mehreren Einheiten besteht, die durch einen Bus verbunden sind. Systeme dieser Art funktionieren dezentralisiert, sind fehlertolerant gegen einzelne System- und Umgebungsfehler und weisen ein berechenbares periodisches Verhalten auf.

Wir benutzten ein Hardware-Model für mehrere Clock-Domänen, welches durch die Formalisierung der Herstellungsinformationen abgeleitet wurde. Wir präsentieren verschiedene Ergebnisse und Verifikationstechniken über die essentiellen Komponenten solcher Bus-Schnittstellen: serielle Schnittstellen, Clock-Synchronisierung, Bus-Kontrolle, usw.

Die Kombination solcher Ergebnisse zu einem einzigen Korrektheitsbeweis führt zu einer nicht-triviallen Theorie über die Realisierung einer direkten Verbindung zwischen verschiedenen Einheiten des Systems (wie das in den einzelnen Beweisen verschiedener Komponente angenommen wird), die auf einer korrekten Kontrolle zeitgesteuerter Busse basiert. Die Korrektheit der gesamten Schnittstelle ergibt sich aus einem Induktionsbeweis, der gleichzeitig über drei Eigenschaften argumentiert: über die Signalübertragung zwischen unterschiedlichen Clock-Domänen, über die Clock-Synchronisierung und über die zeitlich-korrekte Einteilung der Bus-Zugriffe.

Die Implementierung kann automatisch in Verilog-Code übersetzt werden und auf FPGA-Boards ausgeführt werden.

[1] Note that this abstract partially coincides with the abstract of [PM11].

CONTENTS

LIST OF FIGURES

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

Usually, the motivation section of work in the area of verification talks about exploding rockets, expensive bugs in processors, complex technologies and many other things, which have mostly implicit impact on our everyday life. We would like to talk about technologies, which we deal almost every single day with – and not *implicitly* but *explicitly*: the modern automobiles.

According to the Institute of the Motor Industry:

> *Electronics account for around 20% of the value of today's average light vehicle, with one supplier, Siemens, stating that 90% of innovations in a modern car can be attributed to electronics development. But here's the rub: DaimlerChrysler blames electronic faults for 70% of its well-publicised quality problems at Mercedes-Benz.*[1]

While quality problems do not affect safety, the real problems may arise when the automobile electronics begin to assist the driver in safety-critical functions.

In 2009, Volvo introduced a crossover XC60 with the industry's first auto-breaking feature which serves as an automatic collision detector. The system scans the environment of the moving car using built-in laser sensors to detect a potential collision with a pedestrian or another car, and – in case of a detected obstacle – if the system does not register any driver's reaction to avoid the collision, the car will jab the breaks and stop. While some luxury sedans like the BMW 7 Series and Mercedes Benz S-Class have used similar collision detection systems, their functionality was limited to warnings of the driver only. In Volvo's XC60, the system actively interferes with the car control in case of an emergency.

While such a feature is undoubtedly conductive to safety of the motor traffic, it is important to understand that it is not allowed to produce any false-positive results. If a car full of passengers would be automatically stopped by its error-prone electronics on a highway with high density traffic, such a feature may call for more lives than it will possibly save. The same problems concern any sensor-based safety-critical features, like airbags or stability control.

[1] http://www.motor.org.uk/magazine/articles/cover-feature-bugs-in-the-system-101.html

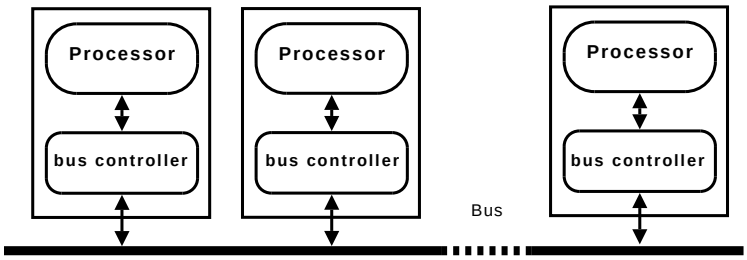


Figure 1.1: Time-Triggered Bus System

Through testing alone, it is infeasible to prove the desired level of reliability [Pik07, BF93]. But on the other hand, verification of an industrial-scale bus system is possibly infeasible in a reasonable amount of time [Pik06]. Previous research contains numerous examples of case studies about verification of isolated parts, protocols and algorithms of time-triggered systems. However, the question whether all these verification efforts carried out with different tools and on different abstraction levels can be combined into one single correctness statement remained open so far.

1.2 The Problem

A round-based time-triggered system consists of several units interconnected by a bus as depicted in Figure 1.1. These units function in the following manner. The global time line of the system is split into equal time segments called *rounds*, consisting of several slots. In each slot exactly one unit broadcasts one message over the bus. All other units are listening to the bus in that slot.

Every unit is clocked by its own oscillator, i.e., units don't share one common discrete time notion. The correctness of message transmission in a time-triggered system can roughly be split into 5 sub-problems as depicted in Figure 1.2. Note that arrows in that figure denote dependencies of sub-problems. Hence, there are five problems to overcome.

1. **Correctness of Serial Interfaces.** In order to establish a message transfer over the bus from one unit to another, we need a formalism for an asynchronous signal exchange between two directly connected units: a sender and a receiver. Additionally, the transfer of complex messages requires a low-level protocol synchronization due to clock drift of the sender and the receiver.

2. **Clock (Timer) Synchronization.** As in the first problem, the speed differences of units make a synchronization of the common time notion among all participants of the communication necessary. Any software clock synchronization algorithm relies on the message exchange. In the presented time-triggered system, this is rather a simple mechanism and will be realized with the help of *sync messages* (prefixes of ordinary messages indicating the start of a new round). However, the sync messages will be transferred over the bus, which has to be collision-free at the time of the transmission.

3. **Bus Contention Control.** The synchronized time notion of all units provides a coordinated behaviour among them, s.t. at each time at most one unit acts as a

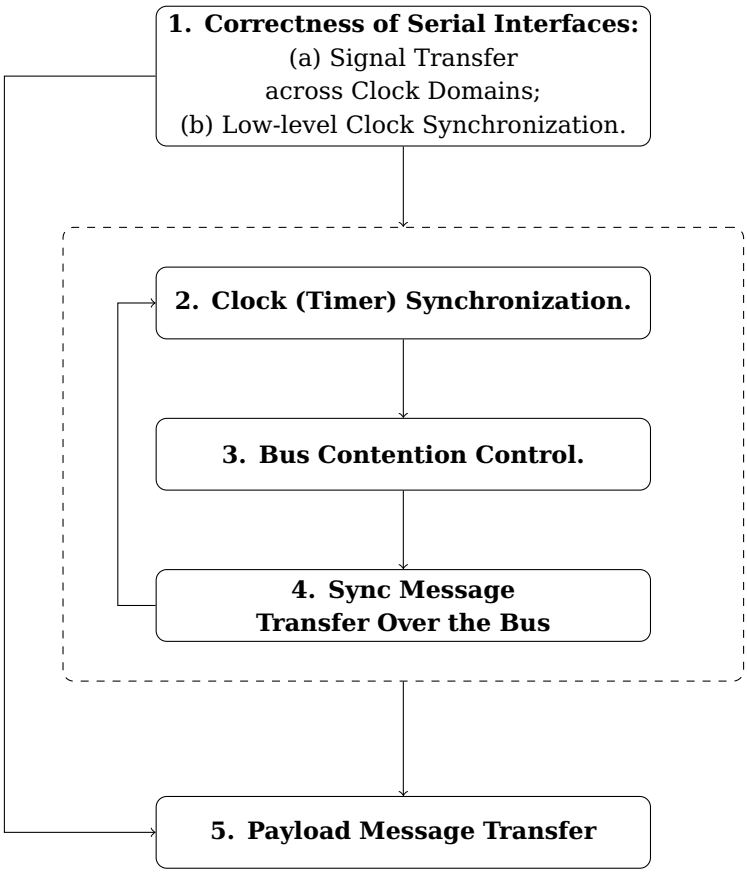


Figure 1.2: Decomposition of the Message Exchange Correctness

sender. Hence, it can be shown that no bus contention (a conflict between two senders) can appear during the system run.

4. **Sync Message Transfer Over the Bus.** Since the bus can be proven to be collision-free during all transmission times, we can show, that the sync messages can be transmitted correctly.

5. **Payload Message Transfer.** The bus contention control provides a transmission window where the bus can be abstracted to a direct connection between the sender and all receivers; thus, applying the correctness of serial interfaces, the correctness of the payload message transfer can be shown.

Note that Problems 2, 3 and 4 have a cyclic dependency. That means, clock synchronization hinges on message transfer (at least for the synchronization messages), message transfer on bus contention control and bus contention control on clock synchronization. Any theorem stating in isolated form the correctness of clock synchronization, bus contention control or message transfer alone must use hypotheses which break this cycle in one way or the other. If the theorem is to be used as part of an overall correctness proof, then one must be able to discharge these hypotheses in the induction step of a proof arguing simultaneously about clock synchronization, bus contention and message transfer. A paper and pencil proof of this nature can be found in [KP07].

Besides the enumerated problems, another important point is fault-tolerance. Time-triggered systems are usually required to operate in an environment with possibly faulty components and flipped bits during transmission. These faults can be caused by the physical environment or by the error-prone software or hardware. Hence, almost every of the presented problems should tolerate a number of faults bounded by the *fault hypothesis* (assumptions about the frequency, type, location of faults, etc.). We do not deal with fault-tolerance in this thesis, however, we recognize that this problem has to be tackled as soon as the message exchange can be proven in an error-free environment.

Despite the fact that probably every of these five problems were solved by previous research in isolation, to the best of our knowledge, there is not a single previous successful combination of the presented challenges into one uniform correctness statement.

We review the related work in the next section in context of the presented problem decomposition and describe our contribution afterwards.

1.3 Related Work

Verification of real-time systems is by no means new. First efforts were done in the late 80's, tackling the clock synchronization algorithms for fault tolerant systems [RvH89, RvH93, SS92, PSHI99]. These results deal with algorithmical correctness of the clock synchronization only and cover the Problem 2 from Section 1.2. Furthermore, over the last years, several architectures suitable for safety-critical real-time systems were developed. Rushby [Rus01a] gives an overview and a comparison of four of them: SAFEbus, SPIDER, FlexRay and TTA. He also describes [Rus99] a general approach for deriving time-triggered implementations from algorithms specified as functional programs under the assumption of correct message exchange across clock domains (Problem 3, 4) and clock synchronization (Problem 2). He specifies timing constraints,

which a schedule of a time-triggered protocol has to fulfill, to form in its system run so-called *cuts*. These cuts are time segments, where the state of the time-triggered system can be related to a state of its synchronous counterpart. The overall correctness of these implementations would then follow from the correctness of concrete functional programs (which is supposed to be relatively easy), and from the correct transformation of synchronous systems into timed asynchronous implementations.

In [Pik06], Pike has presented several results. One part of his work related to this paper was a corrected and significantly extended version of Rushby's formalism [Rus99]. He also applies this approach to verify the schedule timings of two protocols of the SPIDER bus architecture: Clock Synchronization and Distributed Diagnosis.[2] However, it remains unclear how exactly the timing properties of hardware implementations of these protocols were derived and mapped to the formal model. Moreover, the proposed technique assumes even in the proof of timing properties of a Clock Synchronization protocol already synchronized clocks initially. That is, to extend the proof to correctness for all rounds one needs to use the proposed proof as an induction step. However, to show the initial clock synchronization the correctness of the initialization might become necessary.

Other previous verification efforts – to the best of our knowledge – tackle isolated parts or particular algorithms and protocols of different time-triggered real-time systems. For example, Rushby gives an overview [Rus02] of verified algorithms of TTA. The described verification efforts concern some isolated parts, properties or algorithms of TTA such as clock synchronization, transmission window timing, group membership, etc. Rushby says in this overview: "Some of these algorithms pose formidable challenges to current techniques and have been formally verified only in simplified form or under restricted fault assumptions." For example, the window transmission timing (Problem 3) was verified by Rushby himself [Rus01b] for a transmitter, its bus guardian and a receiver. As in [Rus99] this work assumes the correctness of clock synchronization and correct scheduling mechanism (Problem 1, 2 and 4).

Steiner et al. [SRSP04] verify a startup algorithm for TTA against all possible fail scenarios using a model checker (this might serve as solving of Problem 2 for the initial round). Pike and Brown present in [BP06] a largely automated verification of the Biphase Mark and 8N1 protocols, which covers the Problem 1. They derive a generic model for two asynchronously communicating units, which includes modeling of clock jitter. Their clock model is based on so-called *timeout automata*. The progress of global time is enforced cooperatively by sender and receiver clocks. This model deals with clock jitter but does not model the set up and hold times explicitly. The clock modeling is partially protocol-dependent. Since the sender's clock progress depends on the receiver's clock progress, it is not fully clear how to extend this model to several receiver clocks. Unfortunately, no discussion was provided about the gap between the given stack of abstractions and modeling of actual hardware. Pike and Johnson report in [PJ05] a verification of the Reintegration Protocol of the SPIDER architecture, where this approach was developed.

In [BP07], Brown and Pike show an approach, derived from the classic Abadi-Lamport refinement method, how to use a model checker and an SMT solver to prove temporal refinement of the 8N1 protocol (Problem 1).

In [Pfe03], Pfeifer formally verifies two fault-tolerant algorithms implemented in the Time-Triggered Protocol TTP/C: Group Membership and Clock Synchronization (Prob-

[2] Note that no correctness of the synchronous version of these protocols was provided.

lem 2). Both algorithms were analyzed, using a hand-derived mathematical specification of the TTP/C protocol. Group Membership is an algorithm executed on every processor of the bus architecture, which is used to analyze and determine faulty processors to exclude them from the communication. Pfeifer proves the correctness of Group Membership for an untimed system, where all processors work in lock-step sharing one common discrete time notion. Hence, his proof assumes the correctness of underlying asynchronous hardware (Problems 1-5) and could be implemented on the bus interface presented in this thesis.

His Clock Synchronization algorithm uses the arrival times of ordinary messages. However, only the messages comming from processors with accurate clocks will be considered. By his fault hypothesis, in every round, at least the minimal amount of required messages will be transmitted. He does not deal with bus control and message transmission assuming a working message exchange mechanism. Hence, his proof is based on the solved Problem 1 and on an abstraction of the bus to an one to one connection between all processors, or on the solved Problem 4 for a bus architecture.

Both algorithms are mutually dependend for the following reasons.

1. Since the messages whose arrival times are used for computation of the clock adjustment should come from non-faulty processors, the correctness of the Group Membership algorithm has to be assumed.

2. However, the Group Membership algorithm was proven as an untimed algorithm, which is only possible if Clock Synchronization can be proven to be correct, such that an untimed algorithm can be refined to a timed one.

Although the algorithms are mutually dependend, Pfeifer has analyzed them in isolated form and on different levels of abstraction: the Group Membership was analyzed as an untimed (synchronous) mechanism, whereas the Clock Synchronization was verified in a timed model. He also introduces an abstract induction-based principle how to combine both proofs resolving their mutual dependency.

If we want to integrate both algorithms into the time-triggered system presented in this thesis, we need to extend the circular dependency by a new **Problem 6:** the Group Membership. This 6th problem would depend on Problems 1-5, and the Problem 2 becomes dependend on Problem 6.

Some interesting results were recently achieved with the Uppaal model checker. Gerke et al. [GEFP10] presented a fully automatic proof of the message transmission on the physical layer protocol of FlexRay describing the underlying model as a network of *timed automata*. They model the asynchronously communicating part of the FlexRay hardware as proposed in [Sch07], which was developed in Verisoft project for the automotive system presented in this thesis. The proof was carried out under the assumption of working TDMA schedule (this implies clock synchronization), which is maintained by high-level protocols. Hence, they have solved Problem 1 fully automatically.

Furthermore, assuming clock synchronization Zhang [Zha06] has proven two properties of the scheduling protocol of a FlexRay bus guardian: the correct relay and message integrity.

Almost all of these efforts deal with conceptual correctness of algorithms without linking them to concrete asynchronously communicating hardware models. Moreover, it would be highly desirable to reuse some of the developed techniques or proven algorithms and to consolidate them into one single theory. However, it is not clear, how to combine these results taking into account the fact, that these theories were made

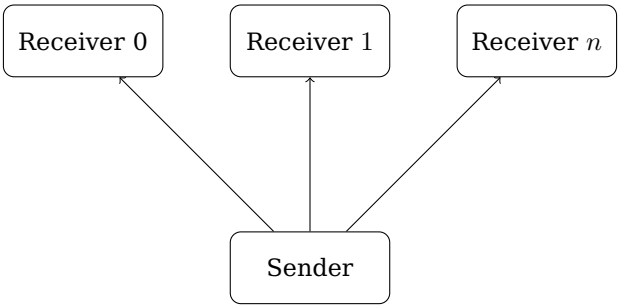


Figure 1.3: Direct Connection of Serial Interfaces

using different formalisms, different verification environments and are formulated at different levels of abstractions.

1.4 Contribution to the "Automotive" Project

The work presented in this thesis was started as subproject "Automotive" of the Verisoft project. Verisoft is a long-term research project funded by the German Federal Ministry of Education and Research. One of the goals of Verisoft is the formal pervasive verification of computer systems including all levels from gate-level hardware to user applications.

In contrast to related work, Verisoft has tackled the problem of pervasive verification of a distributed time-triggered system from another side. Instead of verifying an industrial-scale bus system, a simple but flexible and extendible architecture influenced by the FlexRay standard [Con06] with simple synchronization and communication protocols was developed at the gate and register level. The system consists of bus interfaces implemented on gate and register level, interconnected by a bus. The modeling of the asynchronous communication relies on a precise timing model of hardware registers obtained by formalization of data sheets for hardware components.

The theoretical foundations of this work were formulated in computer science lectures "Computer Architecture 2" [Pau05] by Wolfgang Paul and published in [KP07]. Previous verification efforts were reported in [Sch07] and [Böh07].

In [Sch07], Schmaltz has proven the correctness of serial interfaces (Problem 1) of the presented bus controller and the payload message transfer (Problem 5) under the assumption of two directly connected and appropriatelly initialized units. That means, Schmaltz's results can be used as soon as Problems 2, 3 and 4 are solved.

In [Böh07], Böhm has used the correctness of serial interfaces provided by Schmaltz and has shown the transmission of sync messages (Problem 4) and resulting Clock Synchronization (Problem 2) for one sender and arbitrary receiver under the assumption that every receiver is directly connected to the sender as depicted in Figure 1.3. Thus, he could ignore the Problem 3, because no notion of the bus was introduced.

Hence, to couple these two results into one theory, filling of the last gap, namely, solving of the Problem 3 was necessary. The extension and consolidation of previous results into a single correctness theorem presented in this thesis was reported in [PM11], hence, some passages of this publication coincide with this thesis literally.

In this case study, we have verified the message transmission over the bus relying on the minimal set of hypotheses like bounded clock drift. The design of the bus controller

can be automatically translated into Verilog and deployed on FPGAs. Our contribution is the first complete formal verification of a FlexRay-like bus interface at the gate and register level.

1.5 Tools

The gate-level implementation of the bus controller which will be presented in this thesis was carried out [Kna08] in the specification language of the interactive theorem prover Isabelle/HOL [NPW02]. The Isabelle/HOL specification language is an extension of functional programming language ML.

The verification of the controller was carried out in Isabelle/HOL supported by the symbolic model checker NuSMV [CCG+02], which was integrated [Tve05] into Isabelle/HOL. NuSMV is an open source re-implementation of the model checker CMU SMV [CCG+98] for CTL and LTL properties. We have used NuSMV to verify temporal properties of the system defined over Kripke structures [Gup93] and as an external BDD decision procedure. To verify a property by NuSMV, we have formulated this property as an LTL formula and applied IHaVeIt [Tve05] to it.

IHaVeIt is a translation tool, which translates Isabelle's specification language into NuSMV input language. Moreover, it consists of several algorithms for handling uninterpreted functions and data abstractions used in Isabelle models.

After the Isabelle code is translated from Isabelle to NuSMV language, the NuSMV model checker can be applied to it. We will present all lemmas in higher-order logic notation.

Another benefit of the IHaVeIt tool is the possibility to translate a gate-level hardware implemented in Isabelle/HOL specification language directly to Verilog fully automatically. This was used [End09] to translate the bus controller implementation into Verilog and to deploy it on three interconnected FPGAs.

1.6 Outline

In Chapter 2 we introduce the notation and concepts we use during modeling and verification; we introduce boolean gates and circuits representing basic blocks in the hardware design of the studied bus controller.

In Chapter 3 we formalize the communication scheme used in our FlexRay-like time-triggered system. The communication scheme is based on the Time Division Multiple Access strategy and hinges on recurrent clock synchronization of all units participating in the communication.

In Chapter 4 we describe how a clock domain crossing communication is modeled [Sch07] by developing a register model with precise timing and linking it to a digital hardware model. Furthermore, we extend this model to a communication bus interconnecting all units of the time-triggered system. We also present and extend the results of previous efforts in the verification of the low-level signal tranmission among different clock domains.

In Chapter 5 we formulate the correctness criteria for message transmission in our time-triggered system and decompose it into three milestones.

In Chapter 6 we show the implementation of relevant parts of the bus controller and introduce the used message protocol.

In Chapter 7 we present previous verification [Böh07] of the schedule timing and clock synchronization within a single round under the assumption of a direct connection between the sender and all receivers. We use these results and present the verification of bus contention control property, allowing us to abstract the bus connection between the sender and every receiver as a direct wire. This allows us to apply results from Chapter 4 to show the correctness of high-level message transmission [Sch07].

Finally, in Chapter 8 we use the theorem about the contention control and apply the hardware correctness of serial interfaces to show the high-level transmission correctness.

In Chapter 9 we conclude our work, discuss future work and give some statistics.

CHAPTER

TWO

BASICS

2.1 Notation

In this thesis we use following notation.

Operators

- the operator := denotes an assignment, e.g., in $a := b$ we assign value b to identifier a
- the operator $\equiv$ denotes equivalence of expressions

Sets

- $\mathbb{B} := \{0, 1\}$
- $\mathbb{B}^*$ is a set of all tupels with elements from $\mathbb{B}$
- $\mathbb{R}$ is the set of real numbers
- $\mathbb{Z}$ is the set of integers
- $\mathbb{N}$ is the set of natural numbers without 0
- $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$

Intervals For $a, b, c \in \mathbb{R}$ we define:

- $[a : b] := \{x \mid a \leq x \leq b\}$
- $[a : b) := \{x \mid a \leq x < b\}$
- $(a : b] := \{x \mid a < x \leq b\}$
- $(a : b) := \{x \mid a < x < b\}$

Bits and Bitvectors

- if $b \in \mathbb{B}$ then we call b a *bit*
- if a and b are bits then we denote by $a \circ b$ a concatenation of bits a and b
- a concatenation of bits is also called a *bitvector*
- if a and b are bitvectors then we denote by $a \circ b$ a concatenation of bitvectors a and b, which is a bitvector again
- if b is a bitvector, then $b[i]$ is its i'th bit, counting right to left (little endian)
- if b is a bitvector of length n, then we also refer to it by $b[n-1:0]$
- a bitvector of length 1 is a bit
- if $a \in \mathbb{B}, n \in \mathbb{N}$ then we denote by a^n a bitvector $\underbrace{a \circ a \circ ... \circ a}_{n \text{ times}}$

Functions

- Let b be a bitvector of length n, then we encode b as a natural number by function $\langle \cdot \rangle$ as follows:
$$\langle b \rangle := \sum_{i=0}^{n-1} b[i] \cdot 2^i$$
- if $i < 2^n$ for $n \in \mathbb{N}$ then $bin_n(i)$ returns the bitvector of length n, which represents number i (note: $i = \langle bin_n(i) \rangle$)
- For $x \in \mathbb{R}$ we define the flooring and ceiling operators:
  - $\lfloor x \rfloor = max\{a \in \mathbb{Z} \mid a \leq x\}$
  - $\lceil x \rceil = min\{a \in \mathbb{Z} \mid x \leq a\}$

2.2 Trace Semantics

Note that our hardware model of the entire time-triggered system goes beyound the ordinary digital hardware model. Since we want to argue about a set of asynchronously working units within different clock domains, we need to extend the usual hardware model by more precise timing parameters such as setup and hold times. We also must deal with undefined and metastable signals. We will introduce this extension in Chapter 4.

The hardware presented within a clock domain consists of finite data structures only and has a deterministic behaviour. To model hardware we split it into hardware states (configurations) and a transition function over them. The state of a hardware can be characterized by the contents of its registers and memories. These registers change their content from cycle to cycle according to the logical circuits, placed at their inputs. We can represent a digital hardware model and its computations as a finite Mealy automaton [MP00]:

$$(S, S_0, \Sigma, \Lambda, \delta, \eta)$$

It consists of the following:

- a finite set of hardware states S;
- a set of initial states $S_0 \subset S$;
- a finite set of all possible register inputs Σ;
- a finite set of all possible register outputs Λ;
- a transition function $\delta : S \times \Sigma \to S$;
- an output function $\eta : S \times \Sigma \to \Lambda$.

The transition function takes a hardware state, register inputs and produces a new configuration of the bus controller. The output function takes a hardware state, register inputs and outputs register contents of the next hardware state.

Moreover, we fix an abstract trace function $trace : \mathbb{N} \to S$, which assigns a hardware state to each natural number. We also fix a similar function for register inputs: $inputs : \mathbb{N} \to \Sigma$.

We call the function $trace$ a *valid execution trace* if it produces an initial configuration for cycle 0 and each consecutive configuration is computed by applying the transition function once:

$$trace(0) \in S_0 \wedge \forall i : trace(i+1) = \delta(trace(i), inputs(i))$$

Such a valid execution trace represents a computation of the real hardware. Each hardware state of the model is represented in a real hardware by contents of its registers and RAMs. The transition function is then realized by logical gates and combinational circuits.

In this thesis we will refer to a hardware state of some model h in cycle i by h^i, which denotes the configuration returned by the trace function for number i:

$$h^i := trace(i)$$

We will always assume that used trace functions are valid execution traces.

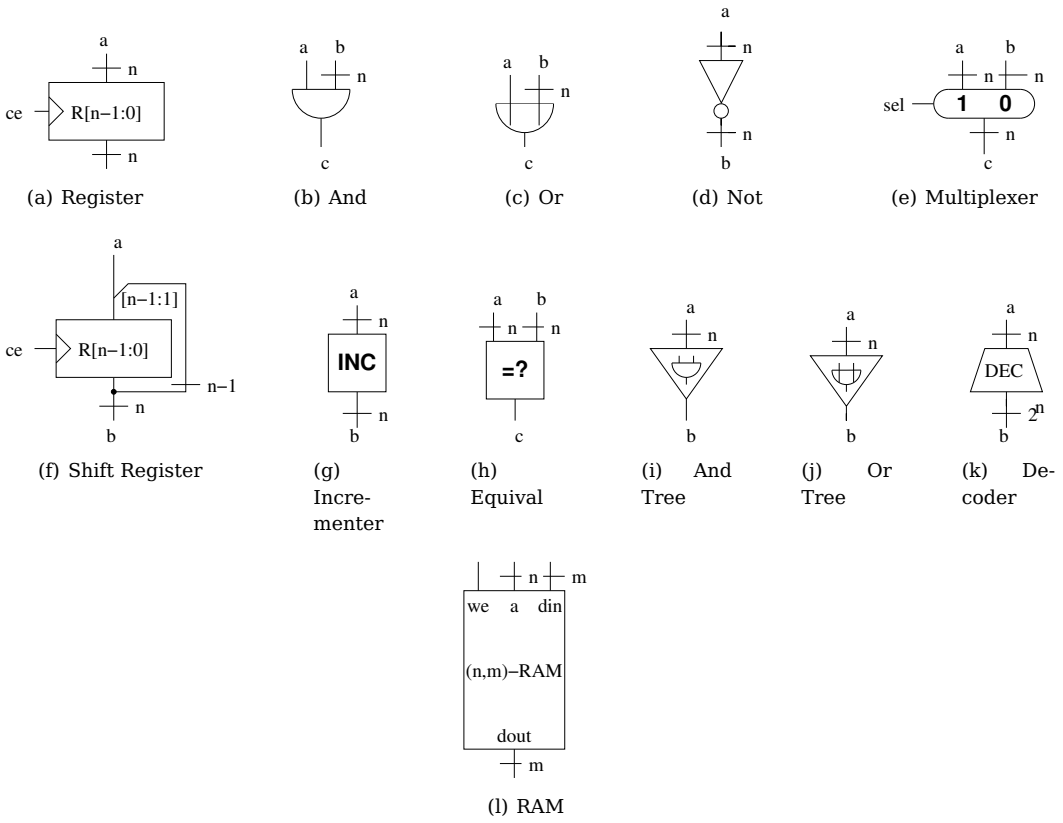
Moreover, we denote by $h^i.R$ the content of register R of hardware state h in cycle i, and by $f(h^i)$ a boolean signal derived from other register contents and boolean gates of hardware state h in cycle i. Sometimes, we will abbreviate $h^i.R$ by R^i and $f(h^i)$ by f^i. The same notation and abbreviatation is also used for a memory content $h^i.M$.

2.3 Boolean Gates and Basic Circuits

In the description of the hardware design we use the following boolean gates and curcuits.

- **Register** is depicted in Figure 2.1(a); it stores a bitvector of length n (or one bit, in case $n = 1$) and has the following semantics. Let R^i be the content of register R in cycle i and let a^i be the input and ce^i be the clock enable signal of register R in cycle i.

$$R^{i+1} = \begin{cases} a^i & : \quad ce^i = 1 \\ R^i & : \quad \text{otherwise} \end{cases}$$



(a) Register (b) And (c) Or (d) Not (e) Multiplexer

(f) Shift Register (g) Incrementer (h) Equival (i) And Tree (j) Or Tree (k) Decoder

(l) RAM

Figure 2.1: Boolean Gates and Circuits

- **AND** Gate depicted in Figure 2.1(b) has the following semantics. Let $a \in \mathbb{B}$ and $b \in \mathbb{B}^n$ with $n \in \mathbb{N}$. Then its output $c \in \mathbb{B}^n$ is specified as:

$$\forall i \in [0 : n-1] : c[i] := a \wedge b[i]$$

  Hence, it computes logical conjunction of a bit and a bitvector.

- **OR** Gate depicted in Figure 2.1(c) computes disjunction of a bit and a bitvector and is specified analogously to the AND Gate in a straightforward way.

- **NOT** Gate depicted in Figure 2.1(d) flips every bit of the input bitvector $a \in \mathbb{B}^n$ with $n \in \mathbb{N}$. Its output $b \in \mathbb{B}^n$ is specified as:

$$\forall i \in [0 : n-1] : b[i] := \neg a[i]$$

- **Multiplexer** Gate depicted in Figure 2.1(e), its output $c \in \mathbb{B}$ is specified as:

$$\forall n \in \mathbb{N}, a, b \in \mathbb{B}^n, sel \in \mathbb{B} : c := \textbf{if } sel \textbf{ then } a \textbf{ else } b$$

- **Shift Register** depicted in Figure 2.1(f) is a tuple of 1-bit registers

$$(R_{n-1}, ..., R_0)$$

with $n \in \mathbb{N}$, which are always clocked. The registers are connected, s.t. for all cycles $c \in \mathbb{N}$ and input $a \in \mathbb{B}$ holds:

$$R^c_{n-1} = a^{c-1} \wedge \forall i \in [0 : n-2] : R^{c+1}_i = R^c_{i+1}$$

Its output $c \in \mathbb{B}^n$ is specified as follows:

$$\forall i \in [0 : n-1] : c[i] = R_i$$

- **Incrementer** circuit is depicted in Figure 2.1(g), it increments a bitvector b of length n by one. The overflow bit will be skiped, thus, incrementing of bitvector 1^n yields 0^n.
- **Equivalence** circuit depicted in Figure 2.1(h) compares two bitvectors. Its output $c \in \mathbb{B}$ is computed as:
$$\forall n \in \mathbb{N}, a, b \in \mathbb{B}^n : c := (a = b)$$
- **And Tree** circuit depicted in Figure 2.1(i) computes the conjunction of all bits of the input bitvector $a \in \mathbb{B}^n$ with $n \in \mathbb{N}$. The output $b \in \mathbb{B}$ is specified as:
$$b = \bigwedge_{i \in [0:n-1]} a[i]$$
- **Or Tree** circuit depicted in Figure 2.1(j) is a counterpart of the And Tree for the logical OR operation.
- **Decoder** circuit depicted in Figure 2.1(k) interprets the given bitvector as a number and outputs this number as a unary bitvector (see Terminology below). Let $a \in \mathbb{B}^n$ with $n \in \mathbb{N}$. Then, the output $b \in \mathbb{B}^{2^n}$ of the Decoder Circuit is specified as:
$$b = 0^{2^n - 1 - \langle a \rangle} 1 0^{\langle a \rangle}$$
- **(n,m)-RAM** circuit depicted in Figure 2.1(l) is used to store bitvectors of length m. Each bitvector will be addressed by a n-bit address bitvector. A RAM circuit has input ports we, a and din – for write enable signal, addrress and input data, respectively. It has following semantics. Let M be a **(n,m)-RAM**. Then:
$$M^{i+1}(addr) = \begin{cases} din^i & : \quad we^i = 1 \wedge a^i = addr \\ M^i(addr) & : \quad \text{otherwise} \end{cases}$$

2.4 Terminology

- We understand under a **clock domain** the scope of a system (a set of registers), where the time can be measured with the help of *one* clock. For example, if two circuits c_1 and c_2 are clocked by two different clocks, then we would refer to all registers of c_1 as being in the clock domain of circuit c_1; analogously, all register of c_2 would be in the clock domain of c_2.
- A value computed by logical gates and circuits within one clock domain is called a **signal**; we say it is **active**, if the computed signal has the value '1'; the signal is **inactive** otherwise.

- Later we will deal with instability and undefined behaviour of analog registors. An undefined register value will be denoted by Ω. Moreover, we will call a function s an **analog signal** if it is typed as follows:

  $$s : \mathbb{R} \to \{0, 1, \Omega\}$$

  The analog signal s is **active** at time t if $s(t) = 1$; it is **inactive** if $s(t) = 0$; it is **undefined** if $s(t) = \Omega$.

- We say a bitvector is **unary** if it has the form $0^i \circ 1 \circ 0^j$ for some $i \in \mathbb{N}$ and $j \in \mathbb{N}_0$.

CHAPTER

THREE

COMMUNICATION MODEL OF A SIMPLE FLEXRAY-LIKE TIME-TRIGGERED BUS SYSTEM

Clean formal definitions of time-triggered systems have been given in [Rus99] and [Pik06]. The term *time-triggered* means that the functionality of a system, e.g., communication, is based on actions scheduled at predefined repetitive points in time. Hence, the main control signal for all nodes of such system is the progress of the global time. Consequently, all nodes have to constantly synchronize their notion of time with at least one reference. Such a system works decentralized, allows some grade of fault-tolerance against a bounded number of single errors and shows a predictable recurrent behaviour.

The simple time-triggered system whose hardware realization is studied in this thesis is inspired by the FlexRay standard [Con06]. In this chapter we will develop a well-defined mathematical model of a TDMA-based communication scheme between asynchronously working units interconnected by a single communication bus.

This chapter is structured as follows. In Section 3.1 we give a top-level overview of the architecture of the studied bus network consisting of several Electronic Control Units (ECUs). We explain the concept of time from the point of view of a single ECU in Section 3.2.1 and explain the idea behind the used synchronization mechanism in Section 3.2.2. Then, we formalize the notion of clocks in Section 3.3 and show lemmas about their bounded drift. Finally, in Section 3.4 we formalize the notion of local time of all types of ECUs used in the studied bus network.

3.1 Top Level Overview

The studied bus architecture consists of $\mathcal{N}$ electronic control units (ECUs) connected to a single bus as depicted in Figure 3.1. The bus is used to broadcast signals in a serial manner from one ECU to all others. Broadcasting means that every ECU reads what was put on the bus. All ECUs are able to read and to change the bus value, but not simultaneously. That is, at any given time at most one ECU can broadcast to avoid the bus contention. Therefore, the communication between ECUs is split into disjoint

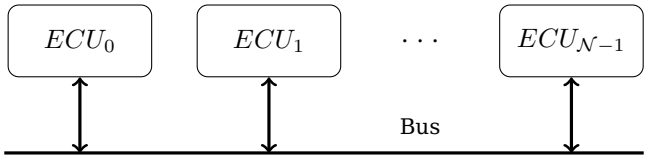


Figure 3.1: Network Topology of the Studied Bus Architecture

time periods according to the Time Division Multiple Access (TDMA) strategy. The TDMA strategy implies, that at each of these periods exactly one ECU has an exclusive write access to the bus, whereas all others are accessing the bus in read-only mode. To achieve such co-ordinated behavior of several ECUs, we have to ensure, that each ECU is aware of the current time interval in the communication and its role in this time interval.

Obviously, all ECUs should know 'what time it is', to agree on the same time points in the communication. The obstacle we have to overcome here, is the fact that all ECUs 'run' at different speed, since they are clocked by different oscillators. This may happen in physical designs because of temperature or voltage fluctuations, production tolerances of the timing source (an oscillator, for example), etc [Con06]. As a result, the clocks of all ECUs constantly drift apart which leads to shifting of local time views of all ECUs relatively to each other. Fortunately, the clock drift is bounded by some constant factor. To 'align' the view of the global time on different ECUs, a synchronization among all ECUs becomes necessary.

Thus, the communication in our bus network is time-triggered and is based on bounded clock drift and recurring synchronizations of all ECUs.

3.2 Communication Scheme

In this thesis, we call a continuous list of bits put on the bus by one ECU a *message*. Such an ECU is called a *sender*. All receiving ECUs will be referred to as *receivers*. We will refer to some concrete ECU in the bus network using its index as ECU_i for $i \in [0 : \mathcal{N} - 1]$. Note that our implementation assumes that every ECU broadcasts one message per round. The implementation and verification can be easily generalized to n messages per round for every ECU.

Our goal is to coordinate message transmissions in the bus network, such that each of $\mathcal{N}$ ECUs has a possibility to broadcast its message while all others are listening to the bus. All ECUs have to send their messages in turn. This consecutive message broadcast has to happen in *rounds*, i.e. once all $\mathcal{N}$ ECUs have finished their message transmission, another round starts and all ECUs have to send their messages in turn all over again. We need $\mathcal{N}$ disjoint *slots* in each round: one for each ECU. To let each ECU keep track of the current slot in each round, we will supply it with a time notion in Section 3.2.1.

However, as mentioned in Section 3.1, due to clock drift, the more time passes, the more the view of the global time on different ECUs differs. Thus, we need to synchronize all ECUs either between slots or between rounds. As the FlexRay standard [Con06] we choose the second option.

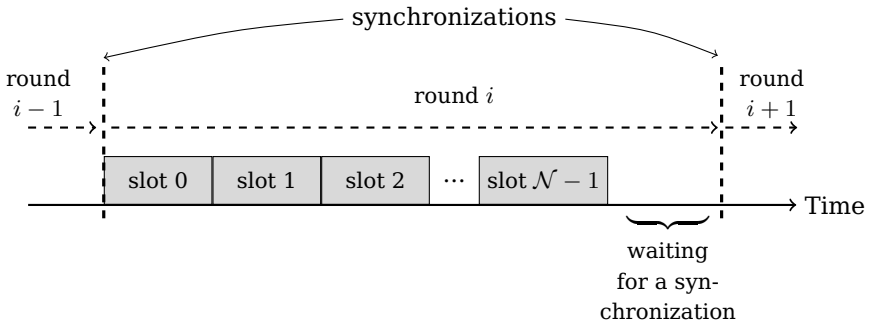


Figure 3.2: Local Time Notion of an ECU

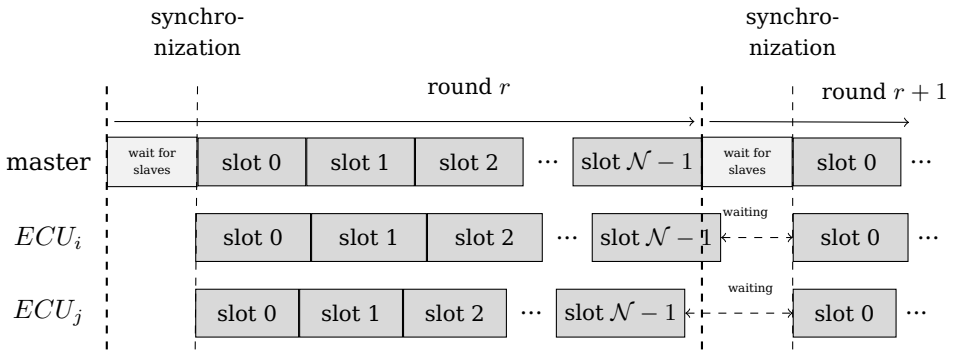


Figure 3.3: Local Time Notions of Different ECUs

3.2.1 Time Notion of an ECU

As depicted in Figure 3.2, on a top level, the local time of an ordinary ECU is split into rounds, where a round is a time segment between two synchronizations. Thus, all rounds are consecutive and disjoint. Moreover, a round contains $\mathcal{N}$ slots. These slots build a static task, which is executed by every ECU after every synchronization. Every ECU counts the number of cycles passed using local counters. The values of these counters determine the index of the current slot. In one of this slots, each ECU will act as a sender, and it will act as a receiver in all other slots. Each ECU knows its sending slot according to a fixed schedule function $send : [0 : \mathcal{N}-1] \to [0 : \mathcal{N}-1]$, which assigns to each slot number s an ECU index, s.t. $ECU_{send(s)}$ is a sender in slot s. After an ECU has executed the last slot in a round, it does not start a new round automatically. Instead, it waits for a synchronization message denoting the start of a new round.

3.2.2 Synchronization

The synchronization message will be produced by a special ECU called *master*. Our bus network has exactly one master ECU; we call all other ECUs *slaves*. Moreover, w.l.o.g. we assume, that the master ECU has the index 0, thus it will be denoted as ECU_0.

This synchronization principle is depicted in Figure 3.3 for two arbitrary ECUs. The master's behavior is very similar to the slave's behavior: within every round it keeps

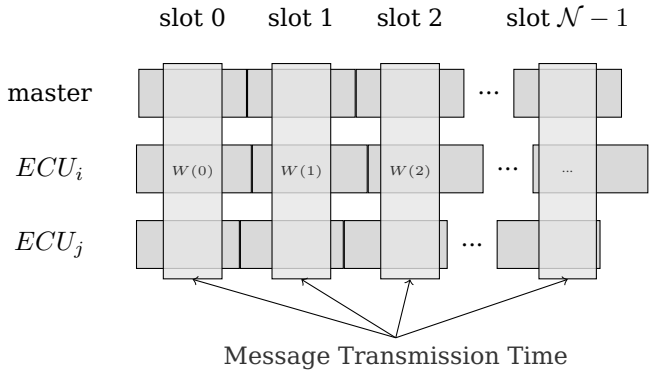


Figure 3.4: Overlapping Time Notions of Different ECUs

track of slots. However, in the beginning of each round, the master waits for some fixed amount of time, which is long enough to let all slaves finish the previous round. Afterwards, the master 'notifies' slaves about a new round by broadcasting a message, which simultaneously serves as a synchronization message for slaves. Thus, slot 0 of every round is always reserved for the master: $send(0) = 0$. The master always executes this static schedule. In contrast to this, the slave's behavior is time-dependent. They behave statically only within round boundaries keeping track of $\mathcal{N}$ slots after they have received a synchronization message. Afterwards, they wait until the next synchronization message to start the next round.

After a synchronization, the time views of different ECUs will continue to shift due to clock drift, however each local slot overlaps now with the same local slot on all other ECUs (see Figure 3.4). This overlapping time of each slot – the time segment where all ECUs agree on the current slot – will be used for a message transmission, and will be denoted as *transmission window* $W(s)$ of slot s. Obviously, this communication scheme will work only if the length of the master's waiting period, the length of each slot, and, hence, the length of a round, is chosen according to the maximal speed difference between two arbitrary ECUs. We will formalize all timing parameters in the next section.

3.3 Clocks and Clock Drift

As mentioned before, the local time notion of a single ECU consists of rounds which consist of slots. Moreover, the local time of an ECU is measured in hardware cycles. In the digital model of hardware circuits there are no states between two hardware cycles. Thus, the local time of an ECU is discrete.

In contrast, if we look at several ECUs simultaneously, we have to consider them outside of their clock domains. All ECUs run asynchronously to each other, and at every point in real time there is some state of the bus network, which depends on the state of every ECU. At this time point an ECU can be either within its local hardware cycle i or it can be in between cycles i and $i + 1$, updating its registers. Hence, we have to deal with continuous (real-valued) global time.

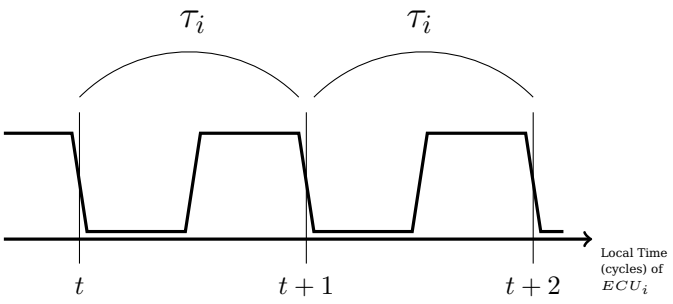


Figure 3.5: Clock Period of a Digital Clock

3.3.1 Formalization of Clocks

We have already mentioned, that all ECUs run at different speed, considering them relatively to each other. That is, one ECU accomplishes *the same amount* of local hardware cycles faster than an other ECU. Obviously, we cannot measure their speed difference in terms of local hardware cycles. Instead, we have to be able to compute 'how much time' it takes one ECU – in terms of global time – to accomplish n hardware cycles. This depends on the length of one hardware cycle of ECU_i known as *clock period* τ_i. As depicted in Figure 3.5, τ_i is the amount of real time, which ECU_i needs to accomplish one hardware cycle.

With the help of this formalization, we can measure the duration of n hardware cycles of two arbitrary ECUs: ECU_i and ECU_j. Obviously, these durations would be $\tau_i \cdot n$ and $\tau_j \cdot n$, respectively.

Besides this relative time measurement, we are also able to compute the absolute real time at which some local hardware cycle of a given ECU starts. For these computations we have to enumerate all hardware cycles starting from an origin in time, i.e. the real time of the start of the very first cycle. Later, we will have to compute local hardware cycles of several ECUs corresponding to a given real time. That is, our origin of the global real time should be valid for all ECUs. However, we cannot assume, that the very first cycles of all ECUs take place at the same time. Thus, we fix an imaginary origin as depicted in Figure 3.6, as being the minimal real time, when all ECUs are up and running. This would be the start time of the first cycle of the last started ECU.

Hence, we refer to the first cycle of each ECU starting at or after this origin as *cycle 0*. Since some ECUs might have started earlier, the fixed time origin will be within an already started cycle as it is the case for ECU_j in Figure 3.6. We will refer the time segment from the origin to the next cycle start as γ_i for a given ECU_i. Obviously, γ_i should be smaller than τ_i, otherwise the new defined cycle 0, would not be the minimal cycle taking place after the origin. Now we have all necessary information to define clocks formally.

Definition 1 (Clocks)**.** *A clock of an ECU_i is defined as a tuple of two real numbers $(\gamma_i, \tau_i) \in \mathbb{R}^2$ with $0 \leq \gamma_i < \tau_i$.*

Definition 2 (Edge Function)**.** *The starting time (edge) of the hardware cycle c of ECU_i is then defined as $e_i(c) = \gamma_i + c \cdot \tau_i$.*

Thus, $e_u(i)$ is the global time, at which the hardware cycle i was started in the clock domain u.

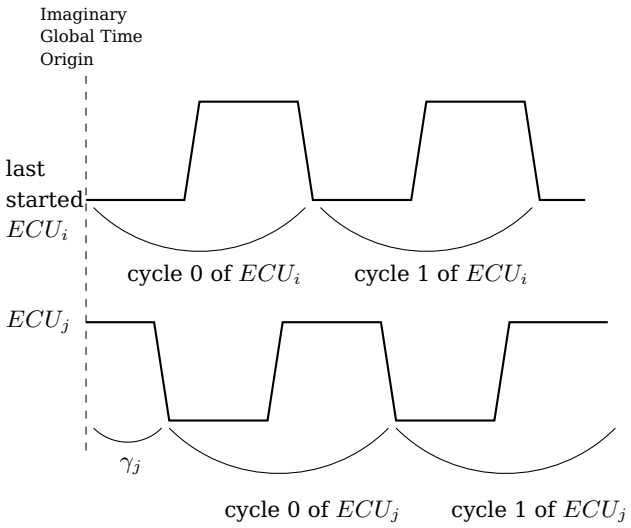


Figure 3.6: Definition of the Global Time Origin

3.3.2 Bounded Clock Drift

Now we can measure the speed of different ECUs in terms of the real time needed to accomplish the same amount of hardware cycles. This speed difference depends on the difference of the length of two clock periods. For further modeling we assume that all clock periods of every clock deviate at most by a percentage $\delta = 0.39\%$ of a reference clock period τ_{ref}. This constant is derived from the concrete parameters of our implementation (64 slots per round, 1 Kb per message). Note that this bound can be easily achieved by current technology. For example, FlexRay specification [Con06] prescribes *maximal* clock period deviation of size 0.15%.

Assumption 1 (Bounded Clock Drift)**.** $\forall i \in [0 : \mathcal{N} - 1] : 1 - \delta \leq \frac{\tau_i}{\tau_{ref}} \leq 1 + \delta$

We will implicitly use all introduced assumptions as premisses in all subsequent lemmas and theorems presented in this thesis. From Assumption 1 we can easily compute the maximal deviation of two arbitrary clocks.

Lemma 1 (Maximal Deviation of Two Clocks)**.** *Two arbitrary clock periods deviate at most by* $\Delta = \frac{2 \cdot \delta}{1 - \delta}$ *percent:*

$$\forall i, j \in [0 : \mathcal{N} - 1] : 1 - \Delta \leq \frac{\tau_i}{\tau_j} \leq 1 + \Delta$$

Proof. Let τ_i, τ_j be two clock periods with $i, j \in [0 : \mathcal{N} - 1]$. By Assumption 1 we know:

$$\forall \tau' \in \{\tau_i, \tau_j\} : 1 - \delta \leq \frac{\tau'}{\tau_{ref}} \leq 1 + \delta$$

thus:

$$\forall \tau' \in \{\tau_i, \tau_j\} : (1 - \delta) \cdot \tau_{ref} \leq \tau' \leq (1 + \delta) \cdot \tau_{ref}$$

Since we are interested in the maximal *deviation* of two arbitrary clock periods, we consider two cases, where the clock periods have the maximal difference lying on the boundaries of the allowed range $[(1 - \delta) \cdot \tau_{ref} : (1 + \delta) \cdot \tau_{ref}]$.

Upper bound computation.

$$\begin{aligned}
\frac{\tau_i}{\tau_j} &\leq \frac{(1+\delta)\cdot\tau_{ref}}{(1-\delta)\cdot\tau_{ref}} = \frac{1+\delta}{1-\delta} \\
&= \frac{1}{1-\delta} + \frac{\delta}{1-\delta} \\
&= \frac{1-\delta+\delta}{1-\delta} + \frac{\delta}{1-\delta} \\
&= \frac{1-\delta}{1-\delta} + \frac{\delta}{1-\delta} + \frac{\delta}{1-\delta} \\
&= 1 + \frac{2\cdot\delta}{1-\delta} = 1+\Delta
\end{aligned}$$

Lower bound computation. By an analogous computation as above we get:

$$\frac{\tau_i}{\tau_j} \geq \frac{(1-\delta)\cdot\tau_{ref}}{(1+\delta)\cdot\tau_{ref}} = 1 - \frac{2\cdot\delta}{1+\delta}$$

Hence, it remains to show:

$$1 - \frac{2\cdot\delta}{1+\delta} \geq 1-\Delta$$

The inequation follows by definition of Δ and $\delta > 0$. □

Now we show two additional helper lemmas.

Lemma 2 (Upper bound of Δ)**.**

$$\delta \leq 0.39\% \Rightarrow \Delta \leq 0.78\%$$

Proof. Trivial computation. □

Lemma 3 (Maximal Drift within 128 Cycles)**.** *Two arbitrary clocks clk_u and clk_v obeying Assumption 1 drift no more than by 1 cycle within 128 cycles.*

$$\forall i \leq 128 : \forall \tau_u, \tau_v : \tau_u \cdot (i-1) \leq \tau_v \cdot i \leq \tau_u \cdot (i+1)$$

Proof.

Lower bound.

$$\begin{aligned}
& & \tau_u \cdot (i-1) \\
(\text{Lemma 1}) &\leq & \tau_v \cdot (1+\Delta) \cdot (i-1) \\
&= & \tau_v \cdot (i + i\cdot\Delta - 1 - \Delta) \\
(\forall i : i\cdot\Delta < 1) &\leq & \tau_v \cdot i
\end{aligned}$$

Upper bound.

$$\begin{aligned}
& & \tau_u \cdot i \\
(\text{Lemma 1}) &\leq & \tau_v \cdot (1+\Delta) \cdot i \\
&= & \tau_v \cdot (i + i\cdot\Delta) \\
(\forall i : i\cdot\Delta < 1) &\leq & \tau_v \cdot (i+1)
\end{aligned}$$

□

3.4 Formalization of the Notion of Local Time

In Figure 3.3 we have schematically shown the local time notions of two types of ECUs: master and slaves. All ECUs of our bus network will be configured by the operating system, running on top of it, before the communication starts. The configuration parameters written to each ECU represent such values as the slot length, the message length, number of slots in a round, a boolean flag for each slot indicating whether an ECU should behave as master or as a slave in that slot, etc. Except for the latter, all configuration parameters are equal for all ECUs. However, message transmissions cannot take place within an entire round due to shifting local time notions of different ECUs. Instead, there are time segments within every round, where all ECUs agree on the current slot as depicted in Figure 3.4. That is, in every slot, except for slot 0[1], there should be some 'space' before the transmission start and after the transmission end. In this section we will introduce these parameters formally.

First, we assume that a slot lasts at most T hardware cycles. With $\mathcal{N}$ participating ECUs, and therewith slots in a round, $\mathcal{N}$ slots last at most $\mathcal{N} \cdot T$ hardware cycles on every ECU. However, due to clock drift one ECU accomplishes these $\mathcal{N} \cdot T$ hardware cycles faster than some other in terms of global time. Now we want to know this difference in terms of local hardware cycles of an arbitrary ECU. From Lemma 1 we know that every clock period is at most $\Delta\%$ longer or shorter than any other. Thus, if some ECU accomplishes $\mathcal{N} \cdot T$ hardware cycles, then ECU_i will accomplish in this time x hardware cycles with

$$(1 - \Delta) \cdot \mathcal{N} \cdot T \leq x \leq (1 + \Delta) \cdot \mathcal{N} \cdot T$$

That is, for any number *off* with $\lceil \mathcal{N} \cdot T \cdot \Delta \rceil \leq \mathit{off} < T$ the following lemma holds.

Lemma 4 (Maximal drift within $\mathcal{N} \cdot T$ cycles)**.** *W.l.o.g., let τ_j, τ_i with $\tau_j \leq \tau_i$ be clock periods of ECU_i and ECU_j. Then:*

$$\mathcal{N} \cdot T \cdot \tau_i - \mathcal{N} \cdot T \cdot \tau_j \leq \mathit{off} \cdot \tau_i$$

Proof.

$$\begin{aligned} & & \mathcal{N} \cdot T \cdot \tau_i - \mathcal{N} \cdot T \cdot \tau_j = \mathcal{N} \cdot T \cdot (\tau_i - \tau_j) \\ (\text{Lemma 1}) & \leq & \mathcal{N} \cdot T \cdot \Delta \cdot \tau_i \\ & \leq & \lceil \mathcal{N} \cdot T \cdot \Delta \rceil \cdot \tau_i \\ (\text{Assumption on } \mathit{off}) & \leq & \mathit{off} \cdot \tau_i \end{aligned}$$

□

Lemma 4 says, that if ECU_i and ECU_j are two arbitrary ECUs, and if ECU_i is faster than ECU_j, then ECU_i outruns ECU_j within $\mathcal{N}$ slots by at most x cycles with $x \leq \mathit{off}$, as depicted in Figure 3.7.

Let $\alpha_i(r, s)$ denote the cycle of ECU_i, at which slot $s \neq 0$ of round r starts. Assume, that ECU_i is the sender in slot s, and that all slaves start the slot 0 roughly at the same time. When ECU_i enters cycle $\alpha_i(r, s)$, then by Lemma 4 every receiver ECU_j will be in cycle $\alpha_j(r, s)$ at last at the time, when ECU_i is in cycle $\alpha_i(r, s) + \mathit{off}$. That is, the sender is allowed to start its transmission in cycle $\alpha_i(r, s) + \mathit{off}$. Furthermore, by Lemma 4 we

[1] Since the slot 0 denotes the start of a new round and, hence, a new synchronization, the starting time of the slot 0 is roughly the same on all ECUs.

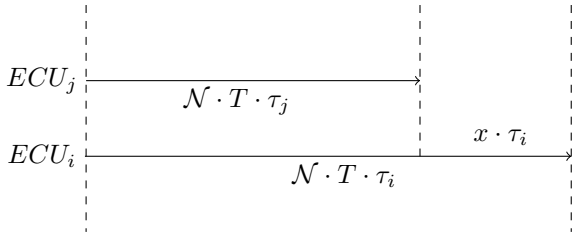


Figure 3.7: ECU_i Outruns ECU_j by x Cycles

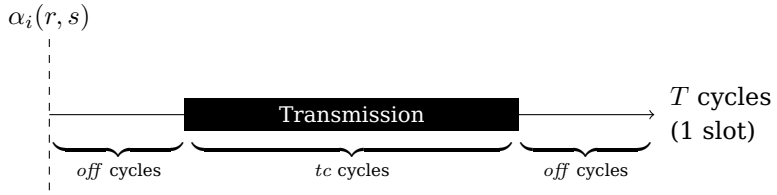


Figure 3.8: Slot Composition of a Sender ECU_i

also know, that every ECU_j will accomplish the slot s (reaching cycle $\alpha_j(r, s) + T$) at the earliest when ECU_i is in cycle $\alpha_i(r, s)+T-\mathit{off}$. From this follows, that the sender (ECU_i) is allowed to transmit a message in the local time interval $[\alpha_i(r, s)+\mathit{off} : \alpha_i(r, s)+T-\mathit{off}]$. Let tc be the number of cycles in a slot, which can be used to transmit a message:

$$tc = T - 2 \cdot \mathit{off}$$

As depicted in Figure 3.8, a slot length can be computed as $T = \mathit{off} + tc + \mathit{off}$. This decreases the upper bound of off:

$$\lceil \mathcal{N} \cdot T \cdot \Delta \rceil \leq \mathit{off} < \frac{T}{2}$$

Note, that the lower bound $\lceil \mathcal{N} \cdot T \cdot \Delta \rceil$ holds only if all ECUs will start the execution of the round simultaneously. However, in the real implementation, they start the execution with some maximal delay of d cycles, which depends on a concrete implementation of an ECU, in particular on the 'transmission pipeline', i.e., the length of the datapath of a transmitted signal in the sender and in the receiver. Moreover, due to effects of clock domain crossing signal transmission (see Section 4) an additional delay cycle may arise under certain circumstances. This increases the lower bound of off by additional $d + 1$ cycles:

$$\lceil \mathcal{N} \cdot T \cdot \Delta \rceil + d + 1 \leq \mathit{off} < \frac{T}{2}$$

Now we have defined everything we need to formalize the local time notion of ECUs. As mentioned before (Figure 3.3), the master ECU waits in the beginning of every round r for some amount of cycles, until all slaves have finished the previous round. Assuming that all slaves will receive the synchronization message at the start of round $r-1$ roughly at the same time (the difference is bounded by $d + 1$), it would be enough to wait for off cycles. For convenience's sake we count these initial off cycles of a new round as a part of slot 0 on the master.

Definition 3 (Local Time Notion of Master)**.** *Let $\alpha_0(0,0)$ be the first cycle of slot 0 of round 0 on the master. Then the first cycle of slot $s \leq \mathcal{N}$ of round r is denoted by $\alpha_0(r,s)$ and is defined as follows:*

$$\alpha_0(r,s) = \begin{cases} \alpha_0(0,0) & : \quad s = 0, r = 0 \\ \alpha_0(r-1,\mathcal{N}) & : \quad s = 0, r \neq 0 \\ \alpha_0(r,s-1) + T & : \quad \textit{otherwise} \end{cases}$$

For example, the start time of slot 0 of the 4'th round $\alpha_0(4,0)$ is computed as:

$$\begin{aligned} \alpha_0(4,0) &= \alpha_0(3,\mathcal{N}) \\ &= \alpha_0(3,\mathcal{N}-1) + T \\ &= \alpha_0(3,\mathcal{N}-2) + T + T \\ &= \alpha_0(3,0) + \underbrace{T + T + \ldots + T}_{\mathcal{N} \cdot T} \\ &= \alpha_0(3,0) + \mathcal{N} \cdot T \end{aligned}$$

Thus, the master ECU makes exactly $\mathcal{N} \cdot T$ cycles between rounds i and $i+1$.

Lemma 5 (α_0 Arithmetic)**.**

$$\forall s \in [0 : \mathcal{N} - 1] : \alpha_0(r,s) = \alpha_0(r,0) + s \cdot T$$

Proof. By induction on slot s.

1. **Induction base:** $s = 0$.

$$\alpha_0(r,s) = \alpha_0(r,0) = \alpha_0(r,0) + 0 \cdot T = \alpha_0(r,0) + s \cdot T$$

2. **Induction step:** $s \to s+1$.

$$\alpha_0(r,s+1) \overset{\text{Def. 3}}{=} \alpha_0(r,s) + T \overset{\text{IH}}{=} \alpha_0(r,0) + s \cdot T + T = \alpha_0(r,0) + (s+1) \cdot T$$

Note that by IH we abbreviate Induction Hypothesis, which is the claim to show for smaller instance of the inductive variable.

□

A new round on a slave ECU starts in the cycle when the synchronization message was sampled and processed. This cycle depends on the concrete implementation of the bus interface of the ECU and delays caused by the effects of signal transmission across different clock domains. We will instantiate the round start time of a slave ECU by a concrete definition later. At the synchronization all slaves adjust their local counters to the value *off*, which should denote the master's local time, when the synchronization message was initiated. The first slot of a round (slot 0) of a slave lasts then only $tc + \mathit{off}$ cycles. The initial offset of *off* cycles is not necessary in the first slot, since the slot starts right after the synchronization, and the clocks of all ECUs are ideally synchronized.

Definition 4 (Local Time Notion of a Slave)**.** *Let $\alpha_i(r,0)$ be the first local cycle of round r. Then $\alpha_i(r,s)$ for arbitrary r and $s < \mathcal{N}$ is defined as follows.*

$$\alpha_i(r,s) = \begin{cases} \alpha_i(r,0) & : \quad s = 0 \\ \alpha_i(r,0) + tc + \mathit{off} & : \quad s = 1 \\ \alpha_i(r,s-1) + T & : \quad \textit{otherwise} \end{cases}$$

Thus, the start time of every non-zero slot depends on the start time of the previous slot. Whereas, the start time of slot 0 of every round depends on the master's synchronization message.

Lemma 6 (α_i Arithmetic)**.**

$$\forall s \in [1 : \mathcal{N} - 1] : \alpha_i(r, s) = \alpha_i(r, 0) + tc + \mathit{off} + (s - 1) \cdot T$$

Proof. By induction on slot s.

1. **Induction base:** $s = 1$.

$$\begin{aligned} \alpha_i(r, s) &= \alpha_i(r, 1) \overset{\text{Def. 4}}{=} \alpha_i(r, 0) + tc + \mathit{off} = \alpha_i(r, 0) + tc + \mathit{off} + 0 \cdot T \\ &= \alpha_i(r, 0) + tc + \mathit{off} + (s - 1) \cdot T \end{aligned}$$

2. **Induction step:** $s \to s + 1$. We have to make a case distinction on s according to Definition 4.

   (a) Case $s = 0$.

$$\alpha_i(r, s + 1) = \alpha_i(r, 1)$$

   The case follows from induction base.

   (b) Case $s > 0$.

$$\begin{aligned} \alpha_i(r, s + 1) &\overset{\text{Def. 4}}{=} \alpha_i(r, s) + T \overset{\text{IH}}{=} \alpha_i(r, 0) + tc + \mathit{off} + (s - 1) \cdot T + T \\ &= \alpha_i(r, 0) + ((s + 1) - 1) \cdot T \end{aligned}$$

□

CHAPTER

FOUR

BUS MODEL FOR CLOCK DOMAIN CROSSING COMMUNICATION

In the previous chapter we have (i) introduced the communication scheme between ECUs, (ii) formalized the notion of a clock relating the local time of an ECU to the common global time, and (iii) formalized the notion of local time of an ECU in terms of slot and round start times.

Computing the maximal local time deviation off of two arbitrary ECUs within one round and using it as a waiting offset at the beginning and at the end of a slot provides us with a sampling window in the remaining time of a slot. The sampling window is a time interval within a slot, where all ECUs agree on the current slot. The sampling window will be used for a message transmission.

Until now, we skipped the interconnection of ECUs and concentrated on the timing of their behaviour. In this chapter we will introduce the formalization of the low-level interconnection of several ECUs. We start with an overview of the hardware connection of an ECU with a bus in Section 4.1. We introduce in Section 4.2 a precise timing model of a register, which allows us to argue about the content of this register outside of its clock domain. Afterwards, we use this model to formalize in Section 4.3 a low-level signal transmission from one clock domain to another. In this section we will present previous verification efforts, their improvement and extension. Finally, in Section 4.4 we will extend the model of hardware interconnection to $\mathcal{N}$ clock domains.

4.1 Hardware Interconnection

In the beginning of the previous chapter we have schematically shown the topology of our network: several nodes interconnected by a bus (see Figure 3.1). The more detailed scheme of the connection of a single ECU to the bus is depicted in Figure 4.1. The bus is connected to every ECU with open collector outputs.

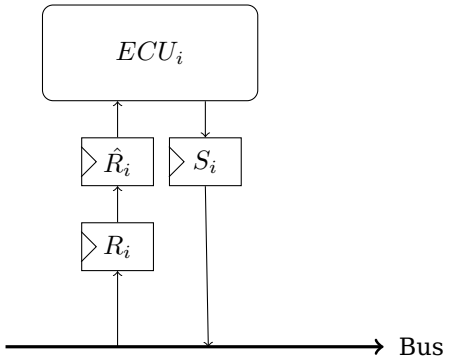


Figure 4.1: Connection of an ECU to the Bus

Output signal of an ECU. The output of every ECU_i will be written first to a *send register* S_i. To mimic the behaviour of the open collector bus, we model our bus value as a *conjunction* of outputs of send registers of all ECUs. The send register value of a non-sender ECU should be a '1', s.t. at any message transmission time the bus value is equal to the send register value of the sending ECU.

However, since every send register S_i is part of a different clock domain and since the bus should be defined at every point in real time, we have to deal with undefined register values. For example, if at some given real time the ECU_i is updating its send register S_i at cycle c, then between cycles c and $c+1$, the content of S_i is undefined. We describe this undefined value by Ω. Note that Ω may describe:

- a metastable value – a voltage between thresholds recognized as 0 or 1;
- a stable but unknown boolean value.

Moreover, we use a *weaker* assumption about the outcome of a conjunction of register outputs, and compute a conjunction of any value with an undefined value as a (possibly new) undefined value again:

$$\forall x \in \mathbb{B} : \Omega \wedge x = x \wedge \Omega = \Omega$$

Assume, $Out_{S,u}$ is the analog signal modeling the output of the send register S at any given real time:

$$Out_{S,u} : \mathbb{R} \to \{0, 1, \Omega\}$$

Then the bus is also modeled as an *analog signal*

$$bus : \mathbb{R} \to \{0, 1, \Omega\}$$

and is computed as a conjunction of all send register outputs of all ECUs:

$$bus(t) = \bigwedge_u Out_{S,u}(t)$$

Input signal to an ECU. As mentioned before, the bus value can be an undefined value Ω. To deal with this fact, we make use of a common technique in hardware design. Before the bus signal enters the receiver ECU_i, it will be consecutively sampled into two *receive registers* R_i and $\hat{R}_i$. It is possible, that an undefined bus value will be sampled

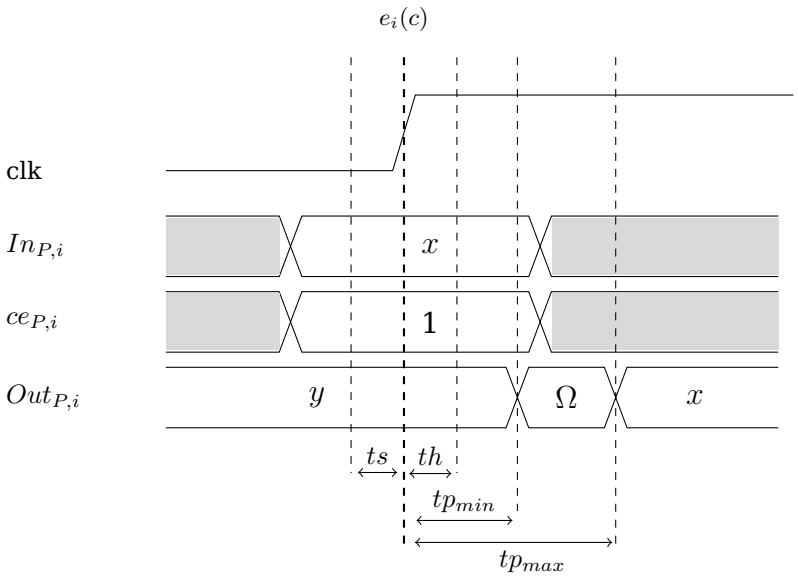


Figure 4.2: An Update in the Detailed Register Model

into the register R_i. In this case, its content either flips to any boolean value or remains metastable hanging between voltages, defining 1 and 0. However, the probability that the content of R_i remains metastable for an entire cycle and that this metastable value will be sampled to $\hat{R}_i$ in the next cycle, is for practical purposes nearly zero. Thus, we assume, that if an undefined value will be sampled into the register R_i in cycle c, then $\hat{R}_i$ will be either 1 or 0 in cycle $c+1$.

4.2 Detailed Register Model (DRM)

Since the time of a digital circuit computation is represented by hardware cycles, a register has the following straightforward semantic. Let R be a register. Let Rce be the clock enable signal of R, let Rin be the input signal of R, and let R^0 be the initial content of R. Then, the content of R in cycle $c \in \mathbb{N}_0$ is defined as follows.

$$R^{c+1} = \begin{cases} Rin^c & : \quad Rce^c = 1 \\ R^c & : \quad \text{otherwise} \end{cases}$$

Thus, the register content is specified only for every given hardware cycle. There is no need to define some intermediate states of a register. However, to formalize a signal transmission across different clock domains, we have to formalize the content (the output signal) of a register for any given real time, also during its update time. This output would be the signal, visible by other ECUs. Therefore, we will extend the discrete model of a digital register with more precise timing parameters obtained by formalization of data sheets for hardware components [Sch07]. The parameters are *setup time* ts, *hold time* th, as well as minimum and maximum *propagation delays* tp_{min} and tp_{max}. Moreover, we model all inputs and outputs of an analog register as *analog signals*. For every register P in clock domain i we denote by:

- $In_{P,i}$ its analog input signal;

- $ce_{P,i}$ its analog clock enable signal;
- $Out_{P,i}$ its analog output signal.

Let P be some register of ECU_i, which holds some value y and will be updated at cycle c with value x. Then the output of P outside of the clock domain i is modeled by the analog output signal $Out_{P,i}$ and behaves during an update as depicted in Figure 4.2. To update P both of his analog input signals $In_{P,i}$ and $ce_{P,i}$ should be stable around the clock edge c, namely, in the time interval $[e_i(c) - ts : e_i(c) + th]$:

$$\forall t \in [e_i(c) - ts : e_i(c) + th] : In_{P,i}(t) = x \wedge ce_{P,i}(t) = 1$$

If both signals were stable during that time interval, the output of the modeled register i) does not change before $e_i(c)+tp_{min}$, ii) becomes undefined in $(e_i(c) + tp_{min}, e_i(c) + tp_{max})$, and iii) assumes the new value $In_{P,i}(e_i(c)) = x$ from $e_i(c) + tp_{max}$ until the next clock edge plus the minimum propagation delay: $e_i(c) + \tau + tp_{min}$. Note that we make the following assumptions about the relation of the timing parameters to each other and to the clock period of an arbitrary ECU.

Assumption 2 (Timing Parameters)**.** *Let ECU_i be arbitrary ECU. Moreover, let (γ_i, τ_i) be the clock of ECU_i. Then we assume:*

1. $0 \leq ts$
2. $0 \leq th$
3. $0 \leq tp_{min} < tp_{max}$
4. $tp_{max} < \tau_i$
5. $ts + th + tp_{max} - tp_{min} \leq 0.9 \cdot \tau_i$

The inequalities 1 and 2 are rather natural assumptions about time intervals (they cannot be negative). The third inequality is justified by the fact, that the observed output of a register cannot change its value in no time at all. With the fourth inequality we require, that a register changes its value during the current cycle. The inequality 5 is needed later in our proofs to relate the introduced timing parameters to the clock period of an arbitrary clock. This assumptions can be easily fulfilled by the current technology.

We formalize the behaviour of an analog register in the following definitions.

Definition 5 (Stable Signal)**.** *Let s be some signal and let c be a cycle of some ECU_j. We define a predicate indicating stability of s around the clock edge of cycle c as follows:*

$$stable_i(s, c) = \exists b \in \mathbb{B} : \forall t \in [e_i(c) - ts : e_i(c) + th] : s(t) = b$$

Definition 6 (Detailed Register)**.** *Let P be some register in clock domain i, and c be some local cycle of this clock domain. The analog output signal of P is defined for all times $t \in (e_i(c) + tp_{min} : e_i(c + 1) + tp_{min}]$ as follows:*

$$Out_{P,i}(t) = \begin{cases} In_{P,i}(e_i(c)) & : \; e_i(c) + tp_{max} \leq t \; \wedge stable(In_{P,i}, c) \wedge \\ & \quad stable(ce_{P,i}, c) \wedge ce_{P,i}(e_i(c)) = 1 \\ Out_{P,i}(e_i(c)) & : \; stable(ce_{P,i}, c) \wedge ce_{P,i}(e_i(c)) = 0 \\ \Omega & : \; otherwise \end{cases}$$

This simple part of the definition which models completely regular clocking has an important consequence. Imagine we have $In_{P,i}(e_i(c)) = Out_{P,i}(e_i(c))$, i.e., we clock the old value again into the register. Then, in the digital abstraction the output is constant in two consecutive cycles. In the detailed hardware model however (and in reality) we have a possible spike $Out(t) = \Omega$ for $t \in (e_i(c) + tp_{min} : e_i(c) + tp_{max})$. If we want to guarantee, that the output really stays stable ($Out(t) = Out(e_i(c))$ for $t \in [e_i(c) : e_i(c) + \tau]$), we need to disable clocking at edge $e_i(c)$: $ce(t) = 0$ for $t \in [e_i(c) - ts : e_i(c) + th]$. If clocking is properly enabled but setup or hold time is violated or the input is undefined, then after the maximum propagation delay the output stays 0, 1 or undefined (the latter case models metastability). In all these cases we model this by Ω.

$$\exists a \in \{0, 1, \Omega\} : \forall t \in [e_i(c) + tp_{max}, e_i(c) + \tau_i + tp_{min}] : Out(t) = a$$

Metastability is highly unlikely to occur. As mentioned before, the possibility that clocking of a metastable value of register R into a second register $\hat{R}$ results in the metastability of $\hat{R}$ too is so unlikely that one models it as impossible. The result is an unpredictable digital value:

$$\exists a \in \{0, 1\} : \forall t \in [e_i(c) + tp_{max} : e_i(c) + \tau_i + tp_{min}] : Out_{\hat{R},u}(t) = a$$

Fortunately, in our case we can restrict the use of the detailed model to the part of the hardware, where clock domain crossing occurs. This portion consists of the bus, the send register S and the receiver register R as depicted in Figure 4.1. The remaining hardware is partitioned into clock domains, one for each ECU. In each clock domain u we can abstract from the detailed model local digital hardware configurations h_u with, e.g., register components $h_u.R \in \{0, 1\}$, hardware cycles $i \in \mathbb{N}$, configuration h_u^i during cycle i in the following way. We couple local cycle numbers i on ECU_u (hence, in clock domain u) with the real time $e_u(i)$ of the i'th local clock edge on ECU_u using Definition 2:

$$e_u(i) = \gamma_i + \tau_i \cdot i$$

This edge starts local cycle i on ECU_u. For $t \in [e_u(i) + tp_{max} : e_u(i) + \tau_u]$, i.e., after the propagation delay the output $Out_{R,u}(t)$ of register R on ECU_u is stable for the rest of local cycle i of ECU_u. We abstract this value to the digital register value:

$$h_u^i.R = dig(Out_{R,u}, e_u(i + 1)) \tag{4.1}$$

The function $dig(s, t)$ digitizes the output of signal s at real valued time t:

$$dig(s, t) := \textbf{if } s(t) \neq \Omega \textbf{ then } s(t) \textbf{ else } x, \text{ for some } x \in \{0, 1\} \tag{4.2}$$

As mentioned before, we assume, that a metastable value is never sampled into register $\hat{R}$. However, modeling the input of R as in 4.1 we also exclude the appearance of an undefined value in the digital model completely. The resulting model is still sound, because the first register R is never used for any computations, but only for suppressing metastable values in a *real* hardware implementation.

If we now define a hypothesis 'Correct detailed timing analysis' stating that for all local cycles setup and hold times for register inputs are met[1], then one can show that the hardware configurations h_u^i we just abstracted are exactly the configurations one would get by applying the transition function δ_H of ordinary digital hardware models.

[1] Summing propagation delays along appropriate paths.

Soundness: Assume correct detailed timing analysis. Then:

$$\forall i : h_u^{i+1} = \delta_H(h_u^i)$$

This justifies the use of ordinary digital logic within clock domains and restricts the use of the detailed model to the boundaries between the domains, in our case the bus.

4.3 Clock Domain Crossing Signal Transmission

Schmaltz has formally verified [Sch07] a bit transmission across *two different* clock domains.[2] His theorem assumes a direct connection of the input and output signals of two registers interpreted with DRM.

Let R and S be two registers from two different clock domains u and v, respectively; they are interpreted using DRM. Assume the analog input signal $In_{R,u}$ of R is directly connected to the analog output signal $Out_{S,v}$ of S:

$$In_{R,u}(t) = Out_{S,v}(t)$$

Assume the clock enable signal of the send register S is active in cycle c and the clock enable signal of R is always active. By the definition of DRM, the output of S will change right after $e_v(c) + tp_{min}$. We want to know the first cycle at which the receive register R will 'notice' the change of its input signal changing its content. We call such cycle the *next affected cycle* and define it as follows [Pau05].

Definition 7 (Next Affected Cycle)**.** *Let u and v be clock domains. Then, $cy_{u,v}(c)$ is the next effected cycle in clock domain u by cycle c of clock domain v and is defined as:*

$$cy_{u,v}(c) = min\{\ x \mid e_v(c) + tp_{min} < e_u(x) + th\}$$

Thus, the receiver's next affected cycle by sender's cycle c is the first cycle, whose hold time ends after the output change of S.

For further computations, we need to estimate the timing relation between the clock edge $e_v(c)$ and the clock edge of the next cycle of clock domain u affected by the cycle c of clock domain v. By Definition 7 we know the lower bound of the next affected cycle:

$$e_v(c) + tp_{min} - th < e_u(cy_{u,v}(c))$$

The next lemma establishes an upper bound of the next affected cycle. We want to show, that if we fix the next affected cycle for some given cycle c as depicted in Figure 4.3, then the previous cycle of the next affected cycle, is smaller than $e_v(c) + tp_{min} - th$.

Lemma 7.

$$e_u(cy_{u,v}(c) - 1) < e_v(c) + tp_{min} - th$$

Proof. By Definition 7, $cy_{u,v}(c)$ is the *first* cycle such that:

$$e_v(c) + tp_{min} < e_u(cy_{u,v}(c)) + th$$

That is, the hold time of cycle $cy_{u,v}(c)$ ends somewhere during interval:

$$(e_v(c) + tp_{min} : e_v(c) + tp_{min} + \tau_u]$$

[2] Formal proofs can be found at [MSB].

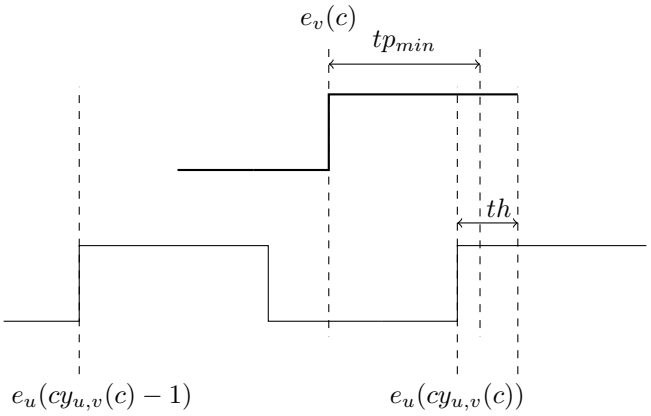


Figure 4.3: Next Affected Cycle

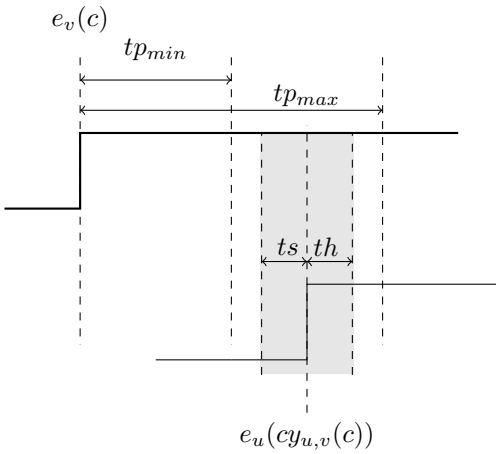


Figure 4.4: Violation of Timing Parameters of the Next Affected Cycle

Which gives us:

$$\begin{aligned} & e_u(cy_{u,v}(c)) + th \leq e_v(c) + tp_{min} + \tau_u \\ \text{(Definition 2)} \quad \equiv \quad & e_u(cy_{u,v}(c) - 1) \leq e_v(c) + tp_{min} - th \end{aligned}$$

□

Obviously, if the set up time of the next affected cycle starts during time interval $(e_v(c) + tp_{min} : e_v(c) + tp_{max})$ then the input signal $In_{R,u}(t) = Out_{S,v}(t)$ of register R is unstable for

$$t \in (e_v(c) + tp_{min} : e_v(c) + tp_{max}) \cap [e_u(cy_{u,v}(c)) - ts : e_u(cy_{u,v}(c)) + th]$$

One possible case for the intersection of these intervals is depicted in Figure 4.4.

Thus, by Definition 6 and by violated timing parameters ($\neg stable(In_{R,u}, cy_{u,v}(c))$) the analog output of register R gets undefined during the next affected cycle:

$$\forall t \in [cy_{u,v}(c) + tp_{max} : cy_{u,v}(c) + \tau_u + tp_{min}] : Out_{R,u}(t) = \Omega$$

That is, the signal transmission across different clock domains may start with sampling of an undefined value. In this case, even if all timing parameters are met during subsequent cycles, the transmission of the correct value will be delayed by one cycle. We formalize this delay cycle as follows.

Definition 8 (Delay Cycle)**.** *Let u, v be two clock domains. We define for a cycle c of clock domain v a delay cycle in clock domain u as:*

$$\sigma_{u,v}(c) = \begin{cases} 1 & : \quad e_u(cy_{u,v}(c)) - ts < e_v(c) + tp_{max} \\ 0 & : \quad \textit{otherwise} \end{cases}$$

4.3.1 Transmission Correctness Across Two DRM Registers

Now we can prove a correct bit transmission between two registers modeled with DRM. That is, the lemma only shows a transmission of a signal from the analog input of one register to the analog output of another. This lemma will be used afterwards, to show a transmission of a *digital* signal from one clock domain to another.

Lemma 8 (Analog Signal Transmission)**.** *We fix the following variables:*

- *u, v as clock domains;*
- *S as send register of clock domain v;*
- *R as receive register of clock domain u;*
- *$n, m \in \mathbb{N}$ as positive integers with $m = 6, n = 7$;*
- *$c \in \mathbb{N}$ as cycle of clock domain v;*
- *$\xi = cy_{u,v}(c)$ as receiver cycle.*

Premises:

(a) The input signal of S is stable during cycle c, and the clock enable signal is stable during cycle c and n subsequent cycles:

$$stable(In_{S,v}, c) \ \wedge \forall i \leq n : stable(ce_{S,v}, c + i)$$

(b) The clock enable signal of S is active in cycle c and inactive during n subsequent cycles:

$$ce_{S,v}(e_v(c)) = 1 \ \wedge \forall i \in [1 : n] : ce_{S,v}(e_v(c + i)) = 0$$

(c) The clock enable signal of R is always active:

$$\forall i \in \mathbb{N}, t \in [e_u(i) - ts : e_u(i) + th] : ce_{R,u}(t) = 1$$

(d) The analog input signal of R is the analog output signal of S during $n+1$ local cycles of clock domain of S:

$$\forall t \in (e_v(c) + tp_{min} : e_v(c + n) + tp_{min}] : In_{R,u}(t) = Out_{S,v}(t)$$

Then: *the output signal of R at $m+1$ cycle edges starting with cycle $\xi + \sigma_{u,v}(c) + 1$ is equal to the input signal of S at $e_v(c)$:*

$$\forall i \in [\xi + \sigma_{u,v}(c) + 1 : \xi + \sigma_{u,v}(c) + 1 + m] : Out_{R,u}(e_u(i)) = In_{S,v}(e_v(c))$$

Proof. We consider the analog output of S during time interval:

$$(e_v(c) + tp_{min} : e_v(c+n) + tp_{min}]$$

Since clock enable signal $ce_{S,v}$ was active and stable at the clock edge $e_v(c)$ (premises (a) and (b)), by Definition 6 we have:

$$\forall t \in e_v(c) + (tp_{min} : tp_{max}) : Out_{S,v}(t) = \Omega$$
$$\forall t \in [e_v(c) + tp_{max} : e_v(c+1) + tp_{min}] : Out_{S,v}(t) = In_{S,v}(e_v(c))$$

We also know from the same premises, that $ce_{S,v}$ was stable and inactive during the next $n = 7$ cycles. Hence, the content of S did not change during these n cycles. Applying Definition 6 we get:

$$\forall t \in (e_v(c+1) + tp_{min} : e_v(c+1+n) + tp_{min}] : Out_{S,v}(t) = Out_{S,v}(e_v(c+1)) \tag{4.3}$$

Putting all together and applying premise (d) we can describe the analog input signal of register R:

$$\forall t \in (e_v(c) + tp_{min} : e_v(c) + tp_{max}) : In_{R,u}(t) = \Omega \tag{4.4}$$
$$\forall t \in [e_v(c) + tp_{max} : e_v(c+1+n) + tp_{min}] : In_{R,u}(t) = In_{S,v}(e_v(c)) \tag{4.5}$$

What remains to show, is a proof of a correct propagation of the received signal to the output of register R.

By premise (c) we know, that R always samples its input value. Moreover, by the same premise and Definition 5 we know that the clock enable signal of R is always stable. Thus, the output of R for any clock edge $e_u(i)$ with

$$i \in [\xi + \sigma_{u,v}(c) + 1 : \xi + \sigma_{u,v}(c) + 1 + m]$$

is equal to the input signal of the previous clock edge by Definition 6:

$$\forall i \in [\xi + \sigma_{u,v}(c) + 1 : \xi + \sigma_{u,v}(c) + 1 + m] : Out_{R,u}(e_u(i)) = In_{R,u}(e_u(i-1)) \tag{4.6}$$

However, this holds only if $In_{R,u}$ is stable around clock edge $e_u(i-1)$. We consider the value of $In_{R,u}$ at time interval $[e_u(i-1) - ts : e_u(i-1) + th]$. Obviously, the claim of the main lemma follows from 4.5 and 4.6 if we prove for all $i \in [\xi + \sigma_{u,v}(c) + 1 : \xi + \sigma_{u,v}(c) + 1 + m]$:

$$[e_u(i-1) - ts : e_u(i-1) + th] \subseteq [e_v(c) + tp_{max} : e_v(c+1+n) + tp_{min}] \tag{4.7}$$

For convenience, we denote the interval on the right side by I and sketch the proof of 4.7.

Case $\sigma_{u,v}(c) = 0$ (Figure 4.5). We know by Definition 8:

$$e_u(\xi) - ts \geq e_v(c) + tp_{max}$$

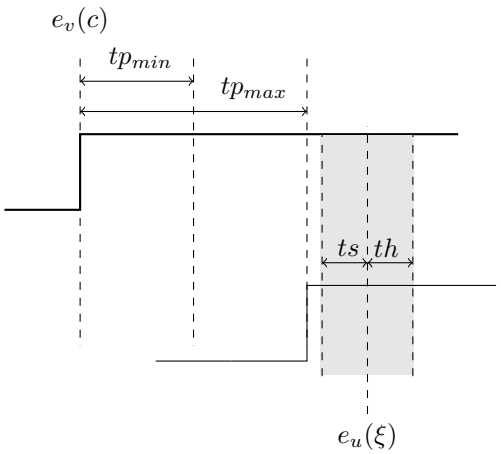


Figure 4.5: No Violation of Timing Parameters of the Next Affected Cycle

Hence, for the smallest $i = \xi + 1$ we have:

$$e_u(\xi) - ts = e_u(i-1) - ts \geq e_v(c) + tp_{max} \tag{4.8}$$

Hence, $e_u(i-1) - ts$ is in I.

Consider $i > \xi + 1$ and $i \leq \xi + 1 + m$. By Lemma 7 we have

$$e_u(\xi - 1) + th \leq e_v(c) + tp_{min} \tag{4.9}$$

By Lemma 1, $m = 6, n = 7$, and Lemma 3 we show:

$$(m+1) \cdot \tau_u \leq (n+1) \cdot \tau_v \tag{4.10}$$

With help of 4.10 we extend 4.9 to:

$$e_u(\xi - 1) + th + (m+1) \cdot \tau_u \leq e_v(c) + tp_{min} + (n+1) \cdot \tau_v$$

Applying Definition 2 we rewrite the inequation as:

$$e_u(\xi + m + 1 - 1) + th \leq e_v(c + 1 + n) + tp_{min} \tag{4.11}$$

Finally, we show:

$$\begin{aligned} & e_u(i-1) + th && (4.12)\\ (\text{since } i \leq \xi + m + 1) \quad \leq \quad & e_u(\xi + m + 1 - 1) + th && (4.13)\\ (\text{Inequation 4.11}) \quad \leq \quad & e_v(c + 1 + n) + tp_{min} && (4.14) \end{aligned}$$

Thus, in case of absence of the delay cycle $\sigma_{u,v}(c)$, we can show, that the input signal $In_{R,u}$ is stable and is equal to analog output of S around the clock edge $e_u(i-1)$, because it lies within interval I by 4.8 and 4.14.

Case $\sigma_{u,v}(c) = 1$ (Figure 4.4). The smallest i is $\xi + 2$. We conclude:

$$\begin{aligned} (\text{Assumption 2}) \quad & e_v(c) + tp_{max} \leq e_v(c) + tp_{min} + \tau_u && (4.15)\\ (\text{Definition 7}) \quad \leq \quad & e_u(\xi) + th + \tau_u && (4.16)\\ (\text{Definition 2}) \quad = \quad & e_u(\xi + 1) + th && (4.17)\\ (\text{since } i = \xi + 2) \quad = \quad & e_u(i-1) + th && (4.18) \end{aligned}$$

Thus, in presence of a delay cycle, the clock edge $e_u(i-1)$ still lies within the interval I.

Consider an $i > \xi + 2$ with $i \leq \xi + 2 + m$. It remains to show that $e_u(i-1)$ does not cross the upper boundary of I:

$$e_u(\xi + 1 + m) + th \leq e_v(c + 1 + n) + tp_{min}$$

Applying Definition 2 we rewrite this as:

$$e_u(\xi) - ts \leq e_v(c) + tp_{max} + (tp_{min} - tp_{max} - ts - th - (1+m) \cdot \tau_u + (n+1) \cdot \tau_v)$$

If we now show:

$$0 \leq tp_{min} - tp_{max} - ts - th - (1+m) \cdot \tau_u + (n+1) \cdot \tau_v \tag{4.19}$$

The claim would follow by Definition 8. We rewrite 4.19:

$$\begin{aligned} tp_{max} + ts + th - tp_{min} &\leq (n+1) \cdot \tau_v - (1+m) \cdot \tau_u \\ \text{(Lemma 1)} &= (n+1) \cdot (1-\Delta) \cdot \tau_u - (1+m) \cdot \tau_u \\ &= ((n+1) \cdot (1-\Delta) - 1 - m) \cdot \tau_u \\ &= (n - n \cdot \Delta + 1 - \Delta - 1 - m) \cdot \tau_u \\ &= (n - m - (n+1) \cdot \Delta) \cdot \tau_u \end{aligned}$$

For $n = 7$ we get:

$$\begin{aligned} tp_{max} + ts + th - tp_{min} &\leq (7 - 6 - (7+1) \cdot \Delta) \cdot \tau_u \\ \text{(Lemma 2)} &= (1 - 8 \cdot 0.0078) \cdot \tau_u \\ &= 0.96 \cdot \tau_u \end{aligned}$$

Which is fulfilled by Assumption 2. Thus, we have shown, that independently of the appearance of the delay cycle $\sigma_{u,v}(c)$ the edge $e_u(i-1)$ never crosses the boundaries of I. □

4.3.2 Improvement of Lemma 8

The original version of Lemma 8 was not applicable in a bus network with multiple senders and receivers. The assumption of a direct connection between the sender and the receiver (premise (d)) was not restricted in the original version to 8 sender cycles, but was allowed for all times:

$$\forall t : In_{R,u}(t) = Out_{S,v}(t)$$

This narrows the usage of the theorem to cases, where the send and the receive registers are connected forever. However, in the case of a bus, we can abstract the bus to a direct connection only at certain times, as we will see later. Hence, we have weakened the premise (d) to the minimal time range as it is presented in Lemma 8 and adjusted the proof.

The reason, why this bug remained undiscovered, is because the Lemma 8 was never applied in context of a bus system, where the sender changes from time to time.

4.3.3 Digital Signal Transmission Across Different Clock Domains

While Lemma 8 shows the transmission correctness of an analog signal across two different detailed registers, we need to extend this result to a correctness of a transmission of *digital* signals across different clock domains. We model the digital input of the received register as a digitized analog value using the function dig (cf. 4.1). To convert digital bits to analog signals, e.g., to model the inputs of register S, Schmaltz assumes the existence of a special function $conv(bv) : \mathbb{R} \to \{0, 1, \Omega\}$, which converts a bitvector bv to a signal. He extended the previous Lemma to the following theorem.

Theorem 1. *Fix the same variables as in Lemma 8 and additionally:*

- u, v *as clock domains;*
- $bvce_S$ *as bitvector, s.t.* $bvce_S[i]$ *is the bit sent as clock enable signal to* S *in cycle* i*;*
- $bvin_S$ *as bitvector, s.t.* $bvin_S[i]$ *is the bit sent as input signal to* S *in cycle* i*;*

Premises

(a) Let $bv_{P,u}$ *be a bitvector, where every bit* $bv_{P,u}[i]$ *is sent as input signal to register* P *of clock domain* u *in cycle* i*. Then there exists a conversion function* $conv(bv_{P,u})$ *which converts every bit* $bv_{P,u}[i]$ *to an analog value around clock edge* $i + 1$*:*

$$\forall i : \forall t \in [e_u(i+1) - ts : e_u(i+1) + th] : conv(bv_{P,u})(t) = bv_{P,u}[i]$$

(b) The bit sent as clock enable signal to S *in cycle* $c - 1$ *is a '1'; in the subsequent* n *cycles it is a '0':* $bvce_S[c-1] = 1 \ \wedge \ \forall i \in [1 : n] : bvce_S[c-1+i] = 0$

(c) The clock enable signal of R *is always active:* $\forall i, t \in [e_u(i) - ts : e_u(i) + th] : ce_{R,u}(t) = 1$

(d) The input signal of R *is the output signal of* S *during* $n + 1$ *local cycles of clock domain of* S*:*

$$\forall t \in (e_v(c) + tp_{min} : e_v(c+n) + tp_{min}] : In_{R,u}(t) = Out_{S,v}(t)$$

Then, the digitized content of register R *is equal to the digital input of* S *in cycle* $c - 1$ *during* $m + 1$ *cycles of clock domain of register* R*:*

$$\forall i \in [0 : m] : h_u.R^{\xi + \sigma_{u,v}(c)+i} = bvin_S[c-1]$$

Proof. The proof is rather straightforward and follows from Lemma 8 and Equation 4.1. First we show, that all premises of Lemma 8 are satisfied. The premises (a) and (b) of Lemma 8 will be satisfied by premises (a) and (b) of Theorem 1 if we instantiate the input and clock enable signals of S by corresponding bitvectors converted to signals:

$$\forall t : In_{S,v}(t) = conv(bvin_S)(t) \ \wedge \ ce_{S,v}(t) = conv(bvce_S)(t) \tag{4.20}$$

Since according to premise (a) of Theorem 1 the function $conv(bvce_S)(t)$ provides for every $t \in [e_v(i+1) - ts : e_v(i+1) + th]$ the boolean value of bit $bvce_S[i]$, such a signal is always stable and corresponds to the digital values of the bitvectors (premise (b) of

Theorem 1). Premises (c) and (d) are identical in both cases. Hence, we can use the result of Lemma 8 which we rewrite as:

$$\forall i \in [0:m] : Out_{R,u}(e_u(\xi + \sigma_{u,v}(c) + 1 + i)) = In_{S,v}(e_v(c)) \tag{4.21}$$

By Equation 4.1 we get:

$$\forall i \in [0:m] : R^{\xi+\sigma_{u,v}(c)+i} = dig(Out_{R,u}, e_u(\xi + \sigma_{u,v}(c) + i + 1)) \tag{4.22}$$

Furthermore, with 4.20, i.e., an instantiation of signal $In_{S,v}(e_v(c))$ by $conv(bvin_S)(e_v(c))$, and by premise (a) we know:

$$In_{S,v}(e_v(c)) = bvin_S[c-1] \tag{4.23}$$

The claim of Theorem 1 follows now from 4.21, 4.22, 4.23, and by definition of dig function. □

Thus, Theorem 1 proves, that if a send register S of clock domain v does not change its value for 8 cycles, then the receive register R of clock domain u will successfully sample this value for at least 7 cycles. The direct linking of registers should be considered as an abstraction of the bus, when there is no bus contention.

4.3.4 Improvement of Theorem 1

The shortcoming of Theorem 1 is the fact, that Schmaltz has assumed in premise (a) the existence of bitvectors $bvin_S$ and $bvce_S$ modeling the lists of input signal to the digital register S independently of a local cycle c, at which register S will be updated. Thus, if we would use Theorem 1 within one clock domain for two different instantiations of cycle c, e.g., by cycles c_1 and c_2, then we have to provide instantiations for both bitvectors $bvin_S$ and $bvce_S$, such that premise (b) is fulfilled for cycles c_1 and c_2 simultaneously. That is, e.g., we have to show that one single vector $bvce_S$ satisfies simultaneously the assumption:

$$bvce_S[c_1] = 1 \wedge bvce_S[c_2] = 1$$

But this would only work, if we could instantiate $bvce_S$ with a list of bits sent as clock enable signals to S in the *entire* (possibly infinite) trace of the clock domain. However, it is impossible to model infinite bitvectors in the Isabelle language, so we could not instantiate these bitvectors with any meaningful values derived from the computational traces, since the trace function provides a clock domain state for a concrete hardware cycle. As a solution, these bitvectors were replaced with functions, which map from local hardware cycles to corresponding bitvectors:

$$bvin_S : \mathbb{N} \to \mathbb{B}^*$$
$$bvce_S : \mathbb{N} \to \mathbb{B}^*$$

Every of these functions, takes a hardware cycle c and produces a bitvector containing the list of bits sent as digital signals to the corresponding register input from cycle 0 and up to cycle c. For example, $bvin_S(5)$ would produce the list of 6 bits, sent as input to register S in cycles 0 to 5. We can fix now premise (b) as follows:

$$bvce_S(c-1)[c-1] = 1 \ \wedge \ \forall i \leq n : bvce_S(n+c-1)[c+i-1] = 0$$

Now, for any given cycle c we can easily model a bitvector of signals sent to a register for any given hardware cycle $i \leq c$, but cycle c **must be provided** before. That is, using Theorem 1 for two different cycles c_1 and c_2 we will parameterize the function $bvin_S$ (and $bvce_S$ analogously) with cycles $c_1 + n$ and $c_2 + n$ which gives us bitvectors of all bits sent as input to S for all cycles up to cycles $c_1 + n$ in case of $bvin_S(c_1 + n)$, and $c_2 + n$ in case of $bvin_S(c_2 + n)$.

Consequently, we also have to fix premise (a). The original version of this premise was defined for *any* bitvector and did not restrict the preimage of the conversion function to only edges smaller than the bitvector size.

Thus, the original version of premise (a) has required that the value $conv(bvin_S)(t)$ has to be defined for all times t for the fixed bitvector $bvin_S$, even if $t \in [e(i)-ts, e(i)+th]$ and the length of $bvin_S$ is smaller than i.[3] That is, the abstraction would only work if we assume that $bvin_S$ will be substituted by an infinite bitvector and, hence, can provide a corresponding bit for any clock edge i. However, as mentioned before, this is not possible in the Isabelle language. We improve the premise (a) by introducing the following global Assumption, which will be used together with applying of improved version Theorem 1.

Assumption 3 (Analog to Digital Conversion)**.** *Let $bv_{P,u} \in \mathbb{B}^n$ be a bitvector of size n, where every bit $bv_{P,u}[i]$ with $i < n$ is sent as input signal to register P of clock domain u in cycle $i < n$. Then there exists a conversion function $conv(bv_{P,u})$ which converts every bit $bv_{P,u}[i]$ to an analog value around clock edge $i + 1$:*

$$\forall i < n : \forall t \in [e_u(i+1) - ts : e_u(i+1) + th] : conv(bv_{P,u})(t) = bv_{P,u}[i]$$

Now, we have to adjust the proof of Theorem 1. In the proof we have to assign to $In_{S,v}(t)$ and $ce_{S,v}(t)$ for $t \in [e_u(i) - ts : e_u(i) + th]$ values derived from the corresponding bitvectors. The main difficulty here is the fact, that we have to fix the original proof, which makes use of Lemma 8. To do so, we have to fix the instantiations of analog signals $ce_{S,v}$ and $In_{S,v}$. Originally, they were substituted by function

$$conv(bvce_S) : \mathbb{R} \to \{0, 1, \Omega\}$$

and, analogously $conv(bvin_S)$. Hence, the conversion function $conv$ was used with some fixed bitvector. But besides some bitvector bv, $conv$ takes only one argument: the real valued time t around some clock edge i, and returns the bit, corresponding to $bv[i - 1]$. However, we have to replace a fixed bitvector bv with a function generating a bitvector for some given cycle. Hence, to fix the instantiation we have to parameterize the $conv$ function with the result of the bitvector generation function $bvin_S$ parameterized with a cycle (which we do not know), corresponding to the given real time t. We could do this, if we provide a function, which gives us the current hardware cycle of clock domain u at any given real-valued time.

This raises the question – for any given real time t, which cycle we should consider as corresponding to t? Since in the end, we are interested in local cycles of an ECU only in connection with its analog output, basing on the definition of the detailed register model, we say, that if t lies in the intervall $(e_u(i) + tp_{min} : e_u(i+1) + tp_{min}]$, then it corresponds to cycle i of clock domain u.

[3] In Isabelle, every list is empty or has a fixed length.

Definition 9 (Real Time to ECU Time)**.** *Let u be a clock domain, and let t be a real-valued global time. Then, we compute the i'th local cycle of clock domain u as follows:*

$$\rho_u(t) = \left\lceil \frac{t - \gamma_u - tp_{min}}{\tau_u} \right\rceil - 1$$

Lemma 9.

$$t \in (e_u(\rho_u(t)) + tp_{min} : e_u(\rho_u(t) + 1) + tp_{min}]$$

Proof. We show that t lies within the boundaries of the interval.
Lower bound.

$$\begin{aligned}
& & e_u(\rho_u(t)) + tp_{min} \\
\text{(Definition 2)} & = & \gamma_u + \rho_u(t) \cdot \tau_u + tp_{min} \\
& < & t \\
& \equiv & \rho_u(t) < \frac{t - \gamma_u - tp_{min}}{\tau_u} \\
\text{(Definition 9)} & \equiv & \left\lceil \frac{t - \gamma_u - tp_{min}}{\tau_u} \right\rceil - 1 < \frac{t - \gamma_u - tp_{min}}{\tau_u}
\end{aligned}$$

Upper bound.

$$\begin{aligned}
& & t \leq e_u(\rho_u(t) + 1) + tp_{min} \\
\text{(Definition 2)} & \equiv & t \leq \gamma_u + \rho_u(t) \cdot \tau_u + \tau_u + tp_{min} \\
& \equiv & \frac{t - \gamma_u - tp_{min}}{\tau_u} - 1 \leq \rho_u(t) + 1 \\
\text{(Definition 2)} & \equiv & \frac{t - \gamma_u - tp_{min}}{\tau_u} - 1 \leq \left\lceil \frac{t - \gamma_u - tp_{min}}{\tau_u} \right\rceil
\end{aligned}$$

□

We change the instantiation as follows:

$$\forall t \in [e_v(i) - ts : e_v(i) + th] : \quad \begin{aligned} In_{S,v}(t) &= conv(bvin_S(\rho_v(t)))(t) \wedge \\ ce_{S,v}(t) &= conv(bvce_S(\rho_v(t)))(t) \end{aligned}$$

We finish the correction of Theorem 1 by changing the statement to:

$$\forall i \in [0 : m] : h_u.R^{\xi + \sigma_{u,v}(c) + i} = bvin_S(c-1)[c-1]$$

Now, Theorem 1 is ready for use in a bus network with arbitrary number of ECUs and during the entire communication under the assumption, that at any transmission time, only one ECU is sending, s.t. we can abstract the bus connection between the sender and every receiver to a direct connection.

The presented flaw of Theorem 1 remained undiscovered because the bitvectors $bvin_S$ and $bvce_S$ were never instantiated with concrete digital signals, but *assumed* to coincide with certain digital signals of a concrete hardware for *all* hardware cycles. Such an assumption cannot be resolved simply because an infinite bitvector cannot be expressed in Isabelle. Hence, in case of a bus architecture with multiple senders this would lead to similar assumptions in the top-level theorem for every sending bus controller. However, the presented correction allowed to get rid of this assumptions completely.

We formulate the correct version of Theorem 1 as a new theorem, which will be used later. Note that we skip the original premise (a) due to the newly introduced global Assumption 3 which will be assumed in all further proofs implicitly.

Theorem 2. *Fix the same variables as in Lemma 8 and additionally:*

- $bvce_S$ *as a function producing a bitvector, s.t.* $bvce_S(n)[i]$ *with* $i \leq n$ *is the bit sent as clock enable signal to* S *in cycle* i*;*
- $bvin_S$ *as a function producing a bitvector, s.t.* $bvin_S(n)[i]$ *with* $i \leq n$ *is the bit sent as input signal to* S *in cycle* i*;*

Premises:

(a) The bit sent as clock enable signal to S *in cycle* c *is a '1'; in the subsequent* n *cycles it is a '0':*

$$bvce_S(c-1)[c-1] = 1 \ \wedge \ \forall i \leq n : bvce_S(n+c-1)[c+i-1] = 0$$

(b) The clock enable signal of R *is always active:*

$$\forall i, t \in [e_u(i) - ts : e_u(i) + th] : ce_{R,u}(t) = 1$$

(c) The input signal of R *is the output signal of* S *during* $n+1$ *local cycles of clock domain of* S*:*

$$\forall t \in (e_v(c) + tp_{min} : e_v(c+n) + tp_{min}] : In_{R,u}(t) = Out_{S,v}(t)$$

Then, the content of digital register R *is equal to the digital input of* S *during* $m+1$ *cycles of clock domain of register* R*:*

$$\forall i \in [0:m] : h_u.R^{\xi+\sigma_{u,v}(c)+i} = bvin_S(c-1)[c-1]$$

These changes led to series of changes in Lemma 10 which is based on the low-level transmission correctness and will be presented in the next chapter.

4.4 Extension to Bus

In a model, where several ECUs are communicating with each other, we need one analog signal bus, which represents the real bus wire and has the same functionality. We model the outputs of send registers S_v facing the bus as open collector output. In this case the bus computes the logical 'and' of the analog output signals put on the bus:

$$\forall t : bus(t) = \bigwedge_v Out_{S,v}(t)$$

The intended use of the bus is to simulate for all slots s the direct connection of a sender register $S_{send(s)}$ to every receive registers R_u on the bus during the transmission window $W(s)$ of slot s, because this permits to apply results about clock domain crossing bit transmission for pairs of directly connected senders and receivers. Obviously, this is achieved by keeping $Out_{S,u}(t) = 1$ for all t in the transmission window and all $u \neq send(s)$.

Finally, we connect the output of the bus with the input of every receive register R_u:

$$\forall t : In_{R,u}(t) = bus(t) \tag{4.24}$$

In the bus controller implementation, both receive registers R and $\hat{R}$ are always clocked. Thus, by definition of DRM together with Equation 4.1, which describes how we model the digital content of the first receive register R, we get for any ECU_u:

$$\begin{aligned} (\text{Equation 4.1}) \quad h_u^i.R &= dig(Out_{R,u}, e_u(i+1)) \\ (\text{Definition 6}, \forall t : ce_{R,u}(t) = 1) &= dig(In_{R,u}, e_u(i)) \\ (\text{Equation 4.24}) &= dig(bus, e_u(i)) \end{aligned}$$

Where h_u^i is to be substituted by the hardware configuration of ECU_u in cycle i.

CHAPTER

FIVE

CORRECTNESS CRITERIA OF MESSAGE TRANSMISSION

As mentioned in Chapter 1, in contrast to the related work, we not only concentrate on the correctness of one particular protocol or property of a time-triggered system, but we rather try to identify all properties and dependencies one has to resolve, to provide one single correctness statement about the crucial property of a TDMA-based time-triggered system: the *message exchange*.

In this short chapter we sketch the correctness statement for the message transmission in the studied bus architecture in Section 5.1. We decompose the final theorem into three milestones. Section 5.2 describes the fundamental property we need – the low-level communication among different clock domains. In Section 5.3 we explain the problem of bus contention control which is provided by the verification of two protocols circularly depending on each other. Finally, in Section 5.4 we describe the last milestone, the correctness of message protocol implementation.

5.1 Correctness Statement

On the top-level, each ECU_i consists of two blocks: the processor p_i and the automotive bus controller abc_i. On the bus controller of each ECU_i pairs of send and receive buffers $ECU_i.sb(j)$ and $ECU_i.rb(j)$ with $j \in \{0, 1\}$ serve both for the local communication between processors and their local bus interface and for communication between bus controllers over the bus. Buffers indexed by '$(s+1) \bmod 2$' face the processor in slot s, buffers indexed by '$s \bmod 2$' face the bus. Thus, the work of every ECU consists of two tasks:

- Local computation. During slot s each processor can access buffers $sb((s+1) \bmod 2)$ and $rb((s+1) \bmod 2)$ of its bus interface via memory mapped I/O for local computation. We will not consider local computation in this thesis.
- Message broadcast. The scheduling function $send$ specifies for each slot s the ECU $ECU_{send(s)}$ whose bus facing send buffer $sb(s \bmod 2)$ will broadcast to the bus facing receive buffers $rb(s \bmod 2)$. Observe that in the following slot $(s+1) \bmod \mathcal{N}$

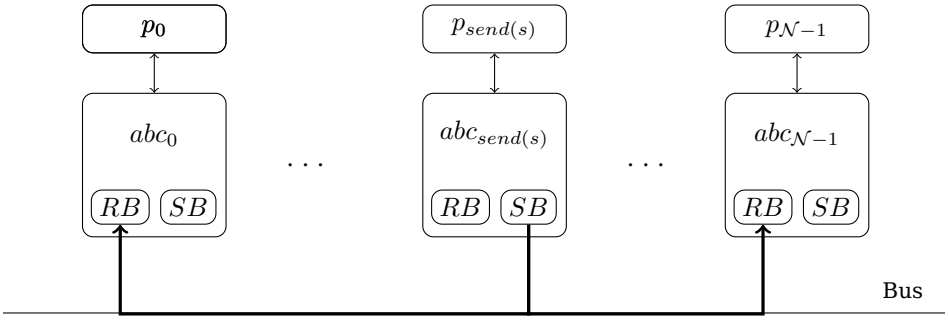


Figure 5.1: Schematic Message Transmission in Slot s

these receive buffers face the processors. This allows to overlap local computation with message broadcast. The message broadcast is schematically depicted in Figure 5.1.

Let ECU_u^j denote the state of ECU_u at hardware cycle j. Let $Slot(r,s)$ denote the slot s of round r with obvious definition of the next slot function:

$$Slot(r,s)+1 = \begin{cases} Slot(r,s+1) & : \quad s < \mathcal{N} \\ Slot(r+1,0) & : \quad s = \mathcal{N} \end{cases}$$

Moreover, let $hcy_u(Slot(r,s))$ be the first hardware cycle of $Slot(r,s)$ on ECU_u. The essential correctness statement of the bus interfaces whose formal proof will be presented in the following chapters is then a very simple and clean statement about message transfer:

$$\forall u, r, s \quad : \quad ECU_{send(s)}^{hcy_{send(s)}(Slot(r,s))}.sb(s \bmod 2) = ECU_u^{hcy_u(Slot(r,s)+1)}.rb((s+1) \bmod 2)$$

That is, we show, that the content of send buffer with index $s \bmod 2$ will be in the next slot in the receive buffer exposed to the processor. Note that an interrupt signal will notify the processor at the end of every slot s, indicating that the accessible receive buffer contains the message transmitted by the sender of slot s. Additionally we show, that in the next slot the content of this receive buffer will not change during the entire slot:

$$\forall x \in [hcy_u(Slot(r,s)+1) : hcy_u(Slot(r,s)+1)+T] :$$
$$ECU_u^{hcy_u(Slot(r,s)+1)}.rb((s+1) \bmod 2) = ECU_u^x.rb((s+1) \bmod 2)$$

A hardware realization of such a bus interface has obviously to deal with the following three problems [PM11].

5.2 Low-Level Transmission Correctness

To establish a TDMA-based communication scheme as described in Chapter 3, we need to ensure that in the beginning of every round a synchronization signal can be transmitted from the master to all slaves.

This milestone is covered by the improved Theorem 2 presented in Chapter 4. Theorem 2 shows that at least one digital bit can be transferred from the sender ECU to any

directly connected receiver ECU. This result abstracts the bus assuming that the analog output of the sender is directly connected to the analog input of the receiver. The correctness of the transmission of one single bit is enough to verify that a synchronization can be established if the bus contains an idle value.

5.3 Bus Control Correctness

At the start of each round ECUs exchange *synchronization messages* in order to synchronize clocks according to some protocol. Non-trivial protocols based on Byzantine agreement are used if one wants to provide fault tolerance against failures of some ECUs. Without fault tolerance a single synchronization message broadcast from a master ECU suffices at the start of each round.

Note however that synchronization messages need to be transferred. The natural vehicle for this transfer is the bus. Obviously, to show transmission correctness of a message in slot s during the transmission window $W(s)$, we not only need to ensure a correct schedule execution by every ECU and correctness of the message protocol implementation on sender and receiver. We also have to show, that during $W(s)$ all non-sending ECUs are producing an idle bus value, ensuring a collision-free message transfer by the sender $ECU_{send(s)}$:

$$\forall t \in W(s) : bus(t) = Out_{S,send(s)}$$

While this lemma is trivial, showing the hypothesis requires not only to show that the S_u have constant value 1 in the digital model for $u \neq send(s)$. One has also to establish the absence of spikes by showing (in the digital model) that clocking of these registers is disabled in the transmission window, which depends on the correct schedule execution.

Moreover, most importantly, the correct schedule execution in round r hinges on the correct synchronization at the beginning of round r, which in turn depends on the absence of bus contention during the time of synchronization, and this again depends on correct schedule execution in round $r - 1$.

Thus, clock synchronization hinges on message transfer (at least for the synchronization messages), message transfer on bus contention control and bus contention control on clock synchronization. Any theorem stating in isolated form the correctness of clock synchronization, bus contention control or message transfer alone must use hypotheses which break this cycle in one way or the other. If the theorem is to be used as part of an overall correctness proof, then one must be able to discharge these hypotheses in the induction step of a proof arguing simultaneously about clock synchronization, bus contention and message transfer. A paper and pencil proof of this nature can be found in [KP07].

That is, we need to argue inductively simultaneously about the correct schedule execution by all ECUs in a round, and about the subsequent correct bus value until the next synchronization. The corresponding theorem and its proof are presented in Chapter 7.

A bus architecture with a fixed TDMA-based schedule with bus contention control – verified this way – provides us with a bus framework for any message transmission protocol. That is, the provided bus controller implementation can be instantiated with an arbitrary message protocol and the bus contention control property would not be affected. The message exchange correctness of a *new* protocol would only require to verify the hardware executing the new protocol.

5.4 High-Level Transmission Correctness

Having the bus contention under control, we know, that during the entire transmission window of every slot s the bus value is equal to the output of the sending ECU. And since the bus value is sampled by every ECU during a transmission window $W(s)$, it remains to show, that the content of sending buffer $ECU_{send(s)}.sb(s \bmod 2)$ will be correctly encoded and transmitted during the transmission window on the sender side, and that it will be correctly received and decoded on the receiver side, respectively.

The message transmission depends on the low-level transmission of a single bit (Section 5.2) and will be implemented as follows. If a message $m[0 : \ell - 1]$ consisting of ℓ bytes is transmitted, then the sender inserts at the start of each byte $m[i]$ so called *sync edges* into the message. Then, the sender puts each bit of the modified message n times on the bus. The purpose of the sync edges is to permit the receiver a *low-level clock synchronization*. Because sync edges occur at regular intervals, the receiver knows when to expect them in the absence of clock drift. If a sync edge before byte $m[i]$ occurs 1 receiver cycle earlier/later than without clock drift, then the receiver knows that his clock has slipped/advanced against the sender clock and adjusts the cycles when it samples the m hardware bits belonging to the same n copies of a message bit accordingly.

The final statement 4 we aim at is clearly a theorem about message transfer using such a mechanism. Hardware devices performing such a transfer are called *serial interfaces*. Correctness theorems about serial interfaces assume a single sender in one clock domain directly connected by a wire to a single receiver in a second clock domain. The correctness of the serial interfaces used in our case study is presented in Chapter 8.

CHAPTER

SIX

BUS CONTROLLER IMPLEMENTATION

In this chapter we will present and explain the crucial parts of the implementation of the studied bus controller and how we model computations of the bus controller hardware over sequences of cycles. Roughly, the bus controller design of a time-triggered system with a TDMA-based communication can be split into two parts.

The first part of the design are the implementations of the scheduling and synchronization protocols presented in Chapter 3. The correctness and functionality of both protocols have a circular dependency on each other and cannot be discussed in isolation.

The second part of the design are serial interfaces – the implementation of the actual message transmission protocol. Obviously, a choice of a concrete message protocol has no impact on the first two protocols as long as all assumptions about serial interfaces are met, e.g., that the Receive Unit initiates a signal to the Scheduler after the reception of the synchronization message. The correctness of the message protocol will be applied after the correctness of the first part of the design is established.

Moreover, the fully digital computation model of the presented hardware can be extended with the DRM model presented in Chapter 4 and combined with the Bus Model from Section 4.4, interconnecting several asynchronously working bus controllers.

This chapter is structured in the following way. In Sections 6.1 and 6.2, we explain how we model the distributed computations of $\mathcal{N}$ ECUs using a local computation model of a single ECU. In Section 6.3, we discribe the top-level view of an ECU, explaining the datapaths between the major building blocks. Section 6.4 contains the implementation of the Scheduler circuit, which maintains states of an ECU realizing the execution of its schedule in each round if a synchronization message was sampled. In Section 6.5, we explain the used message protocol and show its hardware realization. Finally, in Section 6.6 we shortly describe the deploying of the presented implementation of an ECU on three FPGA boards.

6.1 Distributed Computation Model

Our verification is supported by automated tools, which require us to model all hardware computations in terms of traces (Section 2.2). We model parallel computations of $\mathcal{N}$ ECUs as follows.

Assumption 4 ($\mathcal{N}$ Computation Traces)**.** *Assume S_{ECU} is a set of all possible hardware states of an ECU, where each state is defined by the contents of its registers. Let $ecus : \mathbb{N} \to (\mathbb{N} \to S_{ECUs})$ be a function, which returns for every $i < \mathcal{N}$ a trace function corresponding to the i'th ECU. Let $inputs : \mathbb{N} \to (\mathbb{N} \to \mathbb{B}^2)$ be a function, which returns for every $i < \mathcal{N}$ a mapping from local hardware cycle to the ECU inputs in this cycle. We are interested in the two boolean signals, which are part of the input returned by function $inputs$ for some given cycle c: (i) the reset signal: $inputs(i)(c).reset$, (ii) and an input from the bus to the bus controller $inputs(i)(c).bus\_value$. Moreover, we* assume *that $ecus$ returns for every $i < \mathcal{N}$ a valid trace execution function in terms of Section 2.2:*[1]

$$\forall i < \mathcal{N} : inputs(i)(0).reset = 1 \ \wedge$$
$$\forall n \in \mathbb{N} : ecus(i)(n+1) = \delta_{ECU}(ecus(i)(n), inputs(i)(n))$$

Thus, $ecus(i)(j)$ corresponds to the hardware state of ECU_i in cycle j and will be abbreviated by ECU_i^j:

$$ECU_i^j = ecus(i)(j)$$

Furthermore, we abbreviate all signals $inputs(u)(i).x$ denoting the value of the input signal to the bus controller from the bus or the processor of ECU_u in cycle i by x_u^i. Sometimes, we omit the cycle index, writing x_u (or even only x) if we want to refer to the signal in general and not in the context of a concrete cycle or a concrete ECU.

Since we will not present the proofs in a very detailed and technical way, we will not specify the entire implementation of next step function δ_{ECU} limiting it to crucial parts only.

6.2 ECU Computation Model

The schematic top-level implementation of ECU_i is presented in Figure 6.1. The hardware state of ECU_i in cycle j is defined by a tuple of hardware states in cycle j of its processor and a corresponding automotive bus controller:

$$ECU_i^j = (p_i^j, abc_i^j)$$

Before we go into the detailed explanation of the functionality of single components, we concentrate first on the modeling of the computational progress of the entire ECU.

As the name suggests, the bus controller is used by the processor to communicate with other ECUs over the bus. On the ISA-level, the bus is not directly accessible for the processor. Instead, the processor communicates with the bus controller via its internal memory-mapped protocol, whereas the protocol is implemented in the *processor interface* circuit.

The bidirectional communication between the processor p_i and the bus controller abc_i consists of transmissions of the following signals:

[1] W.l.o.g., instead of requiring $ecus(i)(0)$ to be an initial state, we require $input(i)(0)$ to provide a reset signal.

1. in the very first cycle the processor sends (by writing to an I/O-port) a reset signal $reset$ to the bus controller reseting all other registers to necessary idle values;
2. then the processor initializes its bus controller by writing configuration parameters to the configuration registers;
3. after the configuration phase is over, the processor sends (by writing to an I/O-port) a signal $setrd$ (set ready) to the bus controller, indicating, that the bus controller can start its local work;
4. finally, during a system run, the processor recurrently passes to the bus controller the messages to send and reads the received messages.

Thus, potentially, the processor and the bus controller produce some output to each other in every cycle, e.g., if processor reads from a receive buffer.

Let $p\_out(p_i^j)$ denote the output of p_i in cycle j for the bus controller abc_i. Let $abc\_out(abc_i^j)$ denote the output of abc_i in cycle j for the processor p_i. As mentioned before, the hardware state of ECU_i in cycle $j+1$ depends not only on the hardware state of its components in cycle j, but also on the external signals like reset or the signal coming from the bus value bus_value^j, which was sampled during the cycle j:

$$ECU_i^{j+1} = \delta_{ECU}(ECU_i^j, inputs(i)(j))$$

The computation of the next hardware state of ECU_i can be split into two next step functions: the processor next step function δ_p and the bus controller next step function δ_{abc}:

$$\begin{aligned}\delta_{ECU}(ECU_i^j, inputs(i)(j)) \quad = \quad & (\delta_p(p_i^j, inputs(i)(j).reset, abc\_out(abc_i^j)),\\ & \delta_{abc}(abc_i^j, inputs(i)(j).bus\_value, p\_out(p_i^j)))\end{aligned}$$

Thus, we separately compute the next hardware state of the processor in dependence of the reset signal and the bus controller's output, and the next hardware state of the bus controller in dependence of the processor's output and the value, sampled from the bus.

The input from the bus bus_value will be computed as follows. From Section 4.4 we know, that in a digital bus model we would compute:

$$abc_u^i.R = bus\_value_u^{i-1} = dig(bus(e_u(i)))$$

Hence:

$$bus\_value_u^i = dig(bus, e_u(i+1)) \tag{6.1}$$

Since the studied bus controller should not depend on the architecture of its host processor, we do not specify a concrete processor state p_i, the computation of the function $inputs$ and the implementation of its next step function δ_p.

We assume, that the processor initiates a reset signal, writes correct configuration parameters and sends a signal $p\_out(p_i^j).setrd$ (we abbreviate this signal by $setrd_i^j$ from now on) to its bus controller afterwards. Then we are able to verify, that the bus controller i is able to broadcast the content of its send buffer to the receive buffers of all other ECUs in every slot s of all rounds r with $send(s) = i$. The argumentation about the processor's correct filling of the send buffers or reading of the receive buffers has no impact on the correctness of the message exchange executed by the bus controller. Thus, in this thesis we only specify a hardware implementation of the bus controller, which comprises a hardware state and the next step function δ_{abc}.

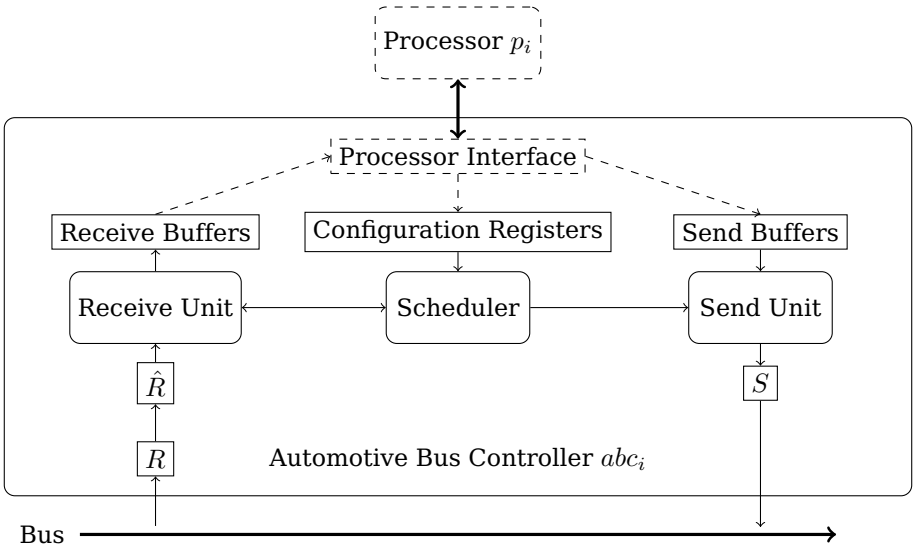


Figure 6.1: Top-level Datapaths of ECU_i

Assumption 5. *($reset$ and $setrd$ signals) Let ECU_i be an arbitrary ECU. Then, the $reset_i$ signal will be activated in cycle 0 only; the $setrd_i$ signal will be activated eventually.*

$$reset_i^0 = 1 \ \wedge \ (\forall x > 0 : reset_i^x = 0) \ \wedge \ \exists j > 0 : setrd_i^j = 1$$

6.3 ECU Datapaths

The dashed components in Figure 6.1 represent abstracted components of the implementation. The implementation of the *Processor Interface* module depends on the protocol used by the processor and initiates only reads and writes to the RAMs and registers located in the send buffers, receive buffers and configuration registers. As in the case of the processor, a particular implementation of this module is irrelevant for the communication protocol, under the assumption, that all configuration registers will be filled with valid parameters until the end of the configuration phase.

The Scheduler is the main control block of the bus controller and consists of a control automaton, slot and cycle counters. It computes the internal state of the controller. The Scheduler uses configuration register values, which specify the number of cycles for different phases in the schedule of an ECU. Moreover, the Scheduler communicates with the serial interfaces of the controller: Send and Receive Units.

If the ECU enters a slot, where it acts as a sender, the Scheduler activates the Send Unit. The Send Unit reads out messages from one of two send buffers, encodes them according to the chosen message protocol as described in Section 5.4. That is, the sender wraps the message with sync edges and puts each bit of the message for n cycles into the send register S. As mentioned before, the sync edges are necessary to allow the low-level synchronization of serial interfaces. The replication of every bit on the sender side and a majority voting on the receiver side makes the message protocol fault-tolerant against random (but bounded) signal distortions leading to bit flips during the low-level transmission. After the entire message was transmitted, the Send Unit returns to its idle state and remains there as long as the ECU acts as a receiver.

The Receive Unit is responsible for receiving messages from the bus. After every sampled bit goes through registers R and $\hat{R}$, the Receive Unit first eliminates the redundancy, built in by the sender, using a majority voter. Besides this, it synchronizes its internal automaton to the protocol flow after each sampled byte of the transmitted message by recognizing the sync edges. Then, every sampled byte is moved into the receive buffer to be read out by the processor later.

The communication of the Scheduler with the Receive Unit is bidirectional. In the beginning of a new round both the Scheduler and the Receive Unit are looping in an idle state. The Receive Unit stays idle until it notices an active signal on the previously idle bus. This signal comes with the very first bit of the very first message of a round sent by the master, which simultaneously serves as the synchronization message. The Receive Unit sends then a signal to the Scheduler indicating that a synchronization signal was received. Subsequently, the Scheduler starts the execution of a round-based schedule consisting of $\mathcal{N}$ slots. At the end of each slot, the Scheduler resets the Receive Unit by setting its control automaton to an idle state. This ensures that the reception of the next message will not be missed by the Receive Unit. In the remaining part of this chapter we will present a schematic implementation of all non-dashed units from Figure 6.1.

Conventions Besides this, we will introduce predicates, indicating certain hardware states, which we will argue about later. For convenience, we will refer to the bus controller of ECU_u by abc_u. We will refer to register x of controller abc_u in cycle i by $abc_u^i.x$ or by x_u^i. We will omit the cycle index i or the ECU index u if the context of the cycle or of the ECU is irrelevant or clear. Moreover, for simplicity, in all figures presenting parts of the bus controller implementation or control automaton, we will omit the depicting and effects (in case of automaton) of the $reset$ signal, since this signal is supposed to stay inactive during an ordinary system run. Note, that in full implementation, every crucial register (like receive registers) will be set to idle values if $reset$ signal gets active.

6.4 Scheduling And Clock Synchronization Protocols Implementation

The implementation of the bus controller consists of a Scheduler, which is fully compatible with the TDMA-based communication presented in Chapter 3. Thus, the Scheduler module of the bus controller is the actual implementation of the Scheduling and the Clock Synchronization protocols.

6.4.1 Configuration Registers Module

The Configuration Registers module is depicted in Figure 6.2. The module has three inputs: (1) write enable signal $crwe$, (2) unary coded write address $crwa$ and (3) and data input $crdin$. These inputs are used in an obvious way to update the register contents with the following bitvectors:

1. $ns \in [0:63]$ represents the number of slots in a round, which is the number $\mathcal{N}$ from Chapter 3;

2. $l \in [0:1023]$ represents the message length in bytes;

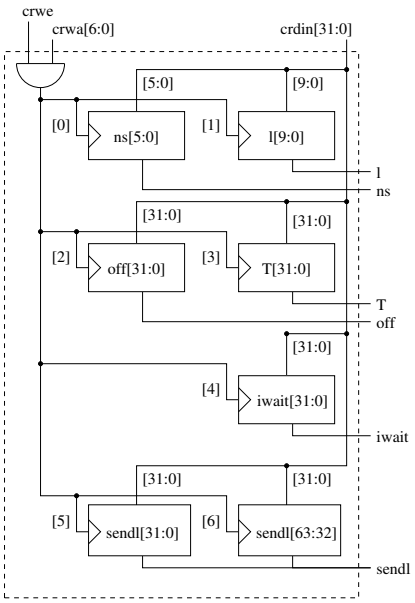


Figure 6.2: Configuration Registers

3. $\mathit{off} \in [0 : 2^{32} - 1]$ represents the length (in cycles) of the offset at the start and at the end of every slot;
4. $T \in [0 : 2^{32} - 1]$ represents the length of a regular slot in cycles;
5. $iwait \in [0 : 2^{32} - 1]$ represents the amount of cycles, which the master ECU has to wait in state `iwait`;
6. $sendl_i \in \mathbb{B}^{64}$ is a bitvector containing a unary mask denoting for a slot $s < ns$ that ECU_i is a sender in slot s if and only if $sendl_i[s] = 1$.[2] Note that we check whether an ECU is the master by evaluating $sendl_i[0] = 1$.

The configuration registers will be updated by Processor Interface module in the configuration phase and never changed after this. In the following proofs, we will refer to each register x by $abc.CR.x$.

Note that all registers except for two registers containing the $sendl$ bitvector will be initialized with the same parameters. The $sendl$ bitvector will be initialized on all ECUs such that in every slot exactly one ECU is sending.

Moreover note that since all counters start counting with 0, all registers containing bitvectors interpreted as number values (ns, l, off, etc.) actually contain the corresponding numbers decremented by one.

Note that our bus controller design [Kna08] assumes, that there are exactly ns slots in a round and ns ECUs in the bus network. Hence, in every slot exactly one ECU is sending and every ECU sends once in each round. From now on, we will use number ns instead of $\mathcal{N}$.

In the remaining chapters of this thesis we will refer to the values of the presented registers as natural numbers, skipping technical details of converting natural numbers to bitvectors and vice versa.

[2] For simplicity, we consider one register $abc.CR.sendl \in \mathbb{B}^{64}$ instead of two 32-bit parts as in the actual implementation.

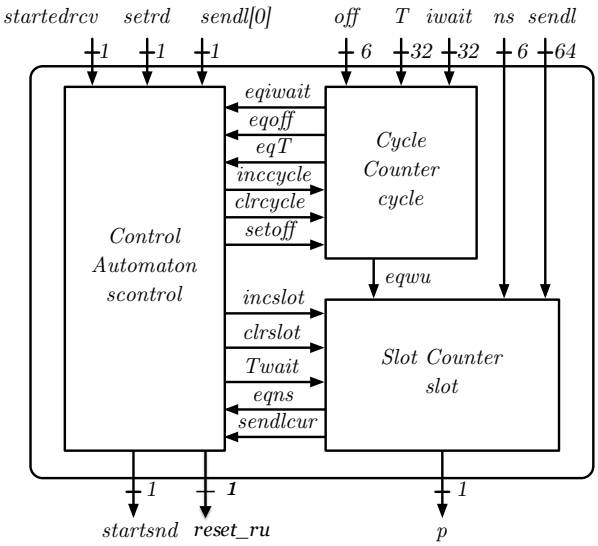


Figure 6.3: Datapaths of the Scheduler Module [Böh07]

Assumption 6 ($sendl$ Register Value)**.** *The register $abc_i.sendl$ will be initialized with a unary bitvector $sendl_i$, which is different for all ECUs. The master (ECU_0) gets the bitvector $0^{63}1$, thus it's sending in the first slot.*

$$\begin{aligned}\forall i < ns: \quad & sendl_i \text{ is a unary bitvector of length } ns \ \wedge \ sendl_0[0] = 1 \ \wedge \\ & \forall j \in [0:63] : sendl_i[j] = 1 \rightarrow \ \forall k < ns : k \neq i \rightarrow sendl_k[j] = 0\end{aligned}$$

6.4.2 Scheduler Module

All ECUs share the same Scheduler module. In Figure 6.3 the top-level datapaths of the Scheduler module are presented. It consists of thee submodules: (1) Cycle Counter module $cycle$, (2) Slot Counter module $slot$ and (3) Control Automaton $scontrol$.

Cycle Counter Module

The cycle counter module counts the local hardware cycles and is implemented as depicted in Figure 6.4(a). The module has a bidirectional communication with the Scheduler Automaton. The cycle counter module signals if the counter $abc.cycle$ value reaches one of the configuration register values: off, T or $iwait$. In every of these three cases it initiates a corresponding signal to the control Automaton of the Scheduler:

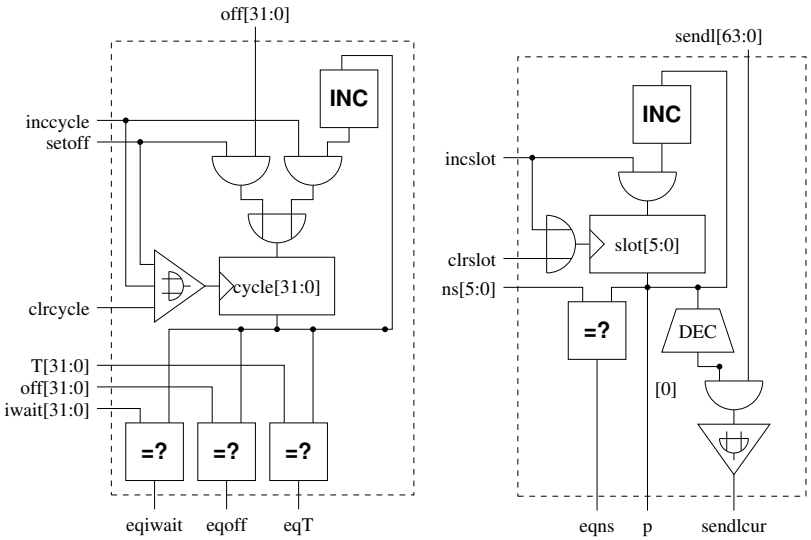
- signal $eqiwait^i$ is active if the counter value is equal to input value $iwait$:[3]

$$eqiwait^i \Leftrightarrow abc^i.cycle = iwait - 1$$

- signals $eqoff^i$ and eqT^i are defined in the same manner.

Moreover, the Control Automaton of the Scheduler controls the cycle counter module by three signals.

[3] Remember, all counters start counting with 0.



(a) Cycle Counter Implementation (b) Slot Counter Implementation

Figure 6.4: Implementation of the Slot Counter and the Cycle Counter Modules

- $inccycle^i$ signal increments the cycle counter by one in the next cycle:

$$inccycle^i \Rightarrow abc.cycle^{i+1} = abc^i.cycle + 1$$

- $clrcycle^i$ sets the cycle counter to zero:

$$clrcycle^i \Rightarrow abc^{i+1}.cycle = 0$$

- $setoff^i$ sets the cycle counter to off:

$$setoff^i \Rightarrow abc^{i+1}.cycle = off$$

Slot Counter Module

The Slot Counter module is implemented as depicted in Figure 6.4(b) and has the following functionality. It outputs three signals:

1. $eqns^i$ which is active if the slot counter is equal to the configuration value ns:

$$eqns^i \Leftrightarrow abc^i.slot = ns - 1$$

2. the parity bit p is just the least significant bit (index 0) of the slot counter value, which is equivalent to the stored value $\bmod 2$:

$$p^i = abc^i.slot[0]$$

3. $sendlcur^i$ indicates whether the bus controller is acting as a sender in the current slot:

$$sendlcur^i = sendl[\langle abc^i.slot \rangle]$$

The Slot Counter module is controlled by the Control Automaton with two input signals:

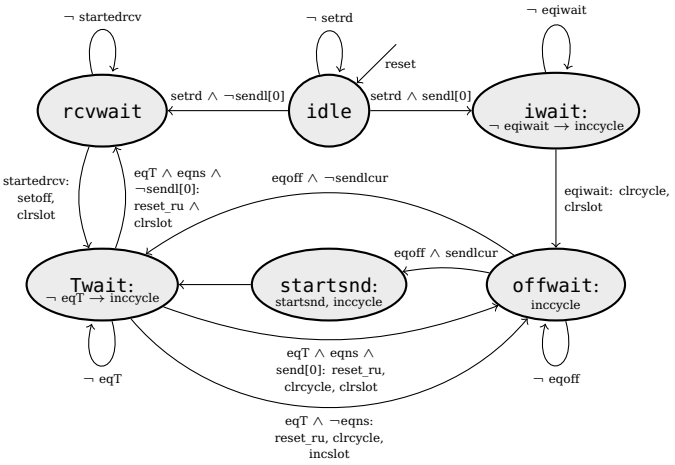


Figure 6.5: Scheduler Automaton

1. $incslot^i$ causes an incrementation of the counter in the next cycle:

$$incslot^i \Rightarrow abc^{i+1}.slot = abc^i.slot + 1$$

2. $\neg inclot^i \wedge clrslot^i$ sets the slot counter to zero:

$$\neg inclot^i \wedge clrslot^i \Rightarrow abc^{i+1}.slot = 0$$

Control Automaton

The control Automaton of the Scheduler computes the next state according to the current state and its input signals. We will refer to the state of the Scheduler Control Automaton of ECU_i in cycle c by $abc_i^c.state$. The signals $eqiwait, eqoff, eqT, eqns$ and $sendlcur$ are coming from the Slot and Cycle Counters. The signal $startedrcv$ will be sent by the Receive Unit at the reception of a synchronization signal. The signal $setrd$ will be initiated by the Processor Interface at the end of the configuration phase. Finally, the Automaton reads the least significant bit $abc.sendl[0]$ of the configuration register $abc.sendl$, which indicates whether the ECU is the master or not. The schematic representation of the Scheduler control Automaton is depicted in Figure 6.5, where every transition between two states is an edge annotated by labels of the form:

$$a_0 \wedge \ldots \wedge a_{n-1} : b_0, \ldots, b_{n-1}$$

or simply

$$a_0 \wedge \ldots \wedge a_{n-1}$$

Here, the a's are predicates whose conjunction causes the transitions and b's are predicates getting active at the cycle of the transition.

By our assumption, every ECU will be reset in its very first cycle. After this initial reset signal, all ECUs are in state `idle`. They remain in this state during the entire configuration phase ($\neg setrd = 1$). After all configuration registers are written, the operating system running on the processor initiates signal $setrd$, which forces all ECUs

to change either to state `iwait` if the ECU was configured as master ($sendl[0] = 1$), or to state `rcvwait` otherwise. The master stays in state `iwait` for $iwait$ cycles, until the cycle counter reaches this value and the signal $eqiwait$ gets active. Assuming that all ECUs are started roughly at the same time, it should be guaranteed that after the master initialization and additional $iwait$ cycles[4] all slaves are initialized and are in state `rcvwait`. The size of $iwait$ can be estimated by industrial worst case execution time analyzers [FW99].

While slaves are still waiting for a synchronization message in state `rcvwait`, the master starts slot 0 of round 0 by entering state `offwait`. Its slot and cycle counters are set to 0. The master is always scheduled as sender in slot 0 ($send(0) = 0$), since its message serves as a synchronization signal for all slave ECUs. After off cycles in state `offwait`, the master starts broadcasting of its message going through state `startsnd`. The first bit of the message should be the non-idle value '0'. This bit serves as the actual synchronization signal for all slaves, listening to the idle bus containing only '1' (since no one else is sending). Note that at the start of the message broadcast, the cycle counter of the master contains number off.

Slaves get into state `rcvwait` not only after the initialization but also after the end of each round to wait for the next synchronization message. As mentioned before, the signal $rcvwait$ will be sent by the Receive Unit to the Scheduler as soon as it samples the first active signal on a previously idle bus (at the end of a round). When this happens, the Scheduler Automaton switches to state `Twait` and resets the slot counter, since a new round is started. But the cycle counter will be set to off, which is the local time of the master at the moment of the synchronization, thus synchronizing its own local time to it.

In state `Twait` each ECU waits until signal eqT gets active, which happens if the cycle counter reaches number $T - 1$. In state `Twait` every ECU is sending or receiving, depending on its role in the current slot. After the slot ends ($eqT = 1$), a slave ECU ($\neg sendl[0] = 1$) either switches back to `rcvwait` if the maximal number of slots in a round is reached ($eqns = 1$), or to state `offwait` if not all slots of a round are accomplished ($\neg eqns = 1$). Every leaving of state `Twait` denotes the end of a slot. The Scheduler Automaton generates a signal $reset\_ru$ which sets the Receive Unit to its idle state and resets all crucial registers to an idle value. Until that point the message reception *should be* finished on all slaves. Signal $reset\_ru$ is computed by the Scheduler module as follows:

$$reset\_ru^i \equiv abc^i.state = \texttt{Twait} \ \wedge \ abc^i.cycle = T - 1 \tag{6.2}$$

In `offwait`, all ECUs are waiting for off hardware cycles until $eqoff = 1$. This is the waiting time (cf. Section 3.4) at the start of every slot, which is necessary due to clock drift to guarantee that all ECUs have started the current slot. Afterwards, if the ECU is a sender ($sendlcur = 1$) in the current slot, the Scheduler sends $startsnd$ signal to the Send Unit going through state `startsnd`. This signal activates the Send Unit which start a message broadcast. Otherwise ($\neg sendlcur = 1$), it switches directly to state `Twait` again.

The master ECU ($sendl[0] = 1$) acts as a sender in the first slot of each round. Afterwards, it switches as a receiver between states `offwait` and `Twait` during remaining $ns - 1$ slots.

Note that in states `iwait`, `offwait`, `startsnd` and `Twait` the control automaton keeps the signal $inccycle$ active. This causes incrementation of the cycle counter in

[4] Actually, even $iwait + off$ cycles would work.

Sequence	Value	Comment
TSS	01	Transmission Start Sequence
BSS	10	Byte Start Sequence
TES	01	Transmission End Sequence

Table 6.1: Sync Edged of the Message Protocol

all of these states.

6.5 Message Protocol

Before we start with the implementation of serial interfaces, we will first explain the message protocol used in the studied architecture.

Assume, that l is the message length in bytes of the message m, which has to be transmitted over the bus. Since the sender and receivers' clocks are constantly drifting apart, we cannot stream the entire message without resynchronizing the serial interfaces of both parties of the communication protocol. Let function f denote the encoding function, and m_i the i'th byte of m with $0 \leq i \leq l-1$. We encode the message m by inserting at the beginning, at the end and before every byte special bit sequences called sync edges. They are listed in Table 6.1 and are used as follows:

$$f(m) = TSS \circ BSS \circ m_0 \circ BSS \circ m_1 \circ ... \circ BSS \circ m_{l-1} \circ TES$$

Thus, the encoded message starts and ends with bits '01'. Every byte starts with a sequence '10'. Since the message is always transmitted when no other ECU is sending and the bus contains idle value '1', by bus construction, the message reception on every ECU starts with a falling edge, produced by TSS[0] = 0.

The encoded message $f(m)$ contains exactly $4 + 10 \cdot l$ bits: TSS and TES produce 4 bits together and every byte with a corresponding BSS yields 10 bits. However, as mentioned before, along the lines of FlexRay specification, we replicate every bit of the encoded message 8 times on the bus. This provides a fault-tolerance against random but bounded bit jitters during a message transfer. Hence, the transmission length tl can be computed as:

$$tl = 8 \cdot (4 + 10 \cdot l) = 32 + 80 \cdot l$$

However, this is the length of the plain transmission of the encoded message $f(m)$ with every bit replicated 8 times. The entire message transmission from the Send Unit of the sender to the Receive Unit of the receiver lasts $tl + 9$ sender cycles [Böh07]. The additional 9 cycles are caused by local register delays of the serial interfaces and effects of the clock domain crossing signal transmission. The output signal enters the bus 2 cycles after the Send Unit starts the broadcast. The Receive Unit of the receiver will need 6 additional cycles to process the sampled signal. The latter delay is caused by eliminating the 8-bit replication built in by the sender and will be explained later in this chapter. The remaining delay cycle might result on the side of the receiver in case of a not correctly sampled bit at the start of the transmission (see Definition 8).

Thus, the amount of transmission cycles tc needed to transmit an encoded replicated message from the Send Unit of the sender to the Receive Unit of the receiver is computed as follows:

$$tc = tl + 9 = 41 + 80 \cdot l$$

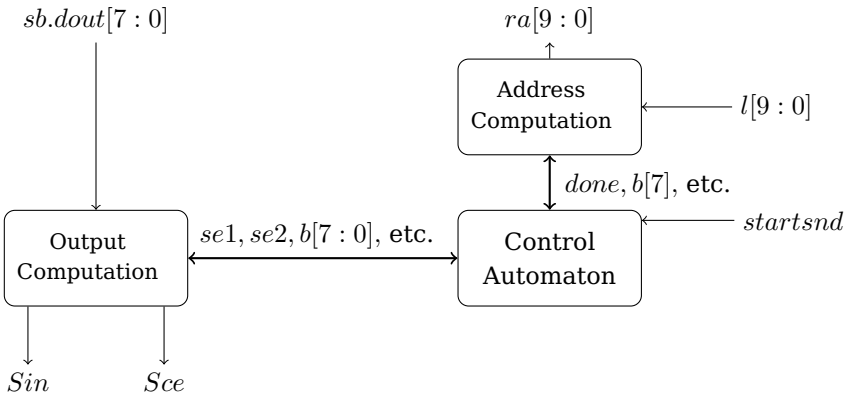


Figure 6.6: Send Unit Datapaths

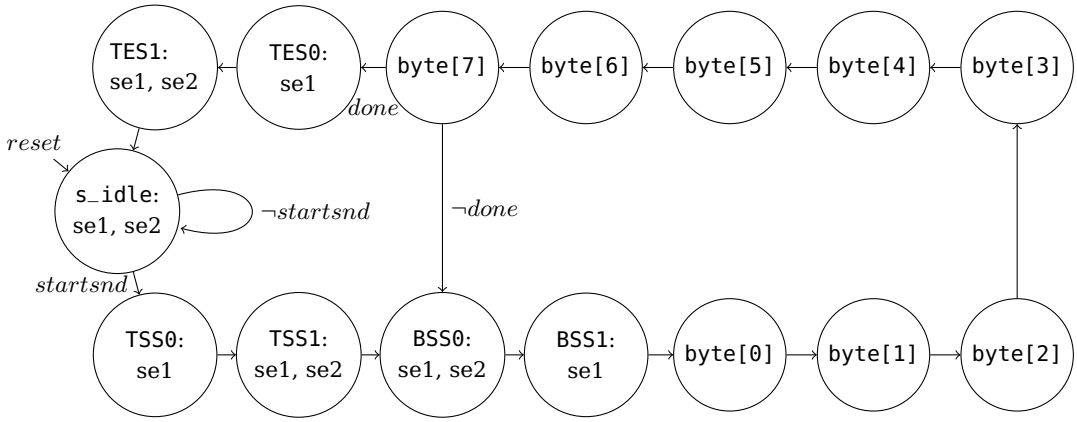


Figure 6.7: Send Unit Control Automaton

In the next two sections we will present the implementation of serial interfaces for the introduced protocol: the send and Receive Units. We will not show the entire implementation, but we will concentrate on the most important parts of their functionality.

6.5.1 Send Unit

The datapaths of the Send Unit are depicted in Figure 6.6. It has an input signal and two input buses, and it has two output signals and an output bus. The Send Unit consists of three submodules: (1) Control Automaton, (2) Address Computation and (3) Output Computation. Input signal $startsnd$ manipulates the Control Automaton. The Address Computation module takes the message length l from the corresponding configuration register and generates memory address ra for the send buffer. The Output Computation module is depicted in Figure 6.8 reads a byte $sb.dout$ returned by the send buffer for the generated address ra. Afterwards, according to the state of the Control Automaton, one of the bits of $sb.dout$ byte, or one of the sync edges of the protocol will be output as Sin, which is the input of the send register S. The clock enable signal Sce of register S gets active every 8 cycles.

The Control Automaton module is the control logic of the Send Unit and is schemati-

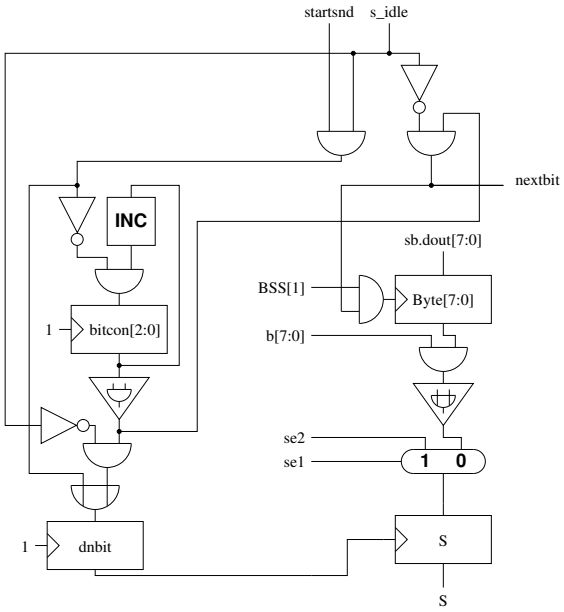


Figure 6.8: Output Computation Module with Send Register S

cally depicted in Figure 6.7. The Automaton implementation is straightforward; it sends to the Output Computation module three signals:

- 1-bit signals $se1, se2$, which are used to produce **s**ync **e**dges of the message protocol;
- a unary bitvector $b[7:0]$ will be produced as output in every state `byte[i]`, such that $b[i] = 1$ (not depicted in Figure 6.7).

According to this information, the Output Computation module either produces one of two bits of a sync edge or choses which bit will be clocked into the send register S as depicted in Figure 6.8. Every transition from one state to another is triggered by output signal $nextbit$ of the Output Computation module every 8 cycles, hence, the Automaton stays in every state for 8 cycles.

After the initial $reset$ activation and before every transmission the Automaton is idle. While signal $startsnd$ remains inactive, the Automaton does not leave its idle state. If the signal gets active, the Send Unit starts with the protocol execution according to Section 6.5. Besides this, the Address Computation module signals to the Control Automaton whether the last byte of the message was sent ($done = 1$). In this case the message transmission will be finished and the transmission end sequence TES will be generated.

In proofs, we will refer to the state of the Send Unit Control Automaton of ECU_i in cycle c by $abc_i^c.sstate$.

6.5.2 Receive Unit

The Receive Unit has a more sophisticated design. As depicted in Figure 6.9 it has three input signals, four output signals and consists of five modules and a shift register $abc.Byte$ of length 8.

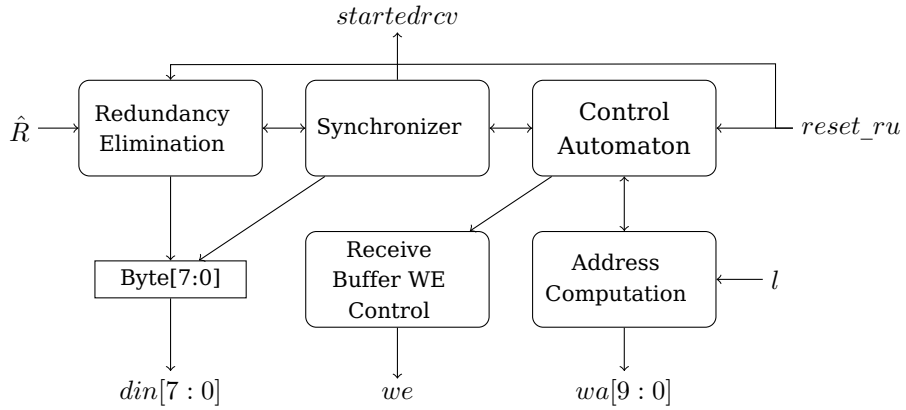


Figure 6.9: Receive Unit Datapaths

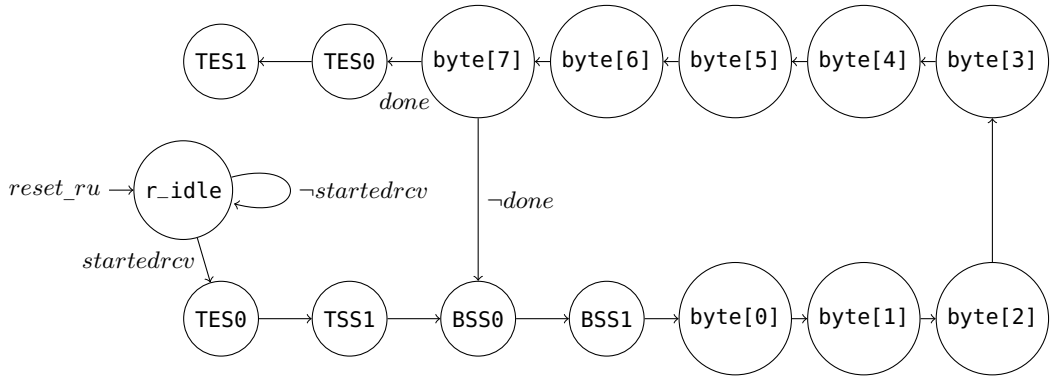
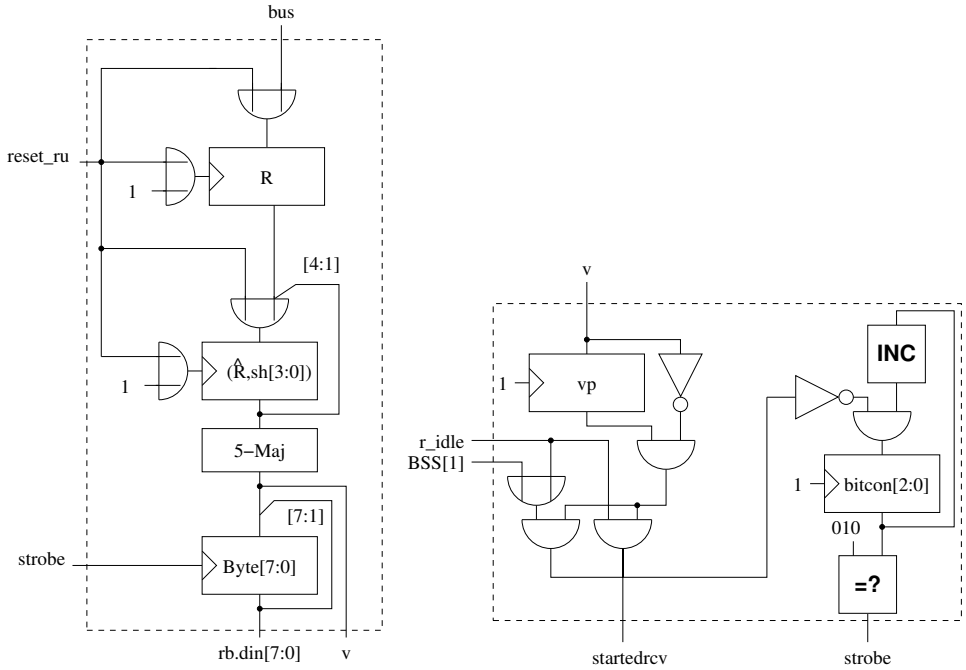


Figure 6.10: Receive Unit Control Automaton

The Address Computation module has the same functionality, as the identically named module of the Send Unit. It takes the length l in bytes of the transmission message and computes addresses for the receive buffer.

As depicted in Figure 6.10, the control Automaton of the Receive Unit has the same states as the control Automaton of the Send Unit and is used to keep track of the protocol run. Every transition will be triggered by the Synchronizer circuit (see Figure 6.11(b)) activating signal $strobe$ every 8 cycles except for one state. As we will describe later, the Synchronizer resets its cycle counter before the reception of every byte in state `BSS[1]`. Note the Automaton does not return into the idle state after the end of a transmission. Instead, it remains non-idle until it will be reset by the Scheduler at the end of a slot ($reset\_ru$ gets active). This makes the entire system more robust, since it prevents the Receive Unit from false start *after* the message transmission in some slot s and *before* the slot s was accomplished on the Scheduler. This could happen, because the Receive Unit is not inevitable aware of the internal state of the Scheduler. In our design, the Receive Unit starts with a message reception as soon as it is idle and notices a bus activity. The message transmission lasts tc cycles, which is a sub-interval of an entire slot (remember, slot length is T with $T = 2 \cdot \mathit{off} + tc$). Thus, by locking the Receive Unit in a non-idle state after the transmission, we only have to argue about the idle bus value beginning from the slot end and not from the transmission end.



(a) Receive Registers, Redundancy Elimination module and the Byte Buffer

(b) Synchronizer Circuit

Figure 6.11: Redundancy Elimination and Synchronizer Circuits

The Redundancy Elimination module takes the output of the second receive register $\hat{R}$ and tries to recognize the bit, whose replication was sent over the bus. The implementation of that module, together with the $Byte$ buffer and both receive registers is depicted in Figure 6.11(a). As described in Section 4.4 the digitized bus output goes directly into the first receive register R. The output of R will be immediately clocked into register $\hat{R}$, and afterwards it will be shifted through the shift register $abc.sh$, which consists of four nested (always clocked) 1-bit registers $abc.sh_i, i < 4$, s.t.:

$$\forall c : abc^{c+1}.sh_0 = abc^c.\hat{R} \wedge \forall i \in [1:3] : abc^{c+1}.sh_i = abc^c.sh_{i-1}$$

Hence, the Receive Unit has a shift register, containing the last five bits sampled from the bus. As depicted in Figure 6.11(a), both receive registers R and $\hat{R}$ as well as the shift register $abc.sh$ will be set to the idle value '1' every time the Receive Unit will be reset by the active $reset\_ru$ signal.

The 5-bit majority voter circuit outputs the most frequently appearing bit v among register $abc.\hat{R}$ and shift register $abc.sh$. The output of the majority voter goes then through the shift register $abc.Byte$. In contrast to the shift register $abc.sh$, the shift register $abc.Byte$ is clocked only if the signal $strobe$ gets active.

The $strobe$ signal is computed by the Synchronizer Circuit depicted in Figure 6.11(b). The Synchronizer Circuit tracks the voted bit v, and counts the number of passed cycles, constantly incrementing the value of a 3-bit register $abc.bitcon$. Signal $strobe$ gets active every time $abc.bitcon$ reaches the value '010', which corresponds to 2. After the value '111' (corresponds to 7), the counter value gets 0. Thus, in most cases, the strobe signal gets active every 8 cycles. However, it will be reset in two cases.

If the voted bit v changes its value from 1 to 0 *and* the Control Automaton of the Receive Unit is idle ($r\_idle = 1$), then a new message reception is started. In this case the Synchronizer also initiates signal $startedrcv$ to the Scheduler module.

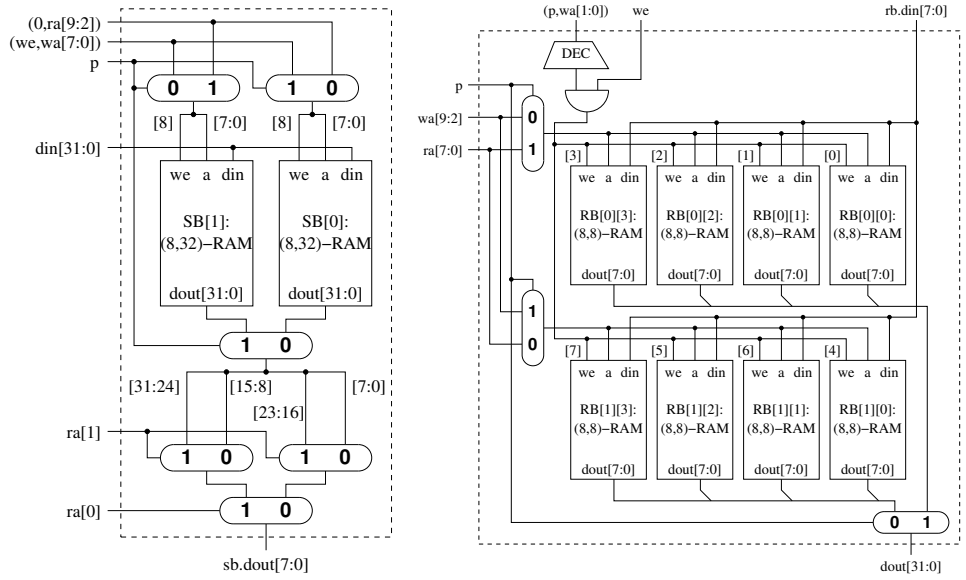
In the second case, if the Receive Unit is about to start the sampling of a new byte (Receive Automaton is in state `BSS[1]`), the cycle counter will be reset to adjust the Receive Unit to the protocol flow. This is necessary, since by Lemma 3 the Receive Unit might have been drifted away from the Send Unit by 1 cycle during the reception of previous byte, which lasts exactly 80 cycles.

Thus, the strobe signal gets active every 8 cycles roughly in the middle of 8 sampled sender bits. Therefore, shift register $abc.Byte$ consists eight values of the most voted bit v, which are exactly the sent bits.

The Receive Buffer WE Control module is responsible for the computation of the **w**rite **e**nable signal of the Receive Buffer. Thus, every time, the output signal we gets active, the output signal $rb.din$ (the content) of the shift register $abc.Byte$ will be written to the Receive Buffer at the address wa. By construction of that module, the we signal gets active every time the Control Automaton leaves state $byte[7]$, denoting the reception of the last bit of the currently transmitted byte.

6.5.3 Send and Receive Buffers

The Send Buffer construction is depicted in Figure 6.12(a).



(a) Send Buffer

(b) Receive Buffer

Figure 6.12: Buffers Construction

It consists of two byte-addressable RAMs $abc.SB[0]$ and $abc.SB[1]$ storing bitvectors of length 32. As inputs it has:

- read address $ra[9 : 0]$ specified by the bus controller to read out byte-wise the message to send;

- write address $wa[7:0]$ and write enable signal we specified by the processor while the buffer is filled with a message to send;
- data input $din[31:0]$ specified by the processor – note the processor writes the messages word-wise[5];
- parity bit p specified by the Scheduler module denoting whether the current slot is odd or even ($p = s \bmod 2$).

The parity bit p routes all write accesses (made by processor only) to the buffer $abc.SB[\neg p]$. All reads (made by bus controller only) are routed to the buffer $abc.SB[p]$. Thus, in every slot, the bus controller and the processor are accessing one of the send buffers exclusively. This allows us to overlap on one ECU the local work of the processor with communication over the bus of the bus controller. The Send Buffer has only one output $sb.dout[7:0]$ which is used by the Send Unit.

The Receive Buffer consists of 8 byte-addressable RAMs, storing bytes. Its design is depicted in Figure 6.12(b). It has following input signals:

- read address $ra[7:0]$ used by the processor to read out received messages word-wise;
- write address $wa[9:0]$ and write enable signal we computed by the Receive Unit at the byte-wise messages reception;
- data input $rb.din[7:0]$ provided by the Receive Unit;
- parity signal analogously to the Send Buffer.

As in the Send Buffer, writes by the Receive Unit and reads by the processor are routed to different RAMs. The single output of the Receive Buffer $dout[31:0]$ is then used by the processor to read out the received messages.

6.6 Deploying on FPGAs

An experimental deploying of the bus controller on three FPGAs (Figure 6.13) with a self-implemented bus was supervised during this work and was reported in [End09, EMST10]. In this work, two different kinds of FPGAs were used: two of three FPGA boards of type Spartan-3 and the third one of type Virtex-2.

Each FPGA board has represented one single ECU. The bus was realized by two ethernet cables, connected to RJ45 registered jacks. The jacks were connected to each board through an IOBUFDS driver interface, provided by the FPGA boards. Two open ends of the bus were closed by terminal resistance ($R_t = 100\Omega$). As a processor a simple version of the VAMP architecture [MP00] was used.

To test the system, a simple assembler program was written, which has configured the FPGAs imitating a simple operating system. The message transmission was tested with help of on-board signal analysis. Besides this, simple signal transfer was realized by switching of eight DIP-switches on the sender FPGA to generate a one byte message, which was displayed on eight LEDs built-in on the receiver FPGAs.

During this work numerous bugs in the implementation of the configuration routine were discovered and fixed.

[5] Here a *word* contains 4 bytes.

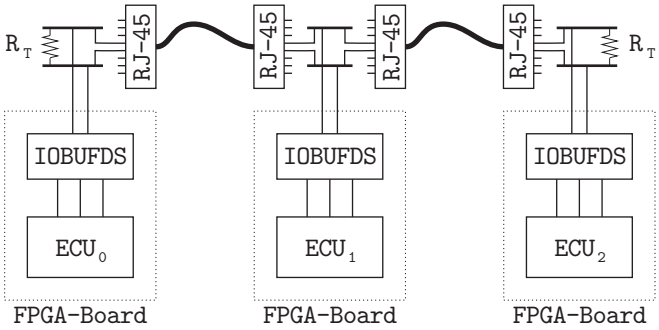


Figure 6.13: Three FPGAs Interconnected by a Bus [End09]

CHAPTER

SEVEN

VERIFICATION OF BUS CONTENTION CONTROL

In this chapter we will establish formally the correctness of the bus contention control during the transmission times of all slots. That is, we will show for all ECUs executing the same schedule, that they *always* agree on a slot s during the corresponding transmission window $W(s)$. Afterwards, we will show, that every non-sending ECU_i with $i \neq send(s)$ produces an idle bus output during $W(s)$. Combining this result with the bus construction, we will conclude that during all slots of all rounds, the bus contains the analog output of the sending ECU:

$$\forall r, s < ns, t \in W(s) : bus(t) = Out_{S,send(s)}(t)$$

Such a proof requires a formal computation model for ns asynchronously working ECUs, which includes a bus model developed in Chapter 4 used for signal exchange among ECUs. Note that to prove the absence of bus contention we initially need only the correctness of signal exchange during the transmission of the synchronization signal from the master to all slaves. For this, Theorem 1 will be applied. The subsequent agreement on the global time progress will then result from the construction of the Scheduler.

Note that all proofs in this chapter are presented in extremely short form. Lots of technical details are hidden.

This chapter is structured as follows. First we start with the description of previous work in Section 7.1. We show which parts were already verified and how they needed to be improved and extended. In Section 7.2 we explain how we extend the semantics used to model computations of single controllers in a post-configuration phase in previous verification results, to a global semantics modeling computation of entire ECUs during their complete sysem run starting from cycle 0. In Section 7.3 we argue about the startup correctness on the master and slaves and explain the main assumption we make on the startup routine. In Sections 7.4 and 7.5 we extend previous proofs from Section 7.1 about the correct schedule execution in one round correctness for all rounds. In case of a slave it requires simultaneous induction about the correct schedule execution and bus control in the end of a round. Finally, in Section 7.6 we show the

correctness of the bus control during every message transmission, using the correct schedule execution for all rounds by the master and all slaves.

7.1 Previous Results and Their Improvements

In this Section we will present the results of Böhm from [Böh07]. In this work Böhm has formally verified that the Scheduler implementation presented in Section 6.4.2 fulfills the timing constraints with respect to a transmission window of a slot. He also has shown a schedule execution within one round. Moreover he has shown the correct transmission of the synchronization signal using Theorem 1.

Informally speaking, Böhm has fixed two bus controller traces: one of them belonging to a master controller and one arbitrary slave. Note that his lemma uses a trace semantics in the sense of Section 2.2 of bus controllers only and not of entire ECUs. Hence, he assumed that all configuration registers of the controllers contain correct values and during the entire system the configuration registers of all hardware states are always stable; moreover, the bus controllers do not receive any reset signals from the input function of the trace semantics. Thus, the configuration phase was abstracted away completely.

Furthermore, he assumed, that the hardware states of the receivers are in a synchronization waiting states (`rcvwait`) with idle Receive Units. He also assumed, that the redundancy elimination shift register of the receives units (Figure 6.11(a)) is filled with idle values, i.e., ones.

On the master side, he assumed that its Scheduler is in state `startsnd`, denoting the start of a synchronization message broadcast. However, for some reason, Böhm did not instantiate the bitvector $bvce_S$ of Theorem 1 with the bit list of clock enable signals derived from the trace semantics of the sender controller. Instead, he left the bitvector as uninterpreted data structure with the assumption about correct values of this bivector. Hence, his original results are based on the assumption, that the clock enable signal of the master controller are computed correctly.

Moreover, Böhm has used the original low-level transmission Theorem 1 and included its broken assumption about the permanent direct connection of the master and two receivers. Hence, all fixes made to Theorem 1 had to be propagated through all results of Böhm.

Note that as in case of low-level transmission, the assumption about the direct connection of the master and receivers abstracts the bus and makes the bus contention problems irrelevant. Hence, he did not need to make *any* assumptions about the *Send Units* of both receivers and did not need to argue about their behaviour, because no notion of bus was used. So he also did not argue about the clock enable signals or analog output of the send registers S of receiver controllers.

Thus, assuming that all the receive bus controllers are in the state of waiting for synchronization, and that the master controller starts the broadcasting, he has shown, that the transmission start sequence ($TSS[0] = 0$) will be put by the Output Computation module of the Send Unit into the send register S for 8 cycles. Since he assumed an idle Receive Unit with the shift register filled with ones, he could easily show the generation of the $startedrcv$ signal triggering the Schedulers of the receivers.

Afterwards, he argued about the correct execution of the schedule for one round consisting of ns slots on all ECUs. This follows from the control automaton and slot / cycle counter constructions. Finally, he has shown that in terms of global time every slot

on every ECU starts before and ends after the message transmission of this slot. This result did not take into account, that the analog output of the send register changes with a two cycle delay with respect to the Scheduler state.

Besides this, all theorems of Böhm contained one too strong assumption. He has *assumed*, that the shift and receive registers of the Receive Unit (register R, $\hat{R}$ and $abc.sh$) are filled with ones in cycle $\xi + \sigma_{u,0}(c+2)$. Thus, he has shown, that the synchronization might be delayed by one cycle depending solely on the value of $\sigma_{u,0}(c+2)$. However, his assumption is not fullfilled if the timing requirements of the first receive register R are not met ($\sigma_{u,0}(c+2) = 1$) *but* the undefined value sampled into R flips to or turns out to be the correct one. In this case the synchronization will not be delayed.

We fix it by defining a synchronization delay cycle in the next definition.

Definition 10 (Synchronization Delay)**.** *Let u and v be two arbitrary ECUs; u acts as a receiver and v as a sender. Let c be the cycle at which ECU_v changes the content of its send register $abc_v.S$ to 0. Let $\xi = cy_{u,v}(c)$ be the next affected cycle of ECU_u. Then $\phi_{u,v}(c)$ denotes a synchronization delay cycle which arises if the timing constraints of the receive register $abc_u.R$ are not met and the sampled undefined value flips to an incorrect value, i.e., to a '1':*

$$\phi_{u,v}(c) \equiv \textbf{\textit{if}}\ \sigma_{u,v}(c) = 1\ \wedge\ abc_u.R^{\xi} = 1\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ 0$$

This flaw was not discovered by Böhm because all his results argue about one round only and he did not provide a proof, that assumptions he fixes at the start of some round r still hold at the start of the next round $r+1$.

7.1.1 Synchronization Correctness

Now we will formally present the improved results of [Böh07]. First we show the lemma stating, that slaves will enter slot 0 after the synchronization. For this we need a formalization of a *slot start*.

Definition 11 (Slot Start)**.** *Let $\Sigma_u(s)^c$ denote the predicate indicating that the hardware state of ECU_u, is at the start of slot s in cycle c. We say that ECU_i is at the start of some slot $s \in [1 : ns-1]$ in cycle c if in that cycle its Scheduler is in state `offwait`, its cycle counter is 0 and its slot counter contains s:*

$$\forall i < ns, s \in [1 : ns-1] : \Sigma_i(s)^c \equiv abc_i^c.state = \texttt{offwait}\ \wedge\ abc_i^c.slot = s\ \wedge\ abc_i^c.cycle = 0$$

The slot start $\Sigma_0(0)^c$ of slot 0 on the master is defined in the same way:

$$\Sigma_0(0)^c \equiv abc_0^c.state = \texttt{offwait}\ \wedge\ abc_0^c.slot = 0\ \wedge\ abc_0^c.cycle = 0$$

Whereas slot 0 starts on a ECU_i ($i \neq 0$) in slot s directly after synchronization, i.e., after leaving the state `rcvwait`. Thus, its Scheduler is in state `Twait`, its slot counter contains 0 and its cycle counter has value off:

$$\forall i \in [1 : ns-1] : \Sigma_i(0)^c \equiv abc_i^c.state = \texttt{Twait}\ \wedge\ abc_i^c.slot = 0\ \wedge\ abc_i^c.cycle = off$$

In the following lemma we show that under the assumption that the master starts the broadcasting of the first message and that slaves are idle and ready for synchronization reception, the synchronization will happen. Note that we present already fixed results according to the improved low-level correctness (Theorem 2) and using the previously introduced synchronization delay cycle.

Lemma 10 (Synchronization Correctness)**.** *If a synchronization-awaiting slave is directly connected to the synchronization-starting master, then the synchronization signal will be transmitted and the slave will get into the start of slot 0. Fix the following constants and variables:*

- abc_0^j *as the state of the master controller in some cycle* j*;*
- abc_u^j *as the state of an arbitrary slave ECU with* $0 < u < ns$*;*
- $c \in \mathbb{N}$ *as a cycle of the master ECU;*
- $\xi_u = cy_{u,0}(c+2)$ *as receiver cycles;*
- $bvce_S$ *as a function generating a bitvector, s.t.* $bvce_S(c)[i]$ *where* $i \leq c$ *is the bit sent as clock enable signal to* $abc_0.S$ *in cycle* i*.*
- $bvin_S$ *as a function generating bitvector, s.t.* $bvin_S(c)[i]$ *where* $i \leq c$ *is the value sent as input signal to* $abc_0.S$ *in cycle* i ***if*** $abc_0.S$ ***was clocked*** *in cycle* i*; this value will be modeled as the bit, which is stored in this register in cycle* $i+1$*:*

$$\forall i, j : i \leq j \wedge bvce_S(j)[i] = 1 \Rightarrow bvin_S(j)[i] = S^{i+1}$$

For convenience we abbreviate the send register $abc_0.S$ *of the master by* S *and the receive register* $abc_u.R$ *of the receiver by* R*.*

Premises:

(a) In all cycles of the system run, the reset signal stays disabled and all configuration registers contain the correct configuration parameters:

$$\begin{aligned} \forall c, i \in \{0, u\} \quad : \quad & reset^c = 0 \wedge \\ & abc_i^c.CR.ns = bin_6(ns) \ \wedge \ abc_i^c.CR.l = bin_{10}(l) \ \wedge \\ & abc_i^c.CR.off = bin_{32}(off) \ \wedge \ abc_i^c.CR.T = bin_{32}(T) \ \wedge \\ & abc_i^c.CR.iwait = bin_{32}(iwait) \ \wedge \ abc_i^c.CR.sendl = sendl_i \end{aligned}$$

(b) The master is in synchronization starting state, its Send Unit is idle:

$$abc_0^c.state = \texttt{startsnd} \ \wedge \ abc_0^c.sstate = \texttt{s\_idle}$$

(c) The bit sent as clock enable signal to S *in cycle* $c+1$ *is a '1'; in the subsequent* n *cycles it is a '0':*

$$bvce_S(c+1)[c+1] = 1 \ \wedge \ \forall i \in [1:7] : bvce_S(c+8)[c+1+i] = 0$$

(d) The clock enable signal of R *is assumed to be stable around each cycle edge:*

$$\forall i : \forall t \in [e_v(i) - ts : e_v(i) + th] : ce_{R,u}(t) = 1$$

(e) The shift register is filled with ones in cycle $\xi + \sigma_{u,0}(c+2)$*:*[1]

$$abc_u^{\xi+\sigma_{u,0}(c+2)}.sh = 1^4$$

(f) The receiver is in synchronization-awaiting state, its Send Unit is idle:

$$abc_u^{\xi+\sigma_{u,0}(c+2)}.state = \texttt{rcvwait} \wedge abc_u^{\xi+\sigma_{u,0}(c+2)}.rstate = \texttt{r\_idle}$$

(g) The input signal of R *is the output signal of* S *during 8 local cycles of the master:*

$$\forall t \in (e_0(c+2) + tp_{min} : e_0(c+9) + tp_{min}] : In_{R,u}(t) = Out_{S,0}(t)$$

(h) Signal $reset\_ru$ *remains inactive during cycle* ξ *and three cycles after* ξ*:*[2]

$$\forall i \leq 3 : reset\_ru^{\xi+i} = 0$$

Then: *slot 0 starts on the receiver* $\phi_{u,0}(c+2) + 4$ *cycles after the next affected cycle:*

$$\Sigma_u(0)^{\xi+\phi_{u,0}(c+2)+4}$$

Proof Sketch of Lemma 10. (Note that the first premise will be used to ensure that configuration values are available in every cycle and that no reset is possible during a system run.)

First we argue about the output of the send register S. By premise (b) we know that the master is in broadcast-starting state `startsnd` in cycle c. In this state, the Scheduler Control Automaton (Figure 6.5) initiates by its construction signal $startsnd^c$ to the Send Unit, which, in turn, by the construction of the Send Unit Control Automaton (Figure 6.7), triggers the first transition from `s_idle` state to state `TSS0` in the next cycle $c+1$. Thus, we have in cycle $c+1$:

$$abc_0^{c+1}.sstate = \texttt{TSS0}$$

In this state, the Send Unit Automaton activates signal $se1$ but not the signal $se2$. Hence, by construction of the corresponding part of the Output Computation module (Figure 6.8) the input of the send register S in cycle $c+1$ is a '0'. The clock enable of register S is taken from the bitvector $bvce_S(c+1)$, which is assumed to contain a '1' as signal sent in cycle $c+1$ by premise (c). Hence, send register S takes value $TSS[0] = 0$ in cycle $c+2$:

$$S^{c+2} = 0$$

Since Böhm took all premises of Theorem 2 and put them into the premise set of his theorems (premises (b), (c) and (g)) we can directly apply this theorem for cycle $c+2$ and get:

$$\forall i \in [0:6] : R^{\xi+\sigma_{u,v}(c+2)+i} = bvin_S(c+1)[c+1]$$

[1] Note that initially this assumption has additionally required $\hat{R}^{\xi+\sigma_{u,0}(c+2)} = 1$ which is wrong.

[2] Note that this assumption was not in the original version of the lemma and was added later after changes we made to the Elimination Redundancy module, introducing signal $reset\_ru$. More on this in the last chapter of this thesis.

Since the register S was clocked in cycle $c+1$ by premise (c), by definition of $bvin_S$ (see fixed variables) we instantiate the input signal of S in cycle $c+1$ by its content in the next cycle:

$$\forall i \in [0:6] : R^{\xi+\sigma_{u,v}(c+2)+i} = S^{c+2} = 0$$

Thus, we know, that the receive register R will contain '0' in seven consecutive cycles starting from $\xi + \sigma_{u,v}(c+2)$. We make a case destinction on $\sigma_{u,v}(c+2)$.

Case 1: $\sigma_{u,v}(c+2) = 0$**.** By definition of the delay cycle (Definition 8) we can assume, that the timing parameters of R were met and a zero was sampled into R in cycle ξ. By the construction of the Redundancy Elimination module (Figure 6.11(a)) we can easily verify, that in cycle $\xi + 3$ we get

$$\forall P \in \{\hat{R}, sh[3], sh[2]\} : abc_u^{\xi+3}.P = 0$$

if we show that signal $reset\_ru$ remains 0 in cycles $\xi+i$ for $i \in [0:4]$, which follows from the premise (h). Moreover, in cycle $\xi + 3$ the voted bit v will flip to 0 for the first time, since 3 out of 5 inputs to the majority voter are zeros. This will immediately activate the $startedrcv^{\xi+3}$ signal by the construction of the Synchronizer (Figure 6.11(b)). By premise (f), the receiver's Scheduler was waiting for synchronization in state `rcvwait` in cycle ξ. Since the $startedrcv$ signal was inactive until $\xi + 3$ and, thus, the Scheduler remained in state `rcvwait`, by the Scheduler Automaton construction (Figure 6.5), in the next cycle $\xi + 4$ it will switch to state `offwait`, set its cycle counter to off and its slot counter to 0. By definition of slot start (Definition 11) and synchronization delay the claim follows:

$$\Sigma_u(0)^{\xi+4} = \Sigma_u(0)^{\xi+0+4} = \Sigma_u(0)^{\xi+\phi_{u,0}(c+2)+4}$$

Case 2: $\sigma_{u,v}(c+2) = 1$**.** If the timing parameters of R were not met, there are two possibilities: the sampled unstable value flips to zero or to one.

- In the first case (value of R flips to 0), by definition of the synchronization delay (Definition 10) we get: $\phi_{u,0}(c+2) = 0$ because of $R^{\xi} = 0$. Thus, technicaly, we are in the same situation as for $\sigma_{u,0}(c+2) = 0$ and make the same proof as in Case 1.
- In the second case (value of R flips to 1), if the unstable value flips to a 1, the correct value will be sampled at least in the next 7 cycles by Theorem 2. Thus, beginning with cycle $\xi + 1$ we make the same hardware proof as in Case 1 and show that the synchronization will happen 4 cycles after the first sampled zero, namely after $\xi + 1$. This yields:

$$\Sigma_u(0)^{\xi+4+1} = \Sigma_u(0)^{\xi+4+\phi_{u,0}(c+2)}$$

□

Now we also can extend Definition 4, by defining the cycle where the first slot of a slave ECU will be started.

Definition 12 (Start Cycle Of Slot 0 of A Slave)**.** *Let $u \in [1 : ns-1]$ be index of a non-master ECU. Then, slot 0 of every round starts on a slave one cycle after the synchronization signal (sent by master $off + 2$ cycles after its slot start) will be sampled and processed by the slave:*

$$\forall r : \alpha_u(r,0) = cy_{u,0}(\alpha_0(r,0) + off + 2) + \phi_{u,0}(\alpha_0(r,0) + off + 2) + 4$$

7.1.2 Schedule Execution Correctness

The next lemma states that once an ECU i starts the slot 0, it will subsequently execute a fixed schedule consisting of ns slots. Note that ECUs have no notion of rounds, thus, the next lemma is shown for an arbitrary but fixed round.

Lemma 11 (Round Schedule Correctness)**.** *If an ECU i is in state $\Sigma_i(0)$, then it will start the remaining $ns-1$ slots. We fix the following variables:*

- *abc_i as bus controller of ECU_i;*
- *$c \in \mathbb{N}$ as a cycle of ECU_i.*

Premises:

(a) The $reset$ signal stays inactive, as well as configuration registers stay stable during the entire system run:

$$\forall c : reset_i^c = 0 \ \wedge \ abc_i^c.CR = abc_i^{c+1}.CR$$

(b) ECU_i is in slot 0 in cycle c: $\Sigma_i(0)^c$.

Then: *the master will start a new slot every T cycles, whereas a slave starts the first slot after $tc + off$ cycles, and all remaining slots every T cycles:*

$$\begin{aligned} \forall s \in [1 : ns-1] : \quad & (i = 0 \rightarrow \Sigma_0(s)^{c+s\cdot T}) \wedge \\ & (i \neq 0 \rightarrow \Sigma_i(s)^{c+tc+off+(s-1)\cdot T}) \end{aligned}$$

Proof. The premise (a) will be used to conclude, that the configuration registers always contain the same correct values, specified in Section 6.4.1. Since this conclusion is obvious, we will not elaborate on it any further and assume in the remaining explanations of the proof the correct content of configuration registers and absence of reset signal in all cycles.

The proof of the claim is done by induction on s.

- **Induction base:** $s = 1$. We make a case distinction on $i = 0$:

  1. Assume, ECU_i is a master, i.e., $i = 0$. By premise (b) we know, that ECU_0 is in the start of slot 0 and by Definition 11 holds:

     $$abc_0.state = \texttt{offwait} \ \wedge \ abc_0.slot = 0 \ \wedge abc_0.cycle = 0$$

     By construction of the Scheduler Automaton (Figure 6.5) we can show:

     (a) the cycle counter $abc_0.cycle$ will be incremented in states `offwait`, `startsnd` and `Twait`;
     (b) state `offwait` will be left only if $abc_0.cycle = off - 1$;
     (c) state `Twait` will be left only if $abc_0.cycle = T - 1$;
     (d) after `Twait`, the Scheduler of the master ($sendl[0] = 1$) changes to state `offwait`, sets cycle counter to 0 and increments the slot counter if it is smaller than $ns - 1$ ($\neg eqns = 1$).

Hence, the controller abc_0 will stay for off cycles[3] in `offwait`, for 1 cycle in `startsnd` and $T-1-(off+1)$ in `Twait`, which collectively sums up to $T-1$ cycles. By Scheduler Automaton construction we have in the subsequent cycle:

$$abc_0^{c+T}.state = \texttt{offwait} \ \wedge \ abc_0^{c+T}.cycle = 0 \ \wedge \ abc_0^{c+T}.slot = s+1$$

which yields by the definition of a slot start $\Sigma_0(2)^{c+T} = \Sigma_0(s+1)^{c+s\cdot T}$ and proves the claim for the master.

2. In case $i \neq 0$ we have by premise (b):

$$abc_i.state = \texttt{Twait} \ \wedge \ abc_i.slot = 0 \ \wedge abc_i.cycle = off$$

Since we already know, that the Scheduler stays in `Twait` until the cycle counter reaches $T-1$, we can show, that it happens in $T-1-off$ cycles. By Scheduler Automaton construction we have in the subsequent cycle for a slave ($sendl[0]=0$):

$$abc_i^{c+T-off}.state = \texttt{offwait} \ \wedge \ abc_i^{c+T-off}.cycle = 0 \ \wedge \ abc_i^{c+T-off}.slot = s+1$$

which gives us by definition of T and Definition 11:

$$\Sigma_i(2)^{c+T-off} = \Sigma_i(s+1)^{c+tc+off}$$

Hence, the claim of the induction base follows.

- **Induction step:** $s \to s+1$. By induction hypothesis, we get:

$$i = 0 \to \Sigma_0(s)^{c+s\cdot T}$$
$$i \neq 0 \to \Sigma_i(s)^{c+tc+off+(s-1)\cdot T}$$

As above, we make a case distinction on $i = 0$:

1. Let $i = 0$. By induction hypothesis we know, that the master has started slot s in cycle $c+s\cdot T$ and it holds: $\Sigma_0(s)^{c+s\cdot T}$. From this point we proceed along the lines of the proof of the induction base case for $i \neq 0$ and for an arbitrary slot s, which yields a proof, that slot $s+1$ will be started T cycles later: $\Sigma_0(s+1)^{c+s\cdot T+T} = \Sigma_0(s+1)^{c+(s+1)\cdot T}$.
2. Let $i \neq 0$. We make a case distinction on $send(s+1) = i$.
   (a) ECU_i is sender in slot $s+1$. Since the master and slaves share the same hardware, the same configuration parameters and have the same behaviour during the sender slot, we use the same hardware proof as for the Induction Base, which gives us a new slot start in T cycles. Together with the induction hypothesis, the claim follows trivially.
   (b) ECU_i is receiver in slot $s+1$. For the same reason as above, because of equal behavior of master and a non-sending slave in a non-zero proof, we use the proof of the induction step for $i \neq 0$ and get a new slot start T cycles later after the induction hypothesis: $S_i(s+1)^{c+tc+off+(s-1)\cdot T+T} = S_i(s+1)^{c+tc+off+((s+1)-1)\cdot T}$.

   This finishes the proof.

□

[3] Note that cycle counter starts with 0.

7.1.3 Schedule Timing Correctness

The last Lemma of Böhm argues about the correct timing of a schedule executed by every ECU. Basically, it says that if all receivers synchronize with the master as shown in Lemma 10, then every ECU will start every slot s before the sender $ECU_{send(s)}$ will start its message transmission, and, respectively, the ECU will end the slot *after* $ECU_{send(s)}$ finishes the message transmission as depicted in Figure 3.4. However, this result contained two flaws.

Flaws of Previously Proven Lemma

Fundamentally, the idea behind the proof is correct – we want to ensure, that every ECU is able to receive the transmitted message. However, this proof will never be used for the correctness of serial interfaces, since we do not need the time overlapping argument for this, because the Receive Unit is not aware of the schedule per se. It will be triggered as soon as it recognizes an active signal on the bus. Hence, for the message transmission, it should be guaranteed, that all ECUs are synchronized at the start of every round *and* that there is no bus contention during the message transmission. And to show the second argument, we have to be able to relate the local schedule times, like local slot start of a *receiver* with the start of the message transmission on the *sender*.

Although, at the first glance, it seems to be exactly what Böhm has verified in his lemma, it is not. The reason why we have to relate local schedule times of receivers with the message transmission times of the sender, is that states of the Scheduler impact the analog output of an ECU. That means, if ECU starts a slot as a receiver, then and only then we can show, that its analog output will remain idle during the entire slot. However, the problem here is that the generation of the correct analog output triggered by the Scheduler state is certainly almost always *delayed*. In our implementation, this delay consists of two hardware cycles. Practically, it means, that if ECU_u enters a slot s in its local cycle $\alpha_u(r, s)$ as a receiver, then its analog send register output will be updated with the correct value 2 cycles later. (We will show it in a lemma later.)

Thus, what should have been shown in a lemma about correct timing of the Scheduler, is:

1. on all ECUs the slot start cycle plus two additional delay cycles lies *before* the message transmission:

$$e_u(\alpha_u(r, s) + 2) \leq e_{send(s)}(\alpha_{send(s)}(r, s) + \mathit{off} + 2)$$

2. and the message transmission end plus two delay cycles lies *before* the slot end of the receive ECU.

$$e_{send(s)}(\alpha_{send(s)}(r, s) + \mathit{off} + tc + 2) \leq e_u(\alpha_u(r, s) + T + 2)$$

The reason why this flaw remained undiscovered is because Böhm did not couple the state of the hardware scheduler with the analog output of the send register S, and because he did not apply this lemma to show, that the correct schedule timing provides a contention-free bus at the time of the *actual* message transmission, the time when the sender's message hits the bus, instead of the start of the message transmission in the hardware scheduler.

The second flaw was, that Böhm has verified, that *all* ns slots on all ECUs start before the corresponding message transmission. For this, Böhm has defined slot start

times of ECUs in terms of global time as follows. Let $\alpha_u(r,0)$ be the cycle in which ECU_u has sampled and processed the synchronization message sent by the master in the beginning of round r. Then, Böhm defines the global slot start time of slot $s \in [0 : ns-1]$ as:

$$e_u(\alpha_u(r,0)) + (s \cdot T - \mathit{off}) \cdot \tau_u$$

Obviously, this definition is not applicable for slot $s = 0$, since it would mean, that the first slot on a slave ECU starts off cycles *before* the synchronization. However, as we know from Definitions 3 and 4 the fixed schedule of the master lasts exactly $ns \cdot T$ cycles, whereas the fixed schedule of a slave lasts $ns \cdot T - \mathit{off}$ cycles. We also know from Section 3.4 that off is the maximal possible clock drift in terms of local cycles between two ECUs. Hence, the master ECU is waiting off cycles before every synchronization to let all slaves finish the last slot of previous round. Thus, with Böhm's definition it is possible, that the end of slot $ns-1$ of some round r would overlap with the start of slot 0 of round $r+1$.

Besides the confusing nature of this definition, the problem is that we only can argue about local slots of a slave ECU if it has synchronized with the master, thus, we consider the earliest cycle – the first cycle after the synchronization – as the start of slot 0. However, since it was triggered by a synchronization message, which is the first bit of a regular message, the first slot of a slave *cannot* lie *before* the transmission start of the master. Thus, with Böhm's definition it is only possible to show that the schedule timing is correct for all non-zero slots. This didn't cause any problems simply because the correct schedule timing was never applied to the bus model, and because arguing about one round didn't expose the problem of potential overlapping of rounds.

7.1.4 Schedule Timing Correctness (Improved)

Note that the next lemma is not about a concrete implementation (we do not need any hardware computations for it), but rather about the correct schedule timing of hardware, obeying the TDMA-based communication described in Chapter 3 and under the synchronization timing shown in Lemma 10.

Moreover, note that Böhm has computed the offset off as:

$$\mathit{off} := 10 + \lceil ns \cdot T \cdot \Delta \rceil \tag{7.1}$$

First we show one helper lemma.

Lemma 12 (Synchronization Time Bound)**.** *Start times of slot 0 on different slaves are bounded by 3 cycles:*

$$e_u(\alpha_u(r,0)) - e_v(\alpha_v(r,0)) \leq 3 \cdot \tau_v$$

Proof.

$$\begin{aligned}
& e_u(\alpha_u(r,0)) - e_v(\alpha_v(r,0)) \\
(\text{Def. 12}) \quad = \quad & e_u(cy_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + \phi_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + 4) - \\
& e_v(cy_{v,0}(\alpha_0(r,0) + \mathit{off} + 2) + \phi_{v,0}(\alpha_0(r,0) + \mathit{off} + 2) + 4) \\
(\text{Def. 2}) \quad = \quad & e_u(cy_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) - 1) + (\phi_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + 5) \cdot \tau_u - \\
& e_v(cy_{v,0}(\alpha_0(r,0) + \mathit{off} + 2)) + (\phi_{v,0}(\alpha_0(r,0) + \mathit{off} + 2) + 4) \cdot \tau_v \\
(\text{Lemma 7}) \quad \leq \quad & e_0(\alpha_0(r,0) + \mathit{off} + 2) + tp_{min} - th + (\phi_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + 5) \cdot \tau_u - \\
& e_v(cy_{v,0}(\alpha_0(r,0) + \mathit{off} + 2)) + (\phi_{v,0}(\alpha_0(r,0) + \mathit{off} + 2) + 4) \cdot \tau_v
\end{aligned}$$

$$
\begin{aligned}
\text{(Def. 7)} \quad &\leq e_0(\alpha_0(r,0)+\mathit{off}+2)+tp_{min}-th+(\phi_{u,0}(\alpha_0(r,0)+\mathit{off}+2)+5)\cdot\tau_u- \\
&\quad e_0(\alpha_0(r,0)+\mathit{off}+2)+tp_{min}-th+(\phi_{v,0}(\alpha_0(r,0)+\mathit{off}+2)+4)\cdot\tau_v \\
&= (\phi_{u,0}(\alpha_0(r,0)+\mathit{off}+2)+5)\cdot\tau_u-(\phi_{v,0}(\alpha_0(r,0)+\mathit{off}+2)+4)\cdot\tau_v \\
\text{(Lemma 1)} \quad &\leq (\phi_{u,0}(\alpha_0(r,0)+\mathit{off}+2)+5)\cdot\tau_v\cdot(1+\Delta)-(\phi_{v,0}(\alpha_0(r,0)+\mathit{off}+2)+4)\cdot\tau_v \\
(\phi_{x,y}(z)\leq 1) \quad &\leq 6\cdot\tau_v\cdot(1+\Delta)-4\cdot\tau_v \\
&= (6\cdot(1+\Delta)-4)\cdot\tau_v=(2+6\cdot\Delta)\cdot\tau_v\leq 3\cdot\tau_v
\end{aligned}
$$

□

Lemma 13 (Schedule Timing Correctness)**.** *Fix following constants:*

- clk_0 *as the clock of the master ECU;*
- clk_u, clk_v *are two arbitrary clocks with* $\forall x\in\{u,v\}: 0<x<ns$*;*

If bus controllers with clocks clk_u *and* clk_v *have synchronized with the bus controller with the clock* clk_0 *in cycles* $\alpha_u(r,0)$ *and* $\alpha_v(r,0)$*, then every slot on every receiver starts before and ends after the message transmission in slot* $s\in[1:ns-1]$*:*

1. $e_u(\alpha_u(r,s)+2)+tp_{min}\leq e_v(\alpha_v(r,s)+\mathit{off}+2)+tp_{min}$
2. $e_u(\alpha_u(r,s)+\mathit{off}+tc+2)+tp_{min}\leq e_v(\alpha_v(r,s)+T+2)+tp_{min}$

Proof. • Slot start lies before transmission start:

$$
\begin{aligned}
& e_u(\alpha_u(r,s)+2)+tp_{min}-(e_v(\alpha_v(r,s)+\mathit{off}+2)+tp_{min}) \\
\text{(Lemma 6)} \quad &= e_u(\alpha_u(r,0)+tc+\mathit{off}+(s-1)\cdot T+2)+tp_{min}- \\
&\quad (e_v(\alpha_v(r,0)+tc+\mathit{off}+(s-1)\cdot T+\mathit{off}+2)+tp_{min}) \\
&= e_u(\alpha_u(r,0)+tc+\mathit{off}+(s-1)\cdot T+2)- \\
&\quad e_v(\alpha_v(r,0)+tc+\mathit{off}+(s-1)\cdot T+\mathit{off}+2) \\
\text{(Definition 2)} \quad &\equiv e_u(\alpha_u(r,0))+(tc+\mathit{off}+(s-1)\cdot T+2)\cdot\tau_u- \\
&\quad e_v(\alpha_v(r,0))+(tc+\mathit{off}+(s-1)\cdot T+\mathit{off}+2)\cdot\tau_v \\
&= e_u(\alpha_u(r,0))-e_v(\alpha_v(r,0))+(tc+\mathit{off}+(s-1)\cdot T+2)\cdot\tau_u- \\
&\quad (tc+\mathit{off}+(s-1)\cdot T+\mathit{off}+2)\cdot\tau_v \\
\text{(Lemma 12)} \quad &\leq (tc+\mathit{off}+(s-1)\cdot T+2)\cdot\tau_u- \\
&\quad (tc+\mathit{off}+(s-1)\cdot T+\mathit{off}-1)\cdot\tau_v \\
\text{(Lemma 1)} \quad &\leq (tc+\mathit{off}+(s-1)\cdot T+2)\cdot\tau_v\cdot(1+\Delta)- \\
&\quad (tc+\mathit{off}+(s-1)\cdot T+\mathit{off}-1)\cdot\tau_v \\
&= (tc+\mathit{off}+(s-1)\cdot T+2+(tc+\mathit{off}+(s-1)\cdot T+2)\cdot\Delta- \\
&\quad (tc+\mathit{off}+(s-1)\cdot T+\mathit{off}-1))\cdot\tau_v \\
&= (3+(tc+\mathit{off}+(s-1)\cdot T+2)\cdot\Delta-\mathit{off})\cdot\tau_v \\
\text{(Def. of } \mathit{off} \text{ and } T) \quad &\leq (3+s\cdot T\cdot\Delta-\mathit{off})\cdot\tau_v \\
(s\leq ns-1) \quad &\leq (3+ns\cdot T\cdot\Delta-\mathit{off})\cdot\tau_v \\
\text{(Definition of } \mathit{off}) \quad &\leq 0
\end{aligned}
$$

- Slot end lies after transmission end.

$$
\begin{aligned}
& e_v(\alpha_v(r,s) + \mathit{off} + tc + 2) + tp_{min} - (e_u(\alpha_u(r,s) + T + 2) + tp_{min}) \\
(\text{Definition 12}) \quad = \quad & e_v(\alpha_v(r,0) + tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2) + tp_{min} - \\
& (e_u(\alpha_u(r,0) + tc + \mathit{off} + s \cdot T + 2) + tp_{min}) \\
= \quad & e_v(\alpha_v(r,0) + tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2) - \\
& e_u(\alpha_u(r,0) + tc + \mathit{off} + s \cdot T + 2) \\
(\text{Definition 2}) \quad = \quad & e_v(\alpha_v(r,0)) + (tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2)\cdot \tau_v - \\
& (e_u(\alpha_u(r,0)) + (tc + \mathit{off} + s \cdot T + 2)\cdot \tau_u) \\
(\text{Definition 2}) \quad = \quad & e_v(\alpha_v(r,0)) - e_u(\alpha_u(r,0)) + (tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2)\cdot \tau_v - \\
& (tc + \mathit{off} + s \cdot T + 2)\cdot \tau_u \\
(\text{Lemma 12}) \quad \leq \quad & (tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2)\cdot \tau_v - \\
& (tc + \mathit{off} + s \cdot T - 1)\cdot \tau_u \\
(\text{Lemma 1}) \quad \leq \quad & (tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2)\cdot (1+\Delta)\cdot \tau_u - \\
& (tc + \mathit{off} + s \cdot T - 1)\cdot \tau_u \\
= \quad & ((tc + \mathit{off} + (s-1)\cdot T + \mathit{off} + tc + 2)\cdot (1+\Delta) - \\
& (tc + \mathit{off} + s \cdot T - 1))\cdot \tau_u \\
= \quad & ((s\cdot T + tc + 2)\cdot (1+\Delta) - (tc + \mathit{off} + s\cdot T))\cdot \tau_u \\
= \quad & ((s\cdot T + tc + 2)\cdot \Delta + 2 - \mathit{off})\cdot \tau_u \\
(\text{Def. } T) \quad \leq \quad & (((s+1)\cdot T + 2)\cdot \Delta + 2 - \mathit{off})\cdot \tau_u \\
(s \leq ns - 1) \quad \leq \quad & (ns\cdot T\cdot \Delta + 2 - \mathit{off})\cdot \tau_u \\
(\text{Def. } \mathit{off}) \quad \leq \quad & 0
\end{aligned}
$$

□

7.2 From Local to Global Computational Semantics

Until now, all lemmas from Section 7.1 have argued about computational traces, modeling a computation of a bus controller only. Every hardware state abc^c in cycle c of some controller was provided by a fixed valid execution trace function $trace$ and input function $inputs$, s.t. $abc^c = trace(c)$ with:

$$\forall i : trace(i+1) = \delta_{abc}(trace(i), inputs(i))$$

That means, such modeling does not provide a straightforward extension of the semantics by a processor computational semantics later. Moreover, the assumptions about stable configuration registers in all cycles and inactive reset signal (e.g., premise (a) in Lemma 10) restrict the modeling to the post-configuration period of the bus controller computation.

But in our final correctness of a message transmission, we want to argue about distributed asynchronously working ECUs, which can be instantiated with a concrete processor model later as specified in Section 6.1. Hence, we want to model every hardware cycle of an ECU computation, including the configuration phase. This requires to discharge some assumptions made in previous results.

The results from Section 7.1 have required – among other technical assumptions – the following:

$$\begin{aligned}\forall c, i \in \{0,u\} \quad : \quad & reset^c = 0 \wedge \\ & abc_i^c.CR.ns = bin_6(ns) \ \wedge \ abc_i^c.CR.l = bin_{10}(l) \ \wedge \\ & abc_i^c.CR.off = bin_{32}(off) \ \wedge \ abc_i^c.CR.T = bin_{32}(T) \ \wedge \\ & abc_i^c.CR.iwait = bin_{32}(iwait) \ \wedge \ abc_i^c.CR.sendl = sendl_i\end{aligned}$$

That means, it is required to show, that in the entire system run no reset signal will be activated, and that in every configuration register the corresponding parameter is stored.

First we introduce the following assumption on the global computational trace of every ECU.

Assumption 7 (Configuration Registers)**.** *Let ns be an even number. Let ECU_i be arbitrary ECU with $i < ns$. Let $setrd\_cycle_i$ be the hardware cycle, where the signal $setrd_i$ gets active:*

$$setrd_i^{setrd\_cycle_i} = 1$$

The configuration registers of every ECU_i will be filled with meaningful values before the end of the configuration phase. The write enable signal $crwe$ stays inactive after the configuration phase.

$$\begin{aligned}&\forall c \geq setrd\_cycle_i : crwe^c = 0 \wedge \\ &ECU_i^{setrd\_cycle_i}.abc.ns = bin_6(ns) \ \wedge \ ECU_i^{setrd\_cycle_i}.abc.l = bin_{10}(l) \wedge \\ &ECU_i^{setrd\_cycle_i}.abc.off = bin_{32}(off) \ \wedge \ ECU_i^{setrd\_cycle_i}.abc.T = bin_{32}(T) \wedge \\ &ECU_i^{setrd\_cycle_i}.abc.iwait = bin_{32}(iwait) \ \wedge \ ECU_i^{setrd\_cycle_i}.abc.sendl = sendl_i\end{aligned}$$

Having this, we easily show by induction, that all registers remain stable after the $setrd\_cycle$.

Lemma 14 (Configuration Register Stability)**.**

$$\begin{aligned}\forall c \geq setrd\_cycle_i \quad : \quad & ECU_i^c.abc.ns = bin_6(ns) \ \wedge \ ECU_i^c.abc.l = bin_{10}(l) \ \wedge \\ & ECU_i^c.abc.off = bin_{32}(off) \ \wedge \ ECU_i^c.abc.T = bin_{32}(T) \ \wedge \\ & ECU_i^c.abc.iwait = bin_{32}(iwait) \ \wedge \ ECU_i^c.abc.sendl = sendl_i\end{aligned}$$

Now we can satisfy the assumption from previous lemmas about stable configuration registers as follows. We show that a global ECU computation trace which models hardware computations of any ECU_i starting from cycle 0, can simulate the bus controller trace, starting from cycle $setrd\_cycle_i$. That is, every cycle c of the bus controller trace used by Böhm was mapped to cycle $setrd\_cycle_i + c$ of the global trace. Let x be one of the configuration registers and y its corresponding value. Instead of satisfying assumptions required by Böhm on the global trace for some configuration register x:

$$\forall c : abc_i^c.CR.x = y$$

we map these assumptions to a post-configuration phase of the global ECU trace:

$$\forall c : ECU_i^{c+setrd\_cycle_i}.abc.CR.x = y$$

The new assumption can be discharged by Lemma 14 using Assumption 7. The assumption of an inactive reset signal will be discharged in the same way using Assumption 5.

This is a very technical work which requires a lot of Isabelle-related notation to be introduced and, hence, we will not elaborate on it in this thesis.

From now on, all lemmas and theorems will be shown not for bus controller hardware only as in Section 7.1, but for global computation traces of ECUs as specified in Section 6.1. Moreover, we do not list all assumption we use explicitly, but, instead, we implicitly use all introduced assumption so far.

7.3 Startup Correctness

We will show that all slave ECUs reach the state `rcvwait`, and the master ECU will start its first slot.

Before we start with stating a lemma about the correctness of the startup routine, we have to figure out which assumptions we can make about the startup routine. As described in Section 6.4.2, the idea of the startup routine is quite simple. All ECUs are intented to be started roughly at the same time. In the very first cycle, every controller will be reset by its processor by activating the reset signal. Afterwards, the configuration phase begins, where all slaves will be initialized with the same configuration parameters except for the content of bitvector $sendl_u$, which determines whether ECU is a slave or the master. Since we want to keep the interface between the bus controller and the processor as general as possible, we do not argue about the correctness of filling the configuration registers, instead we assume in Assumption 7, that all registers are filled with correct values during the configuration phase.

After the configuration phase, all slaves switch to state `rcvwait` and remain there until they get a synchronization signal sent by the master. The master, switches after the configuration to state `iwait` and remains there for $iwait$ cycles. The intention was that the number of cycles $iwait$ should be long enough *to guarantee* that all slaves have had enough time to pass the configuration phase and to switch to sate `rcvwait`, where slaves wait for the next synchronization. However, such an assumption requires an additional assumption – a time bound of the startup times of all ECUs: master *and* slaves.

Instead, we fix one master cycle t_0 such that it is the first cycle after the master has left state `iwait`, and assume that if the master is in cycle t_0, then all slaves are in state `rcvwait`.

Assumption 8 (Startup Correctness)**.** *Let t_0 be the first cycle after the master has left state `iwait`. Then we assume, that all slaves are in state `rcvwait` in cycle $cy_{u,0}(t_0)$.*

$$\begin{aligned} ECU_0^{t_0}.abc.state = \texttt{offwait} \ \wedge \\ \forall c < t_0 : ECU_0^c.abc.state \in \{\texttt{idle}, \texttt{iwait}\} \ \wedge \\ \forall u \in [1 : ns-1] : ECU_u^{cy_{u,0}(t_0)}.abc.state = \texttt{rcvwait} \ \wedge \\ \forall c \leq cy_{u,0}(t_0) : ECU_u^c.abc.state \in \{\texttt{idle}, \texttt{rcvwait}\} \end{aligned}$$

Obviously, t_0 is the smallest master cycle, where its hardware can be interpreted as being in a slot start according to the Scheduler Automaton construction (Figure 6.5) and Definition 11.

Definition 13 (Start Cycle Of Slot 0 of Master)**.**

t_0 is the cycle in which the master starts slot 0 of round 0:

$$\Sigma_0(0)^{t_0} \equiv \Sigma_0(0)^{\alpha_0(0,0)}$$

Finally, we have to show, that Assumption 8 is sound by showing all ECUs will reach states `rcvwait` and `offwait` after the initial reset signal.

Lemma 15 (Startup Correctness)**.**

$$\begin{aligned} \forall i < ns-1 : (sendl_i[0] = 0 \quad &\Rightarrow \quad \exists x : ECU_i^x.abc.state = \texttt{rcvwait} \wedge \\ &\quad\ \forall y \leq x : ECU_i^y.abc.state \in \{\texttt{idle}, \texttt{rcvwait}\}) \\ (sendl_i[0] = 1 \quad &\Rightarrow \quad \exists x : ECU_i^x.abc.state = \texttt{offwait} \wedge \\ &\quad\ \forall y < x : ECU_i^y.abc.state \in \{\texttt{idle}, \texttt{iwait}\}) \end{aligned}$$

Proof. Since we have assumed, that the $setrd$ signal will be activated eventually (Assumption 5), the proof follows from the construction of the Scheduler, mainly automatically. □

In the next two sections we show that all ECUs execute their schedules correctly in every round. That is, we will show, that the slot start state of the bus controller will be established at recurrent points in time on the master and on all receivers. The latter part is non-trivial, since it *assumes* correct synchronization between master and all slaves in the current round and *implies* correct synchronization in the next round.

Although we argue about global traces of ECUs as described in Section 7.2, for convenience, we will abbreviate in most of the following proofs, the bus controller $ECU_i.abc$ of an ECU_i by abc_i.

7.4 Schedule Correctness Of Master for All Rounds

In this section we will show, that the master ECU executes its schedule in every round. This part is easy because, the master's schedule execution is static during the entire system run and depends solely on the hardware correctness. In Chapter 3 we have defined (Definition 3) the local time notion of the master ECU, assuming the existence of a cycle $\alpha_0(0,0)$, where slot 0 of round 0 starts. As mentioned there, this cycle depends on a concrete implementation of a time-triggered system. Obviously, in our case it is the cycle t_0 by Definition 13, since it is the smallest cycle, where predicate $\Sigma_0(0)$ holds, indicating a hardware state which is starting slot 0.

Lemma 16 (Schedule Correctness of the Master for All Rounds)**.** *The master ECU will start all slots of all rounds:*

$$\forall r, s < ns : \Sigma_0(s)^{\alpha_0(r,s)}$$

Proof. Proof by induction on round r.

1. **Induction base:** $r = 0$. We have to show that every slot of the first round will be started in the corresponding slot start cycle:

   $$\forall s < ns : \Sigma_0(s)^{\alpha_0(0,s)}$$

   By definition of cycle t_0 (Definition 13) we have:

   $$\Sigma_0(0)^{t_0} = \Sigma_0(0)^{\alpha_0(0,0)} \tag{7.2}$$

   We apply Lemma 11 for $i := 0$ and $c := \alpha_0(0,0)$. The premise (a) of Lemma 11 will be discharged as described in Section 7.2. Premise (b) is discharged by 7.2. We get:

   $$\forall s \in [1 : ns-1] : \Sigma_0(s)^{\alpha_0(0,0)+s\cdot T}$$

By Lemma 5 we show:

$$\alpha_0(0,0) + s \cdot T = \alpha_0(0,s)$$

and the claim follows.

2. **Induction step:** $r \rightarrow r+1$. By induction hypothesis we get for round r the execution of all ns slots:

$$\forall s < ns : \Sigma_0(s)^{\alpha_0(r,s)}$$

We have to show, that in the next round $r+1$, all slots will be executed at appropriate local time:

$$\forall s < ns : \Sigma_0(s)^{\alpha_0(r+1,s)}$$

From the induction hypothesis we conclude: $\Sigma_0(ns-1)^{\alpha_0(r,ns-1)}$. That is, the last slot of a round $ns-1$ will be started in cycle $\alpha_0(r, ns-1)$:

$$abc_0^{\alpha_0(r,ns-1)}.state = \texttt{offwait} \ \wedge \ abc_0^{\alpha_0(r,ns-1)}.slot = ns-1 \ \wedge abc_0^{\alpha_0(r,ns-1)}.cycle = 0$$

By Scheduler construction we can easily show, that in $T-1$ cycles, the Scheduler of the master will be in state `Twait` with active signals eqT and $eqns$. Hence, one cycle later, the Scheduler will switch to state `offwait` resetting the values of the slot and cycle counters:

$$abc_0^{\alpha_0(r,ns-1)+T}.state = \texttt{offwait} \wedge abc_0^{\alpha_0(r,ns-1)+T}.slot = 0 \wedge abc_0^{\alpha_0(r,ns-1)+T}.cycle = 0$$

which is equivalent to $\Sigma_0(0)^{\alpha_0(r,ns-1)+T}$ and by Definitions 11 and 3 to:

$$\Sigma_0(0)^{\alpha_0(r+1,0)} \tag{7.3}$$

As in the induction base case, we apply Lemma 11 for cycle $\alpha_0(r+1,0)$ and get with 7.3:

$$\forall s \in [1 : ns-1] : \Sigma_0(s)^{\alpha_0(r+1,0)+s \cdot T}$$

By Lemma 6 we conclude:

$$\alpha_0(r+1,0) + s \cdot T = \alpha_0(r+1,s)$$

and the claim follows.

□

7.5 Schedule Correctness Of A Slave for All Rounds

While the schedule execution for all rounds of the master is almost trivial due to its static nature, the correctness for all rounds of slaves is more challenging. First we will extend the local time notion of a slave introduced in Definition 4, by defining the time of the very first slot in every round. This time depends on the master's time of slot 0 of every round, because shortly after the start of slot 0 the master broadcasts the synchronization message over the bus. To receive this synchronization message three requirements should be met: (i) the serial interfaces of the sender and all receivers are in appropriate hardware states, (ii) the low-level signal transmission works (Theorem 2), and (iii) the bus connection can be abstracted to a direct wire connection between the master and every slave (premise (c) of Theorem 2). The main challenge here is to

satisfy the assumptions of Theorem 2, requiring to prove that the analog output of all send registers of all non-receivers does not produce any bus activity in the time period between the end of the last transmission of a round and the first transmission of the next round.

We start arguing about the analog send register output of an ECU in dependence of its Scheduler state in Section 7.5.1. We will show, that a non-transmitting ECU never produce any disturbing non-idle bus activity. Second, using these arguments, we will formally show in Section 7.5.2, that during the time period where all ECUs are waiting for a synchronization, no ECU (including the master) is sending until the synchronization.

The Isabelle counterparts of the following lemmas and proofs are extremely rich in detail and are built of dozens of lemmas. We will skip most of them and will sketch the proof outlining the basic strategy and idea behind the proofs.

7.5.1 Analog Output of a Receiver

In this section we will investigate the correlation of the Scheduler state and the analog output of the send register $abc_i.S$. In contrast to Theorem 2 we do not argue about signal transmission among different DRM models. Instead, we need to couple events of the digital hardware, like states of control automaton with the analog behaviour of the send register S. Since we have to use the DRM interpretation of the send register, we also have to provide instantiations for input signals needed by DRM. Moreover, to stay within the semantics provided in Chapter 4, we need functions returning lists of digital signals sent as inputs to the digital send register S.

Definition 14 (Clock Enable and Input Signal Generators)**.** *Let $genCEs_u : \mathbb{N} \to \mathbb{B}^*$ be a function, which returns for given cycle number n, a list of bits, sent as clock enable signals to the send register S of ECU_u. We compute $genCEs(n)$ as follows:*

$$\begin{aligned} genCEs_u(0) &= abc_u^0.dnbit \\ genCEs_u(n) &= abc_u^n.dnbit \circ genCEs_u(n-1) \end{aligned}$$

Let $genINs_u : \mathbb{N} \to \mathbb{B}^*$ be a function, which returns for given cycle number n, a list of bits which coincides at every index $i \in [0 : n-1]$ with the list of bits sent as input values to the send register S of ECU_u, if S was clocked in cycle i. [4] *We compute $genINs(n)$ as follows:*

$$\begin{aligned} genINs_u(0) &= abc_u^1.S \\ genINs_u(n) &= abc_u^{n+1}.S \circ genINs_u(n-1) \end{aligned}$$

We instantiate in all following lemmas the clock enable input signal $ce_{S,u}$ and input signal $In_{S,u}$ of register S with signals derived from their digital counterparts:

$$\forall c \in \mathbb{N}, t \in \mathbb{R} : t \leq e_u(c) + th \Rightarrow ce_{S,u}(t) = conv(genCEs_u(c))(t) \tag{7.4}$$

$$\forall c \in \mathbb{N}, t \in \mathbb{R} : t \leq e_u(c) + th \Rightarrow In_{S,u}(t) = conv(genINs_u(c))(t) \tag{7.5}$$

Moreover, we show a helper lemma, allowing us to reduce the signals generator functions to needed bits.

[4] Note that we never use $genINs_u(n)[i]$ if S was not clocked in cycle i.

Lemma 17 (Bit pickers)**.**

$$\forall i \leq n : genCEs_u(n)[i] = ECU_u^i.abc.dnbit$$

$$\forall i \leq n : genINs_u(n)[i] = ECU_u^{i+1}.abc.S$$

Proof. Straightforward (by induction on n).

In the next lemma we show, that the clock enable signal $ce_{S,u}$ of send register S of ECU_u returns for all times around a clock edge i the digital value of the $dnbit$ register in cycle $i-1$.

Lemma 18 (Clock Enable Signal Derivation)**.**

$$\forall t \in [e_u(i) - ts : e_u(i) + th] : ce_{S,u}(t) = ECU_u^{i-1}.abc.dnbit$$

Proof. Let $t \in [e_u(i) - ts : e_u(i) + th]$. We instantiate $ce_{S,u}$ as in Equantion 7.4 with $c = i$:

$$ce_{S,u}(t) = conv(genCEs_u(i))(t)$$

By Assumption 3 with instantiations $i := i-1, n = i+1, t := t$ we get:

$$ce_{S,u}(t) = genCEs_u(i)[i-1]$$

And by Lemma 17 the claim follows. □

We proceed with a lemma, which establishes a dependency between states of the Scheduler and states of the Send Unit. As depicted in Figure 6.5, the only state, which triggers the Send Unit by activating signal $startsnd$ is the state `startsnd`. After this state the Scheduler switches to state `Twait`, where the Send Unit is transmitting a message during tc cycles. The Scheduler remains in `Twait` for $off + tc - 1$ cycles, since arriving at `Twait` after `startsnd` is only possible if the cycle counter contains the number $off + 1$.[5] That is, if the Send Unit can be proven to take not more than tc cycles for a message transmission, it will be idle when the Scheduler leaves the state `Twait`. Thus, the Send Unit can be non-idle only if the Scheduler is in state `Twait`.

Lemma 19 (No `Twait` Implies Idle Send Unit)**.** *If the Scheduler of ECU_u is not in state `Twait`, then it's Send Unit is idle.*

$$ECU_u^c.abc.state \neq \texttt{Twait} \Rightarrow ECU_u^c.abc.sstate = \texttt{s\_idle}$$

Proof. By the control Automaton of the Send Unit (Figure 6.7) we know, that it if it is idle, it will only be triggered by an incoming $startsnd$ signal. By Scheduler Automaton construction (Figure 6.5) we know, that this signal is active only in state `startsnd`. We prove the claim by induction on c.

1. **Base case:** $c = 1$. The first cycle takes place after the reset signal gets active. It resets the entire hardware and all automata to their idle states, so the claim follows by construction of the Send Unit, Scheduler and by Assumption 5.

[5] Leaving of state `offwait` when $abc.cycle = off - 1$, arriving in `startsnd` when $abc.cycle = off$ and one cycle will be spent in `startsnd`

2. **Induction step:** $c \rightarrow c+1$. By induction hypothesis we have:

$$ECU_u^c.abc.state \neq \texttt{Twait} \Rightarrow ECU_u^c.abc.sstate = \texttt{s\_idle}$$

We have to show:

$$ECU_u^{c+1}.abc.state \neq \texttt{Twait} \Rightarrow ECU_u^{c+1}.abc.sstate = \texttt{s\_idle}$$

We make a case distinction on the fact whether the Scheduler was in state `startsnd` in cycle c.

(a) **Case 1:** $abc_u^c.state = \texttt{startsnd}$. In this case we know that by Scheduler Automaton construction, it will switch into state `Twait` in the next cycle. However, this contradicts the premise we have in the induction step:

$$abc_u^{c+1}.state \neq \texttt{Twait}$$

Hence, the case is closed by contradiction.

(b) **Case 2:** $abc_u^c.state \neq \texttt{startsnd}$. Here we consider the case distinction on whether the Scheduler was in state `Twait` in cycle c.

i. **Case 1:** $abc_u^c.state \neq \texttt{Twait}$. This case allows us to use the induction hypothesis for cycle c and we can conclude $abc_u^c.sstate = \texttt{s\_idle}$. By hardware construction we can easily prove, that if the Send Unit is idle in cycle c, and since the Scheduler is not in state `startsnd` (by outer Case 2), then there is no way for activation of the $startsnd$ signal and the Send Unit will stay idle in cycle $c+1$.

ii. **Case 2:** $abc_u^c.state = \texttt{Twait}$. This case eliminates the induction hypothesis and the only additional assumption we have is that in cycle $c+1$ the Scheduler has left the state `Twait`: $abc_u^{c+1}.state \neq \texttt{Twait}$. That is, we have to show, that when the Scheduler leaves the state `Twait` in cycle $c+1$ its Send Unit is always idle: $abc_u^{c+1}.sstate = \texttt{s\_idle}$. To show this, first we have to show a lemma, that state `Twait` can only be entered from state `rcvwait`, `offwait` and `startsnd`. In the first two cases the Send Unit will not be triggered and stays idle in the next cycle. In case of `startsnd`, the Send Unit will be started and remains non-idle during the entire transmission lasting tc cycles. Then we show, that if the transmission is over, the Send Unit returns to idle state and the Scheduler is still in state `Twait`. Then we show that the Send Unit cannot be started while the Scheduler is in state `Twait`, hence, we show that the Send Unit remains idle until the Scheduler leaves state `Twait`, and the claim of Lemma follows.

□

Now we will formally couple the state of the Send Unit and the analog output of the send register. Obviously, as depicted in Figure 6.8, the clock enable signal of the send register S is computed in cycle c as the content of register $abc_i^c.dnbit$. This register is clocked in every cycle. By construction of the Output Computation module, we can show, that the input clocked into the $dnbit$ register is a '0' as long as the input s_idle, denoting the idle Send Unit, is active and signal $startsnd$ is inactive. Since the Send Unit will be triggered by the Scheduler initiating the $startsnd$ signal, we can show a lemma, stating, that as long as the bus controller does not act as a sender, its analog output will stay idle ('1').

Lemma 20 (Analog Output of the Send Register)**.** *If the Send Unit of an ECU is idle, its analog output remains idle and spike-free as long as it does not start a message transmission:*

$$\begin{aligned} ECU_u^c.abc.sstate = \texttt{s\_idle} \ \wedge \ \forall x \in [c : c+y] : ECU_u^x.abc.state \neq \texttt{startsnd} \\ \Rightarrow \forall t \in [e_u(c+2) + tp_{min} : e_u(c+y+3) + tp_{min}] : Out_{S,u}(t) = 1 \end{aligned}$$

Proof. We prove Lemma 20 by induction on y.

1. **Induction base:** $y = 0$. For $y = 0$ we have to show:

$$\begin{aligned} ECU_u^c.abc.sstate = \texttt{s\_idle} \ \wedge \ ECU_u^c.abc.state \neq \texttt{startsnd} \\ \Rightarrow \forall t \in [e_u(c+2) + tp_{min} : e_u(c+3) + tp_{min}] : Out_{S,u}(t) = 1 \end{aligned}$$

   By Scheduler construction we can verify, that signal $startsnd$ is inactive in cycle c, since the Scheduler is not in state `startsnd`. Since we know that the Send Unit is idle in c, by Output Computation module construction (Figure 6.8) we can conclude, that a zero will be clocked into $abc_u.dnbit$ in cycle c:

$$abc_u^{c+1}.dnbit = 0 \tag{7.6}$$

   In DRM we get by Definition 6:

$$\forall : t \in [e_u(c+2) + tp_{min} : e_u(c+3) + tp_{min}] : Out_{S,u}(t) = Out_{S,u}(e_u(c+2)) \tag{7.7}$$

   if the condition $stable(ce_{S,u}, c+2) \wedge ce_{S,u}(e_u(c+2)) = 0$ is satisfied. By Equation 7.4 and Lemma 18 we know:

$$\forall t \in [e_u(c+2) - ts : e_u(c+2) + th] : ce_{S,u}(t) = abc_u^{c+1}.dnbit = 0$$

   That is, we have $ce_{S,u}(e_u(c+2)) = 0$, and by Definition 5 we conclude:

$$\begin{aligned} stable(ce_{S,u}, c+2) &\equiv \exists b \in \mathbb{B} : \forall t \in [e_u(c+2) - ts : e_u(c+2) + th] : ce_{S,u}(t) = b \\ &\equiv \forall t \in [e_u(c+2) - ts : e_u(c+2) + th] : ce_{S,u}(t) = 0 \end{aligned}$$

   Hence, by 7.7 we can simplify the initial claim to:

$$abc_u^c.sstate = \texttt{s\_idle} \ \wedge \ abc_u^c.state \neq \texttt{startsnd} \Rightarrow Out_{S,u}(e_u(c+2)) = 1$$

   It remains to show, that the analog output of the send register contains the idle value at time $e_u(c+2)$ if the Scheduler is not in a broadcast starting state and the Send Unit is idle in cycle c. We show this claim by induction on c:

   (a) In the induction base case $c = 0$, we show that in the second cycle after the startup, the send register S will be initialized with idle value '1' and state `startsnd` will not be entered by the Scheduler in the next three cycles.

   (b) In the induction step $c \to c+1$ we get by the induction hypothesis:

$$abc_u^c.sstate = \texttt{s\_idle} \ \wedge \ abc_u^c.state \neq \texttt{startsnd} \Rightarrow Out_{S,u}(e_u(c+2)) = 1$$

We have to show:

$$abc_u^{c+1}.sstate = \texttt{s\_idle} \ \wedge \ abc_u^{c+1}.state \neq \texttt{startsnd} \Rightarrow Out_{S,u}(e_u(c+3)) = 1$$

Note that we can only use the induction hypothesis if its premise is satisfied. However, we can use the same premise for cycle $c+1$, which is the premise of the claim to show:

$$abc_u^{c+1}.sstate = \texttt{s\_idle} \ \wedge \ abc_u^{c+1}.state \neq \texttt{startsnd} \tag{7.8}$$

First we consider the case, that the premise of the induction hypothesis is not satisfied:

$$abc_u^{c}.sstate \neq \texttt{s\_idle} \ \vee \ abc_u^{c}.state = \texttt{startsnd} \tag{7.9}$$

If the left part holds, i.e., the Send Unit is not idle, we know that it will be idle in the next cycle by 7.8. Hence, by the construction of the Send Unit control Automaton (Figure 6.7), we can show that the Send Unit can only be in the last state `TES1` of the sending protocol in cycle c. As we know from the protocol specification, the value $TSS[1] = 1$ has to be sampled into the send register S. By Send Unit control Automaton (Figure 6.7) we can show that in state `TSS1` both signals $se1$ and $se2$ are activated. Hence, by output computation construction (Figure 6.8) the input to the send register S is a '1'. Moreover, we show that according to the output computation module construction this value will be written to S in some cycle $c' \leq c+1$. We also show, that the content of S will not be changed until cycle $c+1$. Then, we show by induction, that the analog output will be equal to the sampled value in cycle c' by Definition of 6:

$$Out_{S,u}(e_u(c+3)) = In_{S,u}(e_u(c'+1)) = 1$$

The case where the second part of 7.9 holds is easier to resolve, because in this case together with 7.8 we have:

$$abc_u^{c}.state = \texttt{startsnd} \ \wedge \ abc_u^{c+1}.sstate = \texttt{s\_idle}$$

By Lemma 19 we get $abc_u^{c}.sstate = \texttt{s\_idle}$. Now we can easily verify that the Scheduler Automaton would activate the signal $startsnd$ in cycle c, which would trigger the idle Send Unit, which becomes non-idle in cycle $c+1$ and this case is closed by contradiction. Thus, we have shown that if the premise of the induction hypothesis does not hold, we still can show the claim. If the premise holds, we have:

$$abc_u^{c}.sstate = \texttt{s\_idle} \wedge abc_u^{c}.state \neq \texttt{startsnd} \wedge Out_{S,u}(e_u(c+2)) = 1 \tag{7.10}$$

It remains to show, that the analog output of the send register did not change:

$$Out_{S,u}(e_u(c+3)) = Out_{S,u}(e_u(c+2)) \tag{7.11}$$

By Definition 6 the equation 7.11 holds if:

$$stable(ce_{S,u}, c+2) \ \wedge \ ce_{S,u}(e_u(c+2)) = 0$$

By Lemma 18 we know: $ce_{S,u}(e_u(c+2)) = abc_u^{c+1}.dnbit$ As in the induction base case, by 7.10, Scheduler Automaton and output computation module construction we can show:

$$abc_u^{c+1}.dnbit = 0$$

And the claim follows.

2. **Induction step:** $y \to y+1$. By induction hypothesis we have:

$$\begin{aligned} ECU_u^c.abc.sstate = \texttt{s\_idle} \wedge \forall x \in [c : c+y] : ECU_u^x.abc.state \neq \texttt{startsnd} \\ \Rightarrow \forall t \in [e_u(c+2) + tp_{min} : e_u(x+3) + tp_{min}] : Out_{S,u}(t) = 1 \end{aligned}$$

We have to show:

$$\begin{aligned} ECU_u^c.abc.sstate = \texttt{s\_idle} \wedge \forall x \in [c : c+y+1] : ECU_u^x.abc.state \neq \texttt{startsnd} \\ \Rightarrow \forall t \in [e_u(c+2) + tp_{min} : e_u(x+3) + tp_{min}] : Out_{S,u}(t) = 1 \end{aligned}$$

By the premise of the induction step we can fulfill the premise of the induction hypothesis and conclude:

$$\forall t \in [e_u(c+2) + tp_{min} : e_u(c+y+3) + tp_{min}] : Out_{S,u}(t) = 1$$

Then, it remains to show:

$$\forall t \in [e_u(c+y+3) + tp_{min} : e_u(c+y+4) + tp_{min}] : Out_{S,u}(t) = 1$$

Since the ECU_u has an idle Send Unit and did not start a message broadcast in cycle $c+y$ and its send register contains the idle value '1' by the induction hypothesis, by the same argumentation as in the induction base, we argue:

$$\forall : t \in [e_u(c+y+3) + tp_{min} : e_u(c+y+4) + tp_{min}] : Out_{S,u}(t) = Out_{S,u}(e_u(c+y+3))$$

and the claims follows.

□

Note that Lemma 20 shows, that the analog output reacts with a two cycles delay to the changes of the Scheduler Automaton. The first delay comes from the fact, that the clock enable signal will be stored in the register $abc.dnbit$ before it clocks the send register S. The second delay cycle comes from the DRM model, since the output of an analog register at the clock edge $e(c)$ is equal to the input signal written to the register at the clock edge $e(c-1)$.

7.5.2 Post-round Correctness of Analog Send Register Outputs

In the previous section we have established a formal dependency between the behaviour of the Scheduler, Send Unit and the analog output of the send register S of the corresponding ECU. Basically, we have shown, that if the Send Unit is idle, then the send register contains idle bus value until the Scheduler of the ECU enters `startsnd` state, which starts a message broadcast.

In this section, we will use this result to show the correctness of recurrent synchronizations during the system run. This will be proven inductively, arguing simultaneously about the clock synchronization and correct schedule execution, since both properties hinge on each other. First, we will show that after the startup routine, the bus is free for the first synchronization. In the induction step, assuming that synchronization has worked in round r, we will show, that all ECUs will accomplish all slots of a round and will switch to a synchronization-waiting state. Then we will show, that no one of them will leave this state until the actual synchronization message will be sent by master. Since no ECUs are sending at that point, we can abstract the bus as a direct wire from the master to every receiver and use Lemma 10 to show the the synchronization in the beginning of round $r + 1$ will happen.

We will need one helper lemma. As mentioned in Section 7.1.3, we cannot show, that slot 0 starts before the master starts the first message transmission, because the first bit of this message transmission serves as a synchronization wich actually triggers the first slot on the slave ECU. However, we have to show, that the slave does not produce any bus activity during the master's message transmission. Since we know, that all slaves should be waiting in state `rcvwait` for the synchronization, and that the analog output is delayed by two cycles (Lemma 20), we have to show, that if the slave is in state `rcvwait`, then its Send Unit was idle in the previous four cycles and remains idle for at least T next cycles.

Lemma 21. *Let ECU_u be a slave ECU. If it is in state `rcvwait` in cycle c, then its analog bus output is idle for 4 cycles before and for at least T cycles after cycle c.*

$$abc_u^c.state = \texttt{rcvwait} \Rightarrow \forall t \in [e_u(c-2) + tp_{min} : e_u(c+T+1) + tp_{min}] : Out_{S,u}(t) = 1$$

Proof. First we show, that the Send Unit of ECU_u was idle in cycles $c - b$ for $b \in [1 : 4]$:

$$abc_u^{c-b}.sstate = \texttt{s\_idle} \tag{7.12}$$

By Scheduler construction, if an ECU is in state `rcvwait` in cycle c, then in cycle $c - b$ it could be in states `rcvwait`, `idle` or `Twait` (since an ECU stays for T cycles in `Twait`). In case of states `rcvwait` and `idle` the claim 7.12 follows immediately by Lemma 19. In case of state `Twait` it would mean, that ECU_u has finished the last slot $ns - 1$ of some round in the previous cycle. Hence, we show by hardware construction, that even if $send(ns - 1) = u$ holds, the transmission would end *off* cycles before ECU_u leaves the state `Twait`:

$$abc_u^{c-off}.sstate = \texttt{s\_idle}$$

Then we show, that in the remaining *off* cycles the state `startsnd` cannot be entered and, thus, by instantionation of the constant *off* (Section 7.1.4) the Send Unit would remain idle until cycle $c - b$ because $off > b$. Furthermore, since by assumption the ECU_u is in state `rcvwait` in cycle c, we can show by Scheduler Automaton construction, that it will not start a message broadcast for at least T cycles. Obviously, this holds because the slave being in state `rcvwait` has first to pass in the next cycles states `rcvwait` (possibly), `Twait` and finally `offwait` to reach the broadcast starting state `startsnd`. Alone in states `Twait` and `offwait` it will spend altogether T cycles. Hence, we have:

$$\forall x \in [c-4 : c+T] : abc_u^x.state \neq \texttt{startsnd}$$

Finally, we apply Lemma 20 for cycle $c - 4$ and $y := T + 1$ and the claim follows. □

Now we are ready to show that a slave will successfully synchronize with master and start all slots of all rounds.

Lemma 22 (Schedule Correctness Of Slaves for All Rounds)**.** *Every slave ECU_u with $u \in [0 : ns - 1]$ will start all slots of all rounds*

$$\forall r, s < ns : \Sigma_u(s)^{\alpha_u(r,s)}$$

Proof. We prove the lemma by Induction on round r.
Induction base: r = 0. We will start with showing, that the synchronization in the beginning of round 0 will take place by showing, that slot 0 of round 0 will be started on all slaves. Thus, by Definition 12 and 13 we have to show:

$$\begin{aligned} \Sigma_u(0)^{\alpha_u(0,0)} &\equiv \Sigma_u(0)^{cy_{u,0}(\alpha_0(0,0)+off+2)+\phi_{u,0}(\alpha_0(0,0)+off+2)+4} \\ &\equiv \Sigma_u(0)^{cy_{u,0}(t_0+off+2)+\phi_{u,0}(t_0+off+2)+4} \end{aligned}$$

By Assumption 8 we know, that ECU_u is in state `rcvwait`, when the master is in state `startsnd` in cycle $t_0 + off$:

$$abc_0^{t_0+off}.state = \texttt{startsnd} \ \wedge \ abc_u^{cy_{u,0}(t_0+off)}.state = \texttt{rcvwait}$$

Here we use the next affected cycle $cy_{u,0}(t_0 + off)$ to fix a cycle of a slave corresponding to cycle $t_0 + off$ of the master. To show, that the first synchronization will hapen, we will apply Lemma 10. However, we have to be able to resolve all of its premises first. For this, we first instantiate all variables fixed in that lemma:

- abc_0^j will be instantiated by $ECU_0.abc^j$;
- abc_u^j will be instantiated by $ECU_u.abc^j$;
- the master's cycle c will be instantiated by $t_0 + off$;
- the generation function $bvce_S$ of bit lists of clock enable signals will be instantiated by function $genCEs_0$;
- the generation function $bvin_S$ of bit lists of input signals will be instantiated by function $genINs_0$;
- clock enable signal $ce_{R,u}$ for register R of ECU_u will be instantiated by a constant function $(\lambda x.1)$.[6]

Then we resolve all premises:

(a) The first premise will be resolved as described in Section 7.2.

(b) In the second premise we have to show, that the master is in state `startsnd` in cycle $t_0 + off$ and its Send Unit is idle. By Assumption 8 we know, that the master starts slot 0 of round 0 in cycle t_0: $\Sigma_0(0)^{t_0}$. By Scheduler construction we easily conclude: $abc_0^{t_0+off}.state = \texttt{startsnd}$. By Lemma 19 we can conclude that in state `startsnd`, the Send Unit of every ECU is idle.

[6] This is sound because the clock enable signal of register R is always clocked and does not depend on any external computations.

(c) After all instantiations we have to show:

$$abc_0^{t_0+off+1}.dnbit = 1 \ \wedge \ \forall i \in [1:7] : abc_0^{t_0+off+i+1}.dnbit = 0$$

By discharging of premise (b) we know, that the master is in state `startsnd` in cycle $t_0 + off$ and its Send Unit is idle ($s\_idle = 1$). Now we can show by Scheduler Automaton (Figure 6.5) and output computation module construction (Figure 6.8), that in the next cycle a '1' will be written to the $dnbit$ register. We also can show, that the $abc_0.bitcon$ counter will be reset in cycle $c + 1$. In the same cycle the Send Unit will be triggered by active $startsnd$ signal and remains non-idle during the entire message transmission. Thus in the next 7 cycles (until $abc_0.bitcon$ reaches value '111') a zero will be sampled into register $abc_0.dnbit$.

(d) Proof of $(\lambda x.1)(t) = 1$ is obvious.

(e) The fifth premise requires that the shift register is filled with ones at the moment, when the master starts the broadcasting of the synchronization message:

$$abc_u^{cy_{u,0}(\alpha_0(0,0)+off+2)}.sh = 1^4$$

We make use of Assumption 8 and conclude, that in cycle $cy_{u,0}(\alpha_0(0,0))$ every slave is in state `rcvwait`, and that all previous states were `idle` or `rcvwait`. Using Lemma 19 we can show, that the Send Unit of all ECUs was idle until that cycle, thus, the send register of every slave contains a '1'. By Lemma 21 we know that the analog output of a slave will stay for at least T cycles idle after cycle $cy_{u,0}(\alpha_0(0,0))$. Moreover, by Scheduler construction we can conclude, that the master will never enter the state `startsnd` until cycle $t_0 + off$. Thus, by Lemma 19 and Lemma 20 we can show, that the master's analog output is idle until $e_0(\alpha_0(0,0) + off + 2) + tp_{min}$. Hence, by bus construction we can show that all slaves will sample only idle values until the start of the synchronization in cycle $cy_{u,0}(\alpha_0(0,0) + off + 2)$.

(f) The sixth premise requires that the receiver is in the state `rcvwait` and its Receive Unit is idle at the time when the master starts its broadcast. We use the same lemmas shown for the fifth premise and argue that since the shift register contained idle values only until the message broadcast, by synchronizer construction (Figure 6.11(b)) the signal $startedrcv$ was never activated. Hence, the Receive Unit was never started and the Scheduler could not leave state `rcvwait`.

(g) In the seventh premise we have to justify that the bus can be abstracted to a direct wire between the master and every slave for the time of the transmission of the first bit of the master's message:

$$\forall t \in (e_0(\alpha_0(0,0)+off+2)+tp_{min} : e_0(alpha_0(0,0)+off+9)+tp_{min}] : In_{R,u}(t) = Out_{S,0}(t)$$

We have already argued in the proof of the previous premise, that every slave will be in state `rcvwait` and its Send Unit is idle in cycle $cy_{u,0}(\alpha_0(0,0) + off + 2)$. By Lemma 21 we know that the slave's analog output is idle during time t in the following interval:

$$[e_u(cy_{u,0}(\alpha_0(0,0) + off + 2) - 2) + tp_{min} : e_u(cy_{u,0}(\alpha_0(0,0) + off + 2) + T + 1) + tp_{min}]$$

Hence, if we show that this interval includes the interval of the seventh premise, the premise can be discharged.

- Left boundary:

$$\begin{aligned}
& e_u(cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)-2)+tp_{min}- \\
& (e_0(\alpha_0(0,0)+\mathit{off}+2)+tp_{min}) \\
= \;& e_u(cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)-2)-e_0(\alpha_0(0,0)+\mathit{off}+2) \\
(\text{Definition 2}) \;= \;& e_u(cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)-1)-\tau_u- \\
& e_0(\alpha_0(0,0)+\mathit{off}+2) \\
(\text{Lemma 7}) \;\leq \;& e_0(\alpha_0(0,0)+\mathit{off}+2)+tp_{min}-th-\tau_u- \\
& e_0(\alpha_0(0,0)+\mathit{off}+2) \\
= \;& tp_{min}-th-\tau_u \\
(\text{Assumption 2}) \;\leq \;& 0
\end{aligned}$$

- Right boundary:

$$\begin{aligned}
& e_0(\alpha_0(0,0)+\mathit{off}+9)+tp_{min}- \\
& e_u(cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)+T+1)-tp_{min} \\
\text{Definition 2} \;= \;& e_0(\alpha_0(0,0)+\mathit{off}+2)+7\cdot\tau_0- \\
& (e_u(cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2))+(T+1)\cdot\tau_u) \\
\text{Definition 7} \;= \;& e_0(\alpha_0(0,0)+\mathit{off}+2)+7\cdot\tau_0- \\
& (e_0(\alpha_0(0,0)+\mathit{off}+2)+tp_{min}-th+(T+1)\cdot\tau_u) \\
\text{Assumption 2} \;\leq \;& 7\cdot\tau_0\leq T\cdot\tau_u \\
\text{Lemma 3} \;\leq \;& 0
\end{aligned}$$

(h) the last premise requires that beginning with cycle $cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)$ of slave ECU_u signal $reset\_ru$ remains inactive. By definition of signal $reset\_ru$ (page 62) we know that it gets active if the Scheduler is in state `Twait` and the cycle counter reaches $T-1$. Since we already know that every slave ECU is in state `rcvwait` in cycle $cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)$, by Scheduler Automaton construction, there is no way to activate signal $reset\_ru$ in the next 3 cycles.

Finally, Lemma 10 can be applied and we get for every slave the start of slot 0 in cycle $\alpha_u(0,0)$:

$$\Sigma_u(0)^{cy_{u,0}(\alpha_0(0,0)+\mathit{off}+2)+\phi_{u,0}(\alpha_0(0,0)+\mathit{off}+2)+4}=\Sigma_u(0)^{\alpha_u(0,0)}$$

Now we just apply Böhm's Lemma 11 and get for the remaining slots:

$$\forall s\in[1:ns-1]:\Sigma_u(s)^{\alpha_u(0,0)+tc+\mathit{off}+(s-1)\cdot T}$$

By Lemma 6 we conclude:

$$\forall s\in[1:ns-1]:\alpha_u(0,0)+tc+\mathit{off}+(s-1)\cdot T=\alpha_u(0,s)$$

And the claim of the induction base ($r=0$) follows.
Induction step: $r\to r+1$. By induction hypothesis we know, that all slaves will start all slots of round r:

$$\forall s<ns:\Sigma_u(s)^{\alpha_u(r,s)}$$

We have to show, that they will start all slots of the next round $r+1$:

$$\forall s<ns:\Sigma_u(s)^{\alpha_u(r+1,s)}$$

Basically we have to show that after all slaves have accomplished the last slot of round r, the synchronization before round $r+1$ will take place, by applying Lemma 10 again. However, in this case we will instantiate the master's cycle c by cycle $\alpha_0(r+1,0)+$ *off*, i.e., the cycle in which the master starts the broadcasting of the synchronization message of round $r+1$. All fixed variables of Lemma 10 can be instntiated exactly as in the base case of the induction, as well as all premises except for premises (e), (f) and (g) can be resolved identically to the proof of the induction base case. Thus, we will only deal with three remaining premises.

- Premise (e): the shift register of all ECUs is filled with ones:

$$\forall u \in [1 : ns-1] : abc_u^{cy_{u,0}(\alpha_0(r+1,0)+off+2)+\sigma_{u,0}(\alpha_0(r+1,0)+off+2)}.sh = 1^4$$

- Premise (f): every slave is waiting for synchronization and his Receive Unit is idle:

$$\forall u \in [1 : ns-1] : \quad abc_u^{cy_{u,0}(\alpha_0(r+1,0)+off+2)+\sigma_{u,0}(\alpha_0(r+1,0)+off+2)}.state = \texttt{rcvwait} \ \wedge \\ abc_u^{cy_{u,0}(\alpha_0(r+1,0)+off+2)+\sigma_{u,0}(\alpha_0(r+1,0)+off+2)}.rstate = \texttt{r\_idle}$$

- Premise (g): the input signal of every receive register is the output of the send register of the master during 7 cycles:

$$\forall u \in [1 : ns-1] : \\ \forall t \in (e_0(\alpha_0(r+1,0)+off+2)+tp_{min} : e_0(\alpha_0(r+1,0)+off+9)+tp_{min}] : \\ In_{R,u}(t) = Out_{S,0}(t)$$

Proof of Premise (e). First, by induction hypothesis we conclude that every slave ECU will reach the last slot of round r:

$$\forall u \in [1 : ns-1] : \Sigma_u (ns-1)^{\alpha_u(r,ns-1)}$$

By simple proof over the construction of the Scheduler Automaton, we can show that every ECU will also reach the end of the last slot:

$$\forall u : \quad abc_u^{\alpha_u(r,ns-1)+T}.state = \texttt{Twait} \ \wedge \ abc_u^{\alpha_u(r,ns-1)+T}.cycle = T-1 \ \wedge \\ abc_u^{\alpha_u(r,ns-1)+T}.slot = ns-1$$

This implies predicates $eqT^{\alpha_u(r,ns-1)+T}$ and $eqns^{\alpha_u(r,ns-1)+T}$ by their definitions. As depicted in Figure 6.5 every time the Scheduler leaves the state `Twait` (on active eqT signal), a signal $reset\_ru$ will be activated. By Figure 6.10 we can verify, that signal $reset\_ru$ resets the Receive Unit putting it into the `r_idle` state. By Figure 6.11(a) we also can verify, that the Redundancy Elimination module of the Receive Unit will be filled with ones on active $reset\_ru$ signal. Thus, formally we have in the next cycle, after the end of slot $ns-1$ is reached:

$$abc_u^{\alpha_u(r,ns-1)+T+1}.state = \texttt{rcvwait} \ \wedge \ abc_u^{\alpha_u(r,ns-1)+T+1}.rstate = \texttt{r\_idle} \ \wedge$$

$$abc_u^{\alpha_u(r,ns-1)+T+1}.sh = 1^4 \ \wedge \ abc_u^{\alpha_u(r,ns-1)+T+1}.\hat{R} = 1 \ \wedge \ abc_u^{\alpha_u(r,ns-1)+T+1}.R = 1$$

Since, the shift register $abc_u.sh$ will be filled with values sampled 2 cycles before on the bus (because the values go through two receive registers first), it is enough to show, that the ECU has received only ones 2 cycles before the next affected cycle plus delay cycle.

Hence, we show, that after the end of the last slot of round r in cycle $\alpha_u(r, ns-1)+T+1$ and until two cycles before the next cycle affected by the synchronization message for the next round in cycle $cy_{u,0}(\alpha_0(r+1,0)+\mathit{off}+2)+\sigma_{u,0}(\alpha_0(r+1,0)+\mathit{off}+2)$ all Receive Units sample only the idle value. For convinience, we abbreviate this time range of ECU_u as $PostRound_u$:

$$PostRound_u := [\alpha_u(r, ns-1)+T+1 : cy_{u,0}(\alpha_0(r+1,0)+\mathit{off}+2)+\sigma_{u,0}(\alpha_0(r+1,0)+\mathit{off}+2)-2]$$

And we have to show:

$$\forall u \in [1 : ns-1] : \forall x \in PostRound_u : abc_u^x.R = 1 \tag{7.13}$$

By our bus modeling in Section 4.4 we model the digital input of the receive register in cycle x as digitized bus value at clock edge $e_u(x)$:

$$abc_u^x.R = dig(bus, e_u(x))$$

Hence, if we show that the bus value contains the idle value at every clock edge $e_u(x)$ with $x \in PostRound$, then by Definition of the dig function (Definition 4.2), the equation 7.13 would follow. Thus, we show

$$\forall u \in [1 : ns-1] : \forall x \in PostRound_u : bus(e_u(x)) = 1$$

We can conclude this by bus construction if we show:

$$\forall x \in PostRound_u : \forall v \in [0 : ns-1] : Out_{S,v}(e_u(x)) = 1 \tag{7.14}$$

That is, the send register S of any ECU_v contains '1' at the time of clock edge x of arbitrary ECU_u. We show this as follows.

We make a case distinction on $sendl_v[0] = 1$, denoting whether ECU_v is the master.

1. **Case 1:** ECU_v is a slave. By induction hypothesis we can conclude, that ECU_v has started the last slot of round r: $\Sigma_v(ns-1)^{\alpha_v(r,ns-1)}$. We know that the Send Unit of ECU_v would be idle in cycle $\alpha_v(r, ns-1)+tc+\mathit{off}$ independendtly of whether it is a sender in slot $ns-1$ or not. If it was a receiver, its Send Unit will never be started. If it was a sender, its Send Unit will be started in cycle $\alpha_v(r, ns-1)+\mathit{off}$ and finishes tc cycles later by returning back to state `s_idle`. Hence, in both cases we can show:

   $$abc_v^{\alpha_v(r,ns-1)+tc+\mathit{off}}.sstate = \texttt{s\_idle} \ \wedge \ abc_v^{\alpha_v(r,ns-1)+tc+\mathit{off}}.state = \texttt{Twait} \tag{7.15}$$

   Moreover, we can easily show, that the Scheduler will stay another off cycles in state `Twait` until signal eqT gets active in cycle:

   $$\alpha_v(r, ns-1)+tc+\mathit{off}+\mathit{off} = \alpha_v(r, ns-1)+T$$

   Since this would be the end of slot $ns-1$ signal $eqns$ would also get active and by Scheduler construction, ECU_v would switch to state `rcvwait` in cycle $\alpha_v(r, ns-1)+T+1$. By Lemma 21 we know, that ECU_v will not enter the state `startsnd` for at least another T cycles. Summing up, we can show:

   $$abc_v^{\alpha_v(r,ns-1)+tc+\mathit{off}}.sstate = \texttt{s\_idle} \ \wedge$$
   $$\forall x \leq \mathit{off}+T : abc_v^{\alpha_v(r,ns-1)+tc+\mathit{off}+x}.state \neq \texttt{startsnd}$$

Applying this to Lemma 20 and the Definition of T we can conclude:

$$\forall t \in [e_v(\alpha_v(r, ns{-}1){+}tc{+}\mathit{off}{+}2){+}tp_{min} : e_v(\alpha_v(r, ns{-}1){+}2{\cdot}T{+}3){+}tp_{min}] : Out_{S,v}(t) = 1$$

Now to show 7.14 we only have to show:

$$\begin{aligned}&\forall x \in PostRound_u :\\ &e_u(x) \in [e_v(\alpha_v(r, ns-1) + tc + \mathit{off} + 2) + tp_{min} : e_v(\alpha_v(r, ns-1) + 2 \cdot T + 3) + tp_{min}]\end{aligned}$$

- **Lower bound**. We show for the smallest $x \in PostRound_u$:

  $$e_v(\alpha_v(r, ns-1) + tc + \mathit{off} + 2) + tp_{min} \leq e_u(\alpha_u(r, ns-1) + T + 1)$$

  Proof.

  $$\begin{aligned}
  & & & e_v(\alpha_v(r, ns-1) + tc + \mathit{off} + 2) + tp_{min} - \\
  & & & e_u(\alpha_u(r, ns-1) + T + 1) \\
  & (\text{Lemma 6}) & = & e_v(\alpha_v(r, 0) + tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) + tp_{min} - \\
  & & & e_u(\alpha_u(r, 0) + tc + \mathit{off} + (ns-1) \cdot T + 1) \\
  & (\tau_u > tp_{min}) & \leq & e_v(\alpha_v(r, 0) + tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) - \\
  & & & e_u(\alpha_u(r, 0) + tc + \mathit{off} + (ns-1) \cdot T) \\
  & (\text{Definition } 2) & = & e_v(\alpha_v(r, 0)) + (tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) \cdot \tau_v - \\
  & & & e_u(\alpha_u(r, 0)) - (tc + \mathit{off} + (ns-1) \cdot T) \cdot \tau_u \\
  & & = & e_v(\alpha_v(r, 0)) - e_u(\alpha_u(r, 0)) + \\
  & & & (tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) \cdot \tau_v - \\
  & & & (tc + \mathit{off} + (ns-1) \cdot T) \cdot \tau_u \\
  & (\text{Lemma 12}) & \leq & (tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) \cdot \tau_v - \\
  & & & (tc + \mathit{off} + (ns-1) \cdot T - 3) \cdot \tau_u \\
  & (\text{Lemma 1}) & \leq & (tc + \mathit{off} + (ns-2) \cdot T + \mathit{off} + tc + 2) \cdot (1 + \Delta) \cdot \tau_u - \\
  & & & (tc + \mathit{off} + (ns-1) \cdot T - 3) \cdot \tau_u \\
  & & = & (((ns-1) \cdot T + tc + 2) \cdot (1 + \Delta) - (tc + \mathit{off} + (ns-1) \cdot T - 3)) \cdot \tau_u \\
  & & = & (((ns-1) \cdot T + tc + 2) \cdot \Delta + 5 - \mathit{off}) \cdot \tau_u \\
  & (\text{Def. } T) & \leq & (ns \cdot T \cdot \Delta + 5 - \mathit{off}) \cdot \tau_u \\
  & (\text{Def. } \mathit{off}) & \leq & 0
  \end{aligned}$$

- **Upper bound**. We show for the largest $x \in PostRound_u$:

  $$\begin{aligned}e_u(cy_{u,0}(\alpha_0(r+1, 0) + \mathit{off} + 2) + \sigma_{u,0}(\alpha_0(r+1, 0) + \mathit{off} + 2) - 2)\\ \leq e_v(\alpha_v(r, ns-1) + 2 \cdot T + 3) + tp_{min}\end{aligned}$$

  Since $0 \leq tp_{min}$ and $\sigma_{u,0}(\alpha_0(r+1, 0) + \mathit{off} + 2) \leq 1$ we can simplify this expression:

  $$e_u(cy_{u,0}(\alpha_0(r+1, 0) + \mathit{off} + 2) - 1) \leq e_v(\alpha_v(r, ns-1) + 2 \cdot T + 3)$$

  By Lemma 7 we conclude:

  $$e_0(\alpha_0(r+1, 0) + \mathit{off} + 2) + tp_{min} - th \leq e_v(\alpha_v(r, ns-1) + 2 \cdot T + 3)$$

By Lemma 6 we have:

$$e_0(\alpha_0(r+1,0)+\mathit{off}+2)+tp_{min}-th \leq e_v(\alpha_v(r,0)+tc+\mathit{off}+ns\cdot T+3)$$

By Definition 12 we have:

$$e_0(\alpha_0(r+1,0)+\mathit{off}+2)+tp_{min}-th$$

$$\leq e_v(cy_{v,0}(\alpha_0(r,0)+\mathit{off}+2)+\phi_{v,0}(\alpha_0(r,0)+\mathit{off}+2)+4+tc+\mathit{off}+ns\cdot T+3)$$

Since $\phi_{v,0}(\alpha_0(r,0)+\mathit{off}+2) \leq 1$ and by Definition 2 we get:

$$e_0(\alpha_0(r+1,0)+\mathit{off}+2)+tp_{min}-th \leq e_v(cy_{v,0}(\alpha_0(r,0)+\mathit{off}+2))+(tc+\mathit{off}+ns\cdot T+7)\cdot\tau_v$$

By Lemma 7:

$$e_0(\alpha_0(r+1,0)+\mathit{off}+2)+tp_{min}-th \leq e_0(\alpha_0(r,0)+\mathit{off}+2)+tp_{min}-th+(tc+\mathit{off}+ns\cdot T+7)\cdot\tau_v$$

Which simplifies to:

$$e_0(\alpha_0(r+1,0)+\mathit{off}+2) \leq e_0(\alpha_0(r,0)+\mathit{off}+2)+(tc+\mathit{off}+ns\cdot T+7)\cdot\tau_v$$

By Lemma 5, Definitions 2 and 3, Assumption 2 we have:

$$e_0(\alpha_0(r,0)+\mathit{off}+2)+ns\cdot T\cdot\tau_0 \leq e_0(\alpha_0(r,0)+\mathit{off}+2)+(tc+\mathit{off}+ns\cdot T+7)\cdot\tau_v$$

Which simplifies to:

$$ns\cdot T\cdot\tau_0 \leq (ns\cdot T+tc+\mathit{off}+7)\cdot\tau_v$$

And can be shown as follows:

$$\begin{aligned} & ns\cdot T\cdot\tau_0-(ns\cdot T+tc+\mathit{off}+7)\cdot\tau_v \\ (\text{Lemma 1}) \quad \leq \quad & ns\cdot T\cdot(1+\Delta)\cdot\tau_v-(ns\cdot T+tc+\mathit{off}+7)\cdot\tau_v \\ = \quad & (ns\cdot T\cdot\Delta-tc+\mathit{off}+7)\cdot\tau_v \\ (\text{Definition of } \mathit{off}) \quad \leq \quad & 0 \end{aligned}$$

2. **Case 2:** ECU_v is master. Since the master ECU sends only in cycle 0, we can easily show by Lemma 16 that it will start the last slot of the previous round:

$$\Sigma_0(ns\ -\ 1)^{\alpha_0(r,ns-1)}$$

and it will skip the state `startsnd` for T cycles, since it is a receiver in the last slot. By Lemma 19 we also can conclude, that the Send Unit is idle in cycle $\alpha_0(r,ns-1)$. However, it will enter a new round right in the next cycle and start a message transmission in cycle $\alpha_0(r,ns-1)+T+\mathit{off}$ which is cycle $\alpha_0(r+1,0)+\mathit{off}$ by Lemma 5 and Definition 3. Thus, by Definition 11 and Lemma 20 we conclude:

$$\forall t \in [e_0(\alpha_0(r,ns-1)+2)+tp_{min} : e_0(\alpha_0(r,ns-1)+T+\mathit{off}+2)+tp_{min}] : Our_{S,0}=1$$

As in the slave-case, we have to show:

$$\forall x \in PostRound_u :$$
$$e_u(x) \in [e_0(\alpha_0(r,ns-1)+2)+tp_{min} : e_0(\alpha_0(r,ns-1)+T+\mathit{off}+2)+tp_{min}]$$

- **Lower bound.** We show for the smallest $x \in PostRound$:

$$
\begin{aligned}
& e_0(\alpha_0(r, ns-1)+2) + tp_{min} - e_u(\alpha_u(r, ns-1) + T + 1) \\
(\text{Lemma 6 and 5}) \quad = \quad & e_0(\alpha_0(r,0) + (ns-1)\cdot T + 2) + tp_{min} - \\
& e_u(\alpha_u(r,0) + tc + \mathit{off} + (ns-1)\cdot T + 1) \\
(\text{Definition 12}) \quad \leq \quad & e_0(\alpha_0(r,0) + (ns-1)\cdot T + 2) + tp_{min} - \\
& (e_u(cy_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + \phi_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + \\
& 4 + tc + \mathit{off} + (ns-1)\cdot T + 1)) \\
(1 \geq \phi_{u,0}(x)) \quad \leq \quad & e_0(\alpha_0(r,0) + (ns-1)\cdot T + 2) + tp_{min} - \\
& e_u(cy_{u,0}(\alpha_0(r,0) + \mathit{off} + 2) + 4 + tc + \mathit{off} + (ns-1)\cdot T + 1) \\
(\text{Definition 2}) \quad = \quad & e_0(\alpha_0(r,0) + \mathit{off} + 2) + ((ns-1)\cdot T - \mathit{off})\cdot \tau_0 + tp_{min} - \\
& (e_u(cy_{u,0}(\alpha_0(r,0) + \mathit{off} + 2)) + (4 + tc + \mathit{off} + (ns-1)\cdot T)\cdot \tau_u) \\
(\text{Definition 7}) \quad \leq \quad & e_0(\alpha_0(r,0) + \mathit{off} + 2) + ((ns-1)\cdot T - \mathit{off})\cdot \tau_0 + tp_{min} - \\
& (e_0(\alpha_0(r,0) + \mathit{off} + 2) + tp_{min} - th + \\
& (4 + tc + \mathit{off} + (ns-1)\cdot T)\cdot \tau_u) \\
= \quad & ((ns-1)\cdot T - \mathit{off})\cdot \tau_0 + th - \\
& (4 + tc + \mathit{off} + (ns-1)\cdot T)\cdot \tau_u \\
(\tau_u > th) \quad \leq \quad & ((ns-1)\cdot T - \mathit{off})\cdot \tau_0 - (3 + tc + \mathit{off} + (ns-1)\cdot T)\cdot \tau_u \\
(\text{Lemma 1}) \quad \leq \quad & (((ns-1)\cdot T - \mathit{off})\cdot (1+\Delta) - 3 - tc - \mathit{off} - (ns-1)\cdot T)\cdot \tau_u \\
= \quad & (((ns-1)\cdot T - \mathit{off})\cdot \Delta - 3 - 2\cdot \mathit{off} - tc)\cdot \tau_u \\
(\text{Definition of } \mathit{off}) \quad \leq \quad & 0
\end{aligned}
$$

- **Upper bound.** For the upper bound we have:

$$
\begin{aligned}
& e_u(cy_{u,0}(\alpha_0(r+1,0) + \mathit{off} + 2) + \sigma_{u,0}(\alpha_0(r+1,0) + \mathit{off} + 2) - 2) \\
& \qquad \leq e_0(\alpha_0(r, ns-1) + T + \mathit{off} + 2) + tp_{min}
\end{aligned}
$$

  Since $\sigma_{u,0}(\alpha_0(r+1,0) + \mathit{off} + 2) \leq 1$, we simplify the inequation above:

$$e_u(cy_{u,0}(\alpha_0(r+1,0) + \mathit{off} + 2) - 1) \leq e_0(\alpha_0(r, ns-1) + T + \mathit{off} + 2) + tp_{min}$$

  As above, by Lemma 7 we get:

$$e_0(\alpha_0(r+1,0) + \mathit{off} + 2) + tp_{min} - th \leq e_0(\alpha_0(r, ns-1) + T + \mathit{off} + 2) + tp_{min}$$

  By Lemma 5 and Definition 3 we reformulate the inequation

$$e_0(\alpha_0(r, ns-1) + T) + (\mathit{off} + 2)\cdot \tau_0 - th \leq e_0(\alpha_0(r, ns-1) + T) + (\mathit{off} + 2)\cdot \tau_0$$

  And the claim follows by Assumption 2 ($th \geq 0$).

Thus, we have verified, that the Premise (e) of Lemma 10 is fullfiled for instantiation of cycle c by $\alpha_0(r+1,0) + \mathit{off} + 2$. Proof of Premise (f) becomes obvious due to the already proven claim 7.13 and Premise (e). First, we have shown in the proof of Premise (e), that the Receive Unit will be reset at the end of the last slot. Since the receive register R has sampled only ones starting from the last slot and up to the next cycle affected by the synchronization plus delay, then by

a simple hardware proof, we can show, that the majority voter never voted for 0 during that time, and, hence, signal $startedrcv$ could not get active. Thus, every receiver's Receive Unit remained idle and, hence, the receiver could not leave the `rcvwait` state.

Premises (g) and (h) can be proven along the lines of the proof in the induction base case. Since we have shown in Premise (g), that all slaves are in state `rcvwait` (or in `Twait` ending the last slot) shortly before the synchronization, we apply Lemma 21 to show, that all slaves will have an idle value in their send registers for the next T cycles. Thus, we can apply Lemma 10 for cycle $\alpha_0(r+1,0)+\mathit{off}+2$ and finally get for all slaves:

$$\forall u \in [1 : ns-1] : \Sigma_u^{\alpha_u(r+1,0)}$$

Then, applying Lemma 11, the claim of round-based schedule correctness follows.

□

7.6 Bus Control Correctness

Until now we have shown, that in every round the synchronization signal transmitted by the master, can be successfully received by all slaves. Afterwards, all slaves will execute a fixed schedule consisting of ns slot. Moreover, we know, that all non-zero slots overlap during a message transmission. In this section we will show first, that in every slot where the ECU acts as a receiver, it will produce an idle bus output. Second, we will show, that despite the fact, that slot 0 will start after the start of the very first transmission, all slaves produce an idle output in the first slot. Then combining all results we will show, that in every slot of every round every non-sending ECU produces an idle bus output. Then, by bus construction we can show, that the bus contains the value of the send register of the sending ECU.

Lemma 23 (Receiver Skips `startsnd` State)**.** *Let ECU_u be an abritrary ECU and $s \in [0 : ns-1]$ any slot, s.t. $send(s) \neq u$. Then we show, that during the entire slot ECU_u does not enter Scheduler state `startsnd`:*

$$\forall u, r, s \in [0 : ns-1] : send(s) \neq u \Rightarrow \forall x \in [\alpha_u(r,s) : \alpha_u(r,s)+T-1] : abc_u^x.state \neq \texttt{startsnd}$$

Proof. We split cases on slot s.

- **Case 1:** $s = 0$. First we can conclude that $u \neq 0$, i.e., we consider a slave, because we know:
$$sendl[0] = 0 = send(0)$$
By Lemma 22 we know $\Sigma_u(0)^{\alpha_u(r,0)}$, i.e., by Definition 11:
$$abc_u^{\alpha_u(r,0)}.state = \texttt{Twait} \ \wedge \ abc_u^{\alpha_u(r,0)}.cycle = \mathit{off} \ \wedge \ abc_u^{\alpha_u(r,0)}.slot = 0$$
By Scheduler Automaton construction we know, that the cycle counter will be incremented in every cycle as long as the bus controller stays in state `Twait` and it will leave it after $T - \mathit{off} - 1$ cycles. Then, since the slot counter was set to zero in the beginning of the slot 0 on the transition from state `rcvwait` to `Twait`, it will be incremented in the next cycle. Since we assume $ns > 1$, we clearly do not have

the number ns in the slot counter, so the slave ECU will switch to state off in cycle $\alpha_u(r,0)+T-\mathit{off}$. As before, it will increment the cycle counter in every cycle and it will stay in that state during $\mathit{off}-1$ cycles. Summing up all cycles of slot 0 and additional off cycles where ECU_u was in states `Twait` and `offwait` we get:

$$\alpha_u(r,0)+T-1-\mathit{off}+\mathit{off}=\alpha_u(r,0)+T-1$$

- **Case 2:** $s>0$. By similar argumentation as above, we have:

  $$abc_u^{\alpha_u(r,s)}.state=\texttt{offwait}\ \wedge\ abc_u^{\alpha_u(r,s)}.cycle=0\ \wedge\ abc_u^{\alpha_u(r,s)}.slot=s$$

  Thus, we simply show, that ECU_u leaves the state `offwait` in $\mathit{off}-1$ cycles, then it will skip `startsnd` because we have assumed $send(s)\neq u$. When ECU_u enter `Twait` its cycle counter shows off cycles. In state `Twait` it will stay until the cycle signals the number $T-1$. That is, it will remain there for $T-\mathit{off}-1$ cycles. Hence, ECU_u needs

  $$\mathit{off}+T-\mathit{off}-1=T-1$$

  cycles for slot s. Since it will skip the state `startsnd`, the claim follows.

□

Now we are ready to show the correctness of bus contention control during every transmission.

Theorem 3 (Bus Contention Control)**.** *During all transmission times of all slots of all round, the bus value is the value of the analog output of the send register of* $ECU_{send(s)}$*:*

$$\forall r,s:\forall t\in[\alpha_{send(s)}(r,s)+\mathit{off}+2:\alpha_{send(s)}(r,s)+\mathit{off}+tc+2]:bus(t)=Out_{S,send(s)}$$

Proof. To show the claim we fix an arbitrary receiver u in slot s with $u\neq send(s)$. We make a case distinction on slot number s.

- **Case 1:** $s=0$. By Lemma 22 we know $\Sigma_u(0)^{\alpha_u(r,0)}$. By Scheduler construction we show, that ECU_u was in state `rcvwait` in cycle $\alpha_u(r,0)-1$. Then we apply Lemma 21 and show that the analog output of ECU_u will be kept in the send register during the entire transmission time in slot 0.

- **Case 2:** $s>0$. By Lemma 23 we know:

  $$\forall x\in[\alpha_u(r,s):\alpha_u(r,s)+T-1]:abc_u^x.state\neq\texttt{startsnd}$$

  By Lemma 22 we know, that ECU_u will start slot s in cycle $\alpha_u(r,s)$ and by Definition 11 and Lemma 19 we can conclude:

  $$abc_u^{\alpha_u(r,s)}.sstate=\texttt{s\_idle}$$

  Applying Lemma 20 for cycle $\alpha_u(r,s)$ we get:

  $$\forall t\in[e_u(\alpha_u(r,s)+2)+tp_{min}:e_u(\alpha_u(r,s)+T+2)+tp_{min}]:Out_{S,u}(t)=1$$

  Finally, the claim follows from Lemma 13.

□

CHAPTER

EIGHT

VERIFICATION OF MESSAGE TRANSMISSION

In the previous chapter we have verified, that during the entire transmission time the bus value is the value of the send buffer of the sending ECU. After we have proven the abstraction of the bus to a direct wire, the proof of message transmission can be boiled down to hardware proofs of serial interfaces. We sketch the remaining arguments in this section.

Note that in this chapter we will refer to the j'th byte of buffer $RB(i)$ or $SB(i)$ by $RB(i)[j]$ or $SB(i)[j]$. In the real implementation, this is more complicated and as depicted in Figures 6.12(a) and 6.12(b), the buffers are implemented as RAMs. Hence, to read out or to write a byte, an reading/writing access has to be set up.

Schmaltz [Sch07] has shown in the following Lemma, that under the same assumptions as in Böhm's Lemma 10, i.e., if the Receive Unit is waiting for message reception and the directly connected send unit is starting to broadcast a message consisting of l bytes, then all messages will be transferred correctly from the send buffer of the sending ECU to the $Byte$ shift register (see Figure 6.11(a)) of the receiving ECU.

Lemma 24 (High-Level Message Transmission Correctness)**.** *Fix following constants:*

- abc_v^j *as the state of the sender ECU in some cycle* j*;*
- abc_u^j *as the state of an arbitrary slave ECU with* $0 < u < \mathcal{N}$*;*
- $c \in \mathbb{N}$ *as cycle of sender ECU;*
- $\xi_u = cy_{u,v}(c+2)$ *as receiver cycle;*
- $\xi_u^j = cy_{u,v}(c+18+80 \cdot j)$ *as receiver cycles;*

Premises:

(a) In all cycles of the system run, the reset signal stays disabled and all configuration registers containt the correct configuration parameter:

$$\forall c, i \in \{0, u\} \quad : \quad reset^c = 0 \wedge$$

$$abc_i^c.CR.ns = bin_6(ns) \ \wedge \ abc_i^c.CR.l = bin_{10}(l) \ \wedge$$
$$abc_i^c.CR.off = bin_{32}(off) \ \wedge \ abc_i^c.CR.T = bin_{32}(T) \ \wedge$$
$$abc_i^c.CR.iwait = bin_{32}(iwait) \ \wedge \ abc_i^c.CR.sendl = sendl_i$$

(b) The master is in synchronization starting state in cycle c and it will skip this state in the next tc cycles:

$$abc_v^c.state = \texttt{startsnd} \ \wedge \ \forall x \leq tc-1 : abc_v^{c+x}.state \neq \texttt{startsnd}$$

(c) The the clock enable signal of R is assumed to be stable around each cycle edge:

$$\forall i : \forall t \in [e_v(i) - ts : e_v(i) + th] : ce_{R,u}(t) = 1$$

(d) The shift register is filled with ones in cycle $\xi + \sigma_{u,v}(c+2)$:

$$abc_u^{\xi+\sigma_{u,v}(c+2)}.sh = 1^4$$

(e) The receiver is in synchronization-awaiting state, its send unit is idle:

$$abc_u^{\xi+\sigma_{u,v}(c+2)}.rstate = \texttt{idle}$$

(f) The input signal of R is the output signal of S during $n+1$ local cycles of the master:

$$\forall t \in (e_v(c+2) + tp_{min} : e_v(c+2+tc) + tp_{min}] : In_{R,u}(t) = Out_{S,v}(t)$$

Then: *every byte will be transmitted correctly from the send buffer of the sender to the shift register of the receiver:*

$$\begin{aligned} \forall i \leq l : \quad & abc_u^{\xi_u^j+79}.Byte = abc_v.SB(s \bmod 2)[i]^{c+18+80\cdot j} \quad \vee \\ & abc_u^{\xi_u^j+80}.Byte = abc_v.SB(s \bmod 2)[i]^{c+18+80\cdot j} \quad \vee \\ & abc_u^{\xi_u^j+81}.Byte = abc_v.SB(s \bmod 2)[i]^{c+18+80\cdot j} \end{aligned}$$

Thus, Schmaltz has shown, that if both serial interfaces are initialized and ready for the message transmission and the transmission starts in sender's cycle c, then every byte of the transmitted message will be transmitted correctly into the receiver's shift register $abc_u.Byte$. Note, that as in Lemmas 8, in the original version of Lemma 24 a permanent connection (premise (f)) between the send and receive registers was required (see Section 4.3.2). This premise was corrected by restricting of the connection time to the length of one transmission lasting tc sender cycles.

We will sketch the proof idea. The detailed proof description can be found in [Sch07].

Proof Sketch. Every byte will be transmitted together with a leading 2-bit sync edge BSS by the sender bit for bit. As we know from the message protocol construction, every bit will be replicated 8 times, hence, the transmission of one byte takes 80 cycles. All sampled bits go through the Redundancy Elimination module (Figure 6.11(a)), where the majority voter checks the last 5 sampled bits and outputs the most frequently occurring bit among them. The Synchronizer circuit (Figure 6.11(b)) counts cycles in

the 3-bit counter $bitcon$ and outputs a $strobe$ signal every 8 cycles. This signal will be used by the shift register $Byte$ as write enable signal. Hence, every time $strobe$ is active, the voted bit will be sampled into $Byte$ and the oldest bit will be overwritten with the previously sampled bit.

During these 80 cycles the clocks of both ECUs can drift by at most one cycle by Lemma 3. However this only means, that the receiver will either not recognize one bit on the bus if it is too slow, or it will recognize one sent bit as two bits, if it is too fast. This error will be smoothed out by the majority voter. Thus, the receiver will receive every transmitted byte either one cycle earlier (79 cycles), in time (80 cycles) or one cycle later (81 cycles).

Furthermore, by the Synchronizer circuit construction 6.11(b) we can verify, that if the voted bit changes from 1 to 0 (falling edge) and the send unit automaton is in state `BSS1`, i.e., the sync edge of a new byte was recognized, the cycle counter $abc_u.bitcon$ will be reset to adjust the receiver automaton to the master's view of the protocol flow. The proof is done by induction on length of the message l in bytes. □

This lemma was proven before Böhm has started his work. Since he used the same semantcis (local bus controller traces) and has adopted the same assumptions, Lemma 24 has had the same wrong assumption as Lemma 10, requiring $\hat{R}^{\xi+1} = 1$. Besides this, the lemma has previously used another wrong assumption making the proof valid only for cases if no drifts occur within 80 cycles between sender and receiver clocks. As in the case of previous bugs, since Lemma 24 was never applied, the error remained undiscovered until we have used it.

Schmaltz has provided an improved version on our demand.

Finally, we show the overall message transmission correctness and receive buffer stability during the consequent slot, as well as during the time after the last slot and until the start of the next round.

Theorem 4 (Overall Message Transmission Correctness)**.**

$$
\begin{aligned}
\forall u, r, s \quad : \quad & ECU_{send(s)}^{\alpha_{send(s)}(r,s)}.sb(s \bmod 2) = ECU_u^{\alpha_u(r,s)+T+1}.rb((s+1) \bmod 2) \wedge \\
& \forall x \in [\alpha_u(r,s) : \alpha_u(r,s) + T] : \\
& ECU_u^{\alpha_u(r,s)}.rb((s+1) \bmod 2) = ECU_u^x.rb((s+1) \bmod 2) \wedge \\
& \forall x \in [\alpha_u(r, ns-1) + T + 1 : \alpha_u(r+1, 0)] : \\
& ECU_u^{\alpha_u(r,ns-1)+T}.rb((s+1) \bmod 2) = ECU_u^x.rb((s+1) \bmod 2)
\end{aligned}
$$

Proof. First we show the correctness of the message transmission from the send buffer of the sender to the receive buffer of the receiver. We apply Lemma 24 for senders cycle $\alpha_{send(s)}(r,s) + off$, since this is the transmission start cycle for every ECU. We resolve Premise (a) and (c) as in the case of Lemma 10.

Premise (b) scan be shown by Scheduler automaton construction. We know from previous lemmas that every ECU will start slot s: $\Sigma_{send(s)}(s)^{\alpha_{send(s)(r,s)}}$. By Definition 11 we can conclude that $ECU_{send(s)}$ is in state `offwait` in the beginning of the slot s. Then by scheduler construction and $sendl[send(s)] = 1$ we show, that it will enter slot `startsnd` in off cycles. Obviously, by automaton construction we can easily show, that $ECU_{send(s)}$ will not enter this state during the message transmission, staying only in state `Twait`.

Premises (d) and (e) are shown for slot $s = 0$ exactly as in the case of Lemma 10. For slot $s > 0$ both assumption are shown in similar manner, as in Lemma 22 where we have

shown that the Receive Unit remains idle until the synchronization message. Here we show, that the slot start cycle $\alpha_u(r,s)$ starts directly after the last cycle of slot $s-1$. By scheduler construction we know, that every slot ends with the active eqT signal which resets the Receive Unit. Then we show, that from the slot start and until the message transmission start no ECU produces any bus activity because of:

- slaves do not produce any bus activity for the entire slot by Lemmas 23 and 20;
- the master behaves as a non-sending slave during all non-zero slots.

Finally, premise (g) is given by Theorem 3.
Hence, we get for $\xi_u^j = cy_{u,send(s)}(\alpha_{send(s)}(r,s) + \mathit{off} + 18 + 80 \cdot j)$:

$$\begin{aligned} \forall i \leq l: \quad & abc_u^{\xi_u^j+79}.Byte = abc_{send(s)}.SB(s \bmod 2)(i)^{\alpha_{send(s)}(r,s)+18+80\cdot j} \quad \vee \\ & abc_u^{\xi_u^j+80}.Byte = abc_{send(s)}.SB(s \bmod 2)[i]^{\alpha_{send(s)}(r,s)+18+80\cdot j} \quad \vee \\ & abc_u^{\xi_u^j+81}.Byte = abc_{send(s)}.SB(s \bmod 2)[i]^{\alpha_{send(s)}(r,s)+18+80\cdot j} \end{aligned}$$

As next we show two following hardware lemmas. The send buffer with index $s \bmod 2$ stays stable during the entire slot s:

$$\begin{aligned} \forall \quad x \in \quad & [\alpha_{send(s)}(r,s) : \alpha_{send(s)}(r,s) + T] : \\ & abc_{send(s)}^{\alpha_{send(s)}}.SB(s \bmod 2)_{s \bmod 2} = abc_{send(s)}^{x}.SB(s \bmod 2)_{s \bmod 2} \end{aligned}$$

This is simple, because by Send Buffer construction (Figure 6.12(a)), the parity bit $p = s \bmod 2$ prevents writing to the send buffer, exposed to the bus. We show that the content of the shift register $Byte$ will be written to the Receive Buffer in one of three possible cycles $\xi_u^i + 80 + \omega$ with $\omega \in \{-1, 0, 1\}$. This is shown by correctness of Receive Buffer WE Control. We show the correctness of Address Computation module and conclude for each possible ω:

$$\begin{aligned} \forall \quad x \in \quad & [cy_{u,send(s)}(\xi_u^i + 80 + \omega : \alpha_u(r,s) + T] : \\ & abc_u^{\alpha_u(r,s)+T}.RB(s \bmod 2)[j] = abc_u^{x}.RB(s \bmod 2)[j] \end{aligned}$$

Finally, we have to show, that the value of the receive buffer exposed to the bus in cycle $\alpha_u(r,s) + T$ is equal to the value of the receive buffer exposed to the processor in the next cycle $\alpha_u(r,s) + T + 1$:

$$abc_u^{\alpha_u(r,s)+T}.RB(s \bmod 2) = abc_u^{\alpha_u(r,s)+T+1}.RB((s+1) \bmod 2)$$

Basically, this statement would follow if we can show, that the parity of the slot changes in cycle $\alpha_u(r,s) + T$ and the receive buffer will not be changed. The latter statement is easy to show by the construction of the Receive Unit and the computation of the write enable signal of the receive buffer, which will be acivated only if the Receive Unit automaton is in state `byte[7]`.

To show that the slot parity changes in cycle $\alpha_u(r,s) + T$, we consider two cases.

1. $s < ns - 1$: in this case, we are at the end of slot s, but not at the end of the round r. By the Scheduler construction (Figure 6.5), signal $incslot$ will be activated at the transition from the state `Twait` to `offwait`. Obviously, the incrementation of the slot counter leads to the flipping of the least significant bit which is the parity bit.

2. $s = ns - 1$. By Assumption 7, the number of slots ns is an even number. Since the slot counter counts from 0, $ns - 1$ is the number of the last slot in the end of a round. Hence, $ns - 1$ is odd and its least significant bit (the parity bit) is 1. In case that the last slot of a round has ended (slot counter has reached $ns - 1$), a transition from the state `Twait` to `rcvwait` will be taken and the slot counter will be set to 0.

Thus, the parity flag changes in all cases and the proof of the message transmission (the first statement) follows.

To show that the receive buffer exposed to the processor (the receive buffer with index $s + 1 \bmod 2$) does not change its content, we have proven by the Scheduler automaton, that the slot counter stays stable during every slot and between the last slot of a round and the new round. Stable slot counter value implies that its last bit denoting the slot parity stays stable too. By the Construction of the receive buffer (Figure 6.12(b)) we could conclude, that the write signal of the receive buffer exposed to the processor remains disable during the entire slot. This proofs the last two statements of the claim. □

CHAPTER

NINE

CONCLUSION

9.1 Summary

We presented results on verification of time-triggered real-time bus system, which was inspired by the FlexRay standard. The network consists of electronic control units interconnected by a bus. We have verified the soundness of startup routine assumptions, the clock synchronization, the correct TDMA scheduling and a low-level bit transmission. Finally, consolidating these results, we have shown the correctness of recurrent message broadcasts during a system run. In contrast to most of previous research, we not only show the algorithmic correctness of protocols, but we also provide justified models linking these protocols to a concrete gate-level hardware.

The bus controller implementation consists of about 1800 lines of code. The ECU implementation has been automatically translated to Verilog directly from formal Isabelle/HOL hardware descriptions and has been synthesized with the Xilinx ISE software. The implementation which consists of several FPGA boards interconnected into a prototype of a distributed hardware network was reported in [End09].

The entire formal proof [MSB] consists of 900 lemmas and theorems, and of 426 definitions, which all together comprise about 66000 lines. About 43% of lines account for the (improved) proofs of previous results.

9.2 Discussion

During the presented verification work, several bugs were discovered in the hardware implementation, as well as in the formulation of lemmas in previous verifications. It turns out that formal verification of theorems, which are intended to be used as a lemmas of other theorems later, often results in assumptions which cannot be easily instantiated. This seems to happen due to lack of overview about the complete behavior and requirements of upper levels. As a result we have discovered in previous verified theories (cf. Section 7.1 and Subsection 4.3.3) too strong assumption (e.g., unnecessary quantifications), inconsistent assumptions (due to syntactic mistakes), wrong assumption, different formalizations of same semantics, and so on. Fixing these issues led to new unproven cases in existing proofs, which were studied and adjusted to new assumptions. This was one of the most time consuming tasks.

The time spent on the verification can be split into:

- Verification of the low-level message transmission (Theorem 1) and hardware verification of serial interfaces (Lemma 24) which took Julien Schmaltz about one person-year.
- Verification of the Scheduler hardware (Lemma 11), Synchronization (Lemma 10) and schedule timings (Lemma 13) took Peter Böhm one person-year [ABK08].
- Remaining verifications, extensions of previous results, adjusting of proofs and consolidation into one theory, described in this work, took another 2.5 person-years.

All three results were achieved in isolation and not in a team work. This had a huge disadvantage during the consolidation of previous results. Every time a problem was discovered during the consolidation work, first a study of the problem nature was necessary. Unfortunately, the impossibility to collectively discuss such problems did cost a huge amount of time, since all possible problem causes, e.g., wrong assumptions, wrong formulations, wrong abstractions, imprecise semantics, typos, etc., had to be investigated. Then, if the cause of the problem was in existing theories, the corresponding proofs had to be studied and adjusted. If the verification would happen in a collaborative work at the same time, the time spent on the verification of the entire implementation could be probably decreased by factor 2.

9.2.1 Tools

The verification was carried out on several abstraction layers using a combination of an interactive theorem prover Isabelle/HOL supported by the model checking technique [TA08].

The use of the NuSMV model checker had a significant impact on the verification speed. By our estimation, it reduced the overall verification time by about 20%. NuSMV is a symbolic model checker and helped to argue about hardware properties with the help of temporal logic. However, most proofs dealt with inequalities about real numbers, which is not applicable for NuSMV.

The Isabelle/HOL environment would benefit from a more advanced tool for search of existing theories and lemmas in the libraries provided by Isabelle.

9.2.2 Found Bugs

During our work several hardware bugs were detected. The Processor Interface module (see Figure 6.1) has accessed the configuration register in big-endian mode, whereas we are using little-endian. The bug was discovered by Erik Endres during testing the bus controller on FPGA boards (Section 6.6).

The reset signal did not reset a lot of registers and the control automata of serial interfaces.

Moreover, an interesting bug was discovered during verification of Lemma 20. As depicted in Figure 6.8, the input of the send register will be computed as a disjunction of signal $startsnd \wedge s\_idle$ and of a conjunction of signals $\neg s\_idle$ and the output of an and-tree. The problem here was, that the conjunction checking whether the signal s_idle is inactive was missing in the original implementation. Instead, the output of the and-tree was taken directly. Since the and-tree takes the output bitvector of a $bitcon$

counter which is incremented every cycle, the output of the and-tree was a '1' every eight cycles, namely if the value of the $bitcon$ counter reached '111'. In all other cycles, signal $startsnd$ signal was inactive, i.e., '0' was written into the $dnbit$ register.

The $dnbit$ register stores the clock enable value of the send register S. Hence, the S register was clocked every 8 cycles, even when the send unit was idle. By construction of the Send Unit control automaton (Figure 6.7), the signal which was clocked into the S register was the active signal $se2 = 1$. But according to DRM, after every clocking in clock edge $e(c)$, there is a possible spike at the output of the register at the time $(e(c) + tp_{min} : e(c) + tp_{max})$. Thus, our bus controller could produce a spike on the bus every eight cycles, independently on whether is was a sender or a receiver. Interestingly, this bug *was not* discovered during our tests of interfaces on FPGAs.

9.3 Future Work

Undoubtedly, the presented time-triggered system is not applicable in the context of safety-critical applications if it will not be extended with fault-tolerant features.

As part of the future work we see, e.g., an extension of the Clock Synchronization protocol by fault-tolerance as sketched in [ABK08].

Another possibility would be supplying each ECU with a bus guardian, tackling the problem of 'babbling idiot', a fault of the scheduler mechanism, causing the defect ECU to send messages at a wrong time. This could be done as follows. We supply each controller abc_i with a corresponding bus guardian bg_i. The bus guarding bg_i should have its own clock source and the same scheduler implementation as abc_i, such that it has the same local time notion, as the bus interface. Moreover, the bus guarding gets one configuration register $bg_i.sendl$, s.t. $bg_i.sendl = abc_i.CR.sendl$, a special register $bg_i.S'$ and parts of the sending unit, s.t. holds:

$$\begin{aligned} \forall s < ns \quad : \quad & (i = send(s) \Rightarrow \forall x \in [\alpha_i(r,s) + \mathit{off} : \alpha_i(r,s) + \mathit{off} + tc] : bg_i.S' = 0) \\ \wedge \quad & (i \neq send(s) \Rightarrow \forall x \in [\alpha_i(r,s) : \alpha_i(r,s) + T] : bg_i.S' = 1) \end{aligned}$$

Finally, the bus would be computed as:

$$\forall t : bus(t) = \bigwedge_u (Out_{S,u}(t) \vee Out_{S',u}(t))$$

Obviously, such a construction would protect the bus for all times except for the actual broadcasting time $W(s)$ and at most off cycles before or after $W(s)$.

Moreover, due to redundancy in the message protocol, the bus controller is already fault-tolerant against signal jitter. The computing and verifying of its maximal fault assumptions remains as future work.

BIBLIOGRAPHY

[ABK08] Eyad Alkassar, Peter Boehm, and Steffen Knapp. Correctness of a fault-tolerant real-time scheduler algorithm and its hardware implementation. In *MEMOCODE'2008*, pages 175–186. IEEE Computer Society Press, 2008.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19:3–12, 1993.

[Böh07] Peter Böhm. Formal Verification of a Clock Synchronization Method in a Distributed Automotive System. Master's thesis, Saarland University, Saarbrücken, 2007.

[BP06] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

[BP07] Geoffrey Brown and Lee Pike. Temporal refinement using smt and model checking with an application to physical-layer protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'2007)*, pages 171–180. OmniPress, 2007. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/refinement.html`.

[CCG+98] E. Cimatti, Clarke, Giunchiglia, M. Roveri, Alessandro Cimatti, Fausto Giunchiglia, and Marco Roveri. Nusmv: a reimplementation of smv. In *In Proc. STTT'98*, pages 25–31, 1998.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking, 2002.

[Con06] FlexRay Consortium. FlexRay – the communication system for advanced automotive control applications. `http://www.flexray.com/`, 2006.

[EMST10] Erik Endres, Christian Müller, Andrey Shadrin, and Sergey Tverdyshev. Towards the formal verification of a distributed real-time automotive system. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*,

NASA/CP-2010-216215, pages 212–216, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[End09] Erik Endres. FlexRay ähnliche Kommunikation zwischen FPGA-Boards. Master's thesis, Wissenschaftliche Arbeit, Saarland University, Saarbrücken, 2009.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Fast and efficient cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, 1999.

[GEFP10] Michael Gerke, Ruediger Ehlers, Bernd Finkbeiner, and Hans-Joerg Peter. Model checking the flexray physical layer protocol. In Stefan Kowalewski and Marco Roveri, editors, *FMICS*, volume 6371 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2010.

[Gup93] Vineet Gupta. Concurrent kripke structures. In *In Proceedings of the North American Process Algebra Workshop, Cornell CS-TR-93-1369*, 1993.

[Kna08] Steffen Knapp. *The Correctness of a Distributed Real-Time System*. PhD thesis, Saarland University, 2008.

[KP07] S. Knapp and W. J. Paul. Pervasive verification of distributed real time systems. In T. Hoare M. Broy, J. Gronbauer, editor, *Software System Reliability and Security*, NATO Security Through Science Series. Sub-Series: Information and Communication Vol.9. IOS Press, 2007.

[MP00] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[MSB] Christian Müller, Julien Schmaltz, and Peter Böhm. Formal proof of a FlexRay-like bus interface. `http://www-wjp.cs.uni-sb.de/~cm/abc.html`.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau05] Wolfgang Paul. Computer Architecture 2. Lecture notes. `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur2/ws0506/`, 2005.

[Pfe03] Holger Pfeifer. *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture*. PhD thesis, Universität Ulm, Germany, 2003.

[Pik06] Lee Pike. *Formal Verification of Time-Triggered Systems*. PhD thesis, Indiana University, 2006. Available at `http://www.cs.indiana.edu/~lepike/phd.html`.

[Pik07] Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 231–238. IEEE, 2007. Best paper award. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/fmcad.html`.

[PJ05] Lee Pike and Steven D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html`.

[PM11] Wolfgang Paul and Christian Müller. Complete Formal Hardware Verification of Interfaces for a FlexRay-like Bus. In *Computer Aided Verification 2011 (CAV2011)*. To appear, 2011.

[PSHI99] Holger Pfeifer, Detlef Schwier, Friedrich W. Von Henke, and Fakultat Fur Informatik. Formal verification for time-triggered clock synchronization, 1999.

[Rus99] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, sep 1999.

[Rus01a] John Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[Rus01b] John Rushby. Formal verification of transmission window timing for the time-triggered architecture. Technical report, csl, mp, March 2001.

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–105, Oldenburg, Germany, September 2002. Springer-Verlag.

[RvH89] John Rushby and Friedrich von Henke. Formal verification of the interactive convergence clock synchronization algorithm. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, feb 1989. Revised August 1991.

[RvH93] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, jan 1993.

[Sch07] J. Schmaltz. A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In J. Baumgartner and M. Sheeran, editors, *7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 223–230, Austin, TX, USA, November 11–14 2007. IEEE Press Society.

[SRSP04] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The International Conference on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.

[SS92] N. Shankar and Natarajan Shankar. Mechanical verification of a generalized protocol for byzantine fault tolerant clock synchronization, 1992.

[TA08] Sergey Tverdyshev and Eyad Alkassar. Efficient bit-level model reductions for automated hardware verification. In *TIME 2008*, pages 164–172. IEEE, 2008.

[Tve05] Sergey Tverdyshev. Combination of isabelle/hol with automatic tools. In *Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005. Volume 3717 of Lecture Notes in Computer Science*, pages 302–309. SpringerVerlag, 2005.

[Zha06] Bo Zhang. On the Formal Verification of the FlexRay Communication Protocol. In Stephan Merz and Tobias Nipkow, editors, *Automatic Verification of Critical Systems Automatic Verification of Critical Systems (AVoCS 2006)*, pages 184–189, Nancy/France, 09 2006.

APPENDIX

In this section we show the mapping of all presented lemmas to their formal counterparts in Isabelle. Please note that the proof structure and, hence, the formulation of some lemmas slightly differ in the Isabelle proof.

Name in Thesis	**Isabelle Name / Comment**
Lemma 1	`bounded_drift`
Lemma 2	`Δ_bound`
Lemma 3	*proven inside of parent theorems*
Lemma 4	*not needed in that form formally, serves as an explanation only*
Lemma 5	`alpha_t_ecu_0`
Lemma 6	`alpha_t_ecu_u`
Lemma 7	*proven inside of parent theorems*
Lemma 8	`TransCorr`
Lemma 9	`realtime2cycle'_helper`
Lemma 10	`βsync_to_startedrcv_frontend`
Lemma 11	`schedule_round_exec_master, alpha_to_alpha, slot_boundary`
Lemma 12	*proven inside of parent theorems*
Lemma 13	`slot_start_of_u_plus_off_gt_slot_start_of_w, alpha_w_transmission_lt_alpha_u_end_of_slot`
Lemma 14	`constant_ns_CR, constant_T_CR, constant_L_CR, constant_SENDL_CR, constant_OFF_CR, constant_OFF_CR`
Lemma 15	`startup_soundness, s1_to_eqiwait`
Lemma 16	`alpha_and_alpha_t_eq_master`
Lemma 16	`alpha_and_alpha_t_eq_master`
Lemma 17	`get_SB_in_from_ecu_shorthand, get_SB_ces_from_ecu_shorthand`
Lemma 18	*proven inside of parent theorems*
Lemma 19	`no_s4_implies_no_g0`

Lemma 20	`no_s5_implies_idle_analog_value_on_the_bus`
Lemma 21	`after_s2_no_s5_for_bigT_cycles`
Lemma 22	`alpha_and_alpha_t_eq_slave_part1`
Lemma 23	`from_alpha_bigT_no_s5_and_only_g0`
Lemma 24	`TransmissionCorrectness`
Theorem 1	*replaced by Theorem* 2
Theorem 2	`dig_thm`
Theorem 3	`bus_output_correctness`
Theorem 4	`OverallTransmissionCorrectness`