The Correctness of a Distributed Real-Time System

Dissertation

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

Steffen Knapp

sknapp@wjpserver.cs.uni-sb.de

Saarbrücken, Juli 2008

Tag des Kolloquiums:	07.07.2008
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Reinhard Wilhelm
1. Berichterstatter:	Prof. Dr. Wolfgang Paul
2. Berichterstatter:	Prof. Dr. Wolfgang Kunz
akademischer Mitarbeiter:	Dr. Mark Hillebrand

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig 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 einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, Juli 2008

Steffen Knapp

Zitat:

“Man kann niemanden überholen,
wenn man in seine Fußstapfen tritt.”
Francois Truffaut

Danke

An dieser Stelle möchte ich mich bei Allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben:

Zuerst Herrn Professor Dr. Wolfgang J. Paul für die Vergabe des spannenden und herausfordernden Themas und die wissenschaftliche Unterstützung meiner Promotion.

Außerdem danke ich meiner Freundin Kerstin Hahn für die ständige Ermutigung und Ermunterung insbesondere in der heißen Schlussphase: Ohne Egon und Eulilie sähe die Welt ganz anders aus!

Mein Dank gilt auch meinen (ehemaligen und derzeitigen) Arbeitskollegen am Lehrstuhl von Herrn Paul. Stellvertretend seien hier Peter Böhm, Mark Hillebrand und Dirk Leinenbach genannt. Insbesondere möchte ich mich bei Eyad Alkassar bedanken, der es nun schon viele Jahre mit mir in einem Zimmer ausgehalten hat :-)

Ich danke meinen Freunden Daniel Schmitt, Verena Kremer, Benedikt Grundmann und Ulrich Seyfarth für die Unterstützung sowie die nötige Ablenkung: “We rocked the Garage!”

Und – last but not least – danke ich meiner Familie für die volle Unterstützung in allen Lebenslagen.

Die vorliegende Arbeit wurde teilweise von der International Max Planck Research School for Computer Science (IMPRS) sowie im Rahmen des Projektes Verisoft vom Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01 IS C38 gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei mir.

Abstract

In this thesis we review and extend the pervasive correctness proof for an asynchronous distributed real-time system published in [KP07a]. We take a two-step approach: first, we argue about a single electronic control unit (ECU) consisting of a processor (running the OSEKtime-like operating system OLOS) and a FlexRay-like interface called automotive bus controller (ABC). We extend [KP07a] among others by a local OLOS model [Kna08] and go into details regarding the handling of interrupts and the treatment of devices.

Second, we connect several ECUs via the ABCs and reason about the complete distributed system, see also [KP07b]. Note that the formalization of the scheduling correctness is reported in [ABK08b]. Through several abstraction layers we prove the correctness of the distributed system with respect to a new lock-step model COA that completely abstracts from the ABCs. By establishing the DISTR model [Kna08] it becomes possible to literally reuse the arguments from the first part of this thesis and therefore to simplify the analysis of the complete distributed system. To illustrate the applicability of DISTR, we have formally proven the top-level correctness theorem in the theorem prover Isabelle/HOL.

Throughout the thesis we tie together theorems regarding: processor, ABC, compiler, micro kernel, operating system, and the worst case execution time analysis of applications and systems software.

Zusammenfassung

In dieser Arbeit betrachten und erweitern wir den durchgängigen Korrektheitsbeweis für ein asynchrones verteiltes Echtzeitsystem aus [KP07a]. Wir gehen in zwei Schritten vor: Zuerst betrachten wir eine einzelne elektronische Kontrolleinheit (ECU) bestehend aus einem Prozessor (welcher das OSEKtime ähnliche Betriebsystem OLOS ausführt) und einem FlexRay ähnlichem Interface, auch automobiler Bus Controller (ABC) genannt. Wir erweitern [KP07a] unter anderem um ein lokales OLOS Model [Kna08] und detaillieren die Behandlung von Interrupts sowie den Umgang mit Geräten.

Im zweiten Schritt verbinden wir mehrere ECUs durch die ABCs und argumentieren über das gesamte System, siehe auch [KP07b]. Über die Formalisierung der Scheduler Korrektheit wird in [ABK08b] berichtet. Über mehrere Abstraktionsebenen beweisen wir die Korrektheit des verteilten Systems bezüglich eines neuen gleichgetakteten Modells COA in dem vollständig von den ABCs abstrahiert wird. Durch die Einführung des DISTR Models [Kna08] ist es möglich die Argumente aus dem ersten Teil dieser Arbeit in der Analyse des gesamten verteilten Systems wörtlich wieder zu verwenden. Um die Anwendbarkeit von DISTR zu verdeutlichen haben wir formal die oberste Korrektheits-Aussage im Theorembeweiser Isabelle/HOL beweisen.

Im Zuge dieser Arbeit verbinden wir Theoreme bezüglich: Prozessor, ABC, Compiler, Mikrokern, Betriebsystem und der Worst-Case Laufzeit Analyse von Applikationen und System Software.

Extended Abstract

In this thesis we consider a distributed system consisting of several electronic control units (ECUs) based on [KP07a]. An ECU consists of a DLX processor called VAMP [BJK+03, DHP05, BJK+05], and a FlexRay-like interface [Fle08] called automotive bus controller (ABC). System software is a C0 compiler [LP08] and the OSEKtime-like [OSE08b] operating system OLOS [Kna05, IdRK05, Kna08]. The latter is realized using the generic micro-kernel CVM [GHLP05, KP07a, IdRT08]. Applications are compiled C0 programs that are communicating via a FTCom-like [OSE08a] message buffer of the operating system. Drivers for the ABC are part of OLOS.

Several ECUs are connected by a bus via the ABCs. Communication is done in a time-triggered fashion similar to the static segment of the FlexRay protocol [Fle06]. Time is divided into rounds each consisting of a fixed number of slots. In each slot one ECU is allowed to broadcast one message according to a fixed slot-based schedule.

We clearly split the argumentation about this distributed system in two separate issues: the inter-ECU communication and the local computations on the ECUs. We present a verification approach that separates the argumentation for both issues and combines the results later on in an overall correctness statement.

In the first part of this thesis we argue about a single ECU only. We outline the specification of the DLX instruction set [HP96, MP00] including the handling of interrupts. Using the VAMP [BJK+05] as an example, we explain how to verify hardware designs of complex processors with internal and external interrupts, as reported in [Bey05, Dal06, Tve08]. The resulting correctness proofs are based on the scheduling functions introduced in [SH98, KMP00, MP00].

After this we introduce an I/O device theory. We show how to specify an I/O device and explain how to integrate it into the processor [HIP05, AHK+07]. Furthermore, we extend the processor correctness theorem to deal with a device.

An extension of the VAMP processor by memory management units (MMUs) gives the hardware support for multi-processing operating system kernels and for virtual machine simulation [Hil05, Dal06].

Then we survey a correctness proof for a compiler from the high-level C-like programming language C0 to the DLX instruction set, as reported in [Pet07, LP08, Lei08]. We show how to extend the C0 language by permitting portions of inline assembly code. The resulting language is called $\mathrm{C0_A}$.

We describe the generic operating system kernel called CVM [GHLP05, KP07a, IdRT08]. Seen from the programmers perspective, it consists of a so-called abstract kernel and a set of user processes. The user processes are virtual DLX machines. The abstract kernel is a C0 program that can execute special statements called CVM primitives. These primitives allow the transport of data between kernel and user processes. The CVM correctness proof uses the virtual machine simulation, compiler correctness, and the arguments regarding inline assembly code.

Next we specify the OSEKtime-like operating system OLOS [Kna05, IdRK05, Kna08]. The user processes running under OLOS are C0 programs. These programs can communicate via FTCom-like message buffers with user processes running on the same *or* on remote ECUs. We have implemented an initial version of OLOS by instan-

tiating the abstract CVM kernel. The correctness of the implementation is proven on paper with respect to the OLOS specification.

The main extensions done in the first part (compared to [KP07a]) are the local OLOS specification and a more detailed argumentation regarding the treatment of devices and the handling of interrupts throughout the model stack.

In the second part of this thesis we argue about the complete distributed system. Since the ECUs are running with local oscillators of almost but not exactly equal clock frequency, we cannot guarantee that set-up and hold times of registers are respected when data is being transmitted between ECUs. In such situations serial interfaces are used for data transmission. We review a correctness proof for a serial interface from [BBG$^+$05].

Next we introduce our automotive bus controller (ABC) implementation which has been inspired by [Pau05]. It consists among other things of a message buffers, a serial interface, and a local timer. We show how to prove the correctness of the ABC [BBG$^+$05, KP07b]. Note that we have already reported on the formalization of the scheduling correctness in [ABK08b]. Then we connect our ABC implementation to the processor implementation, using techniques from [HIP05, AHK$^+$07].

The desired communication behavior is specified in a parameterized model called DISTR [Kna08]. This model provides the basis for the separation of the arguments for the inter-ECU communication and the local computations.

We show how to prove the correctness of the inter-ECU communication with respect to the DISTR model. Note that this proof combines arguments regarding asynchronous bit-level transmission with arguments regarding the clock synchronization mechanism at the gate level.

By instantiating the DISTR model with each model of the local stack established in Part I we obtain a distributed version of this stack. When proving correctness, we run into a problem regarding the timer interrupts generated by the ABC: timer interrupts occur in fixed time intervals. At the gate level it is trivial to determine in which cycle such an interrupt is generated. But when we proceed up the model stack we have to argue about the corresponding instruction that gets interrupted on the instruction set architecture (ISA) level. This can inherently not be done at the ISA level alone. The execution time of an instruction depends on cache hits and cache misses, but the memory hierarchy is not visible at the ISA level. Only if we are allowed to look inside the hardware at the gate level, the occurrence of timer interrupts is well-defined.

To solve this problem we first prove the correctness of our ABC implementation with respect to the DISTR model being instantiated with a processor implementation. Then we combine classical program correctness proofs at the ISA level, worst case execution time (WCET) analysis at the gate level, and processor correctness proofs into a pervasive correctness proof for our real-time system from the gate level to the ISA level.

Once we have reached the ISA level we can reuse the local simulation theorems introduced in the first part of this thesis to prove the correctness of the corresponding upper layers.

Finally, we define an automata-theoretic lock-step model called communicating OLOS automata (COA). This model completely abstracts from the devices; commu-

nication is done via the local message-buffers of the operating system directly. In the end, we revise our formal correctness proof of DISTR instantiated with our local OLOS model with respect to the COA model, as reported in [Kna08]. Note that this proof has been formalized in Isabelle/HOL [NPW02].

So, using DISTR, we extended [KP07a] by a more detailed analysis of the interrupt sampling, of all distributed models themselves, and of the propagation of the WCET throughout the distributed model stack resulting in the easy to use COA model.

By that we have established a distributed model stack in which all layers are linked by simulation theorems. To do so, we tie together theorems regarding processor correctness [Bey05, Dal06, Tve08], I/O device integration [HIP05, AHK+07], scheduling correctness [KP07b, ABK08b], message transmission correctness [BBG+05, KP07b], compiler correctness [Pet07, Lei08], micro kernel correctness [GHLP05, IdRT08], operating system correctness [Kna05, IdRK05, Kna08], and the worst case execution time analysis [KP07b] of applications and systems software.

Contents

List of Figures

1 Introduction

More and more safety-critical functions in modern automobiles are controlled by applications running on embedded distributed computer systems. The growing complexity of these systems reduces the coverage of simulation and testing. Both techniques, however useful they are for debugging and evaluation of designs, are "hopelessly inadequate" for proving the absence of errors [Dij72]. Formal verification emerges as the only technique to ensure the demanded degree of reliability.

When analyzing the correctness of the applications, it is desirable to argue in a synchronous model of distributed electronic control units (ECUs) broadcasting messages in lock-step. However, such models are implemented at gate level as highly asynchronous systems. To bridge this gap, it does not suffice to verify certain aspects of a system, such as algorithms or protocols in isolation.

Correctness is shown with respect to some given specification model. Even slight incompatibilities in the used models can allow fatal errors to creep in. With the number of system components increasing, such errors proliferate.

Therefore, the verification objective must tie together the correctness of applications, the operating system, and the hardware implementation itself. The only solution is *pervasive* verification, also called systems or end-to-end verification [BHMY89, Moo03]. To achieve pervasiveness and hereby to minimize the chance of an undiscovered error, we develop a model stack that reaches from the hardware gate level up to an automata-theoretic model of communicating applications. The term "pervasiveness" expresses the fact that parts of the system are not picked out and verified in isolation from the rest. Instead, the correctness of the system is shown by simulation theorems between adjacent models in the model stack.

Note that the verification techniques introduced in this thesis are not limited to the automotive sector. Other sectors like avionics or robotics in which time-triggered system are being used, would benefit from the application of these techniques, too.

Overview

We start with a discussion of related work in Section 2 and an introduction to our notation in Sections 3. The thesis splits in two parts:

In Part I, we discuss the correctness of a *single* electronic control unit (ECU) based on [KP07a]. The instruction set architecture (ISA) of the DLX processor is introduced in Section 4. We show how to implement a processor based on boolean gates and how to prove the correctness of the implementation with respect to the ISA in Section 5. In Section 6, we specify an I/O device and integrate it in both the ISA and the processor using techniques from [HIP05, AHK$^+$07]. This requires an adaptation of the processor correctness theorem from Section 5.

By extending the processor with memory management units, as done in [Hil05, Dal06], and reviewing the correctness proof of the virtualization mechanism in Section 7, we lay a basis for the process separation featured by our operating system.

In Section 8, we introduce the semantics for a high-level C-like programming language and sketch the compiler correctness theorem, see also [Pet07, LP08, Lei08]. In

Section 9, we extend the language by an additional statement allowing the execution of inline assembly code.

In Section 10, we show how to specify, implement, and prove the correctness of the CVM, a generic micro-kernel [GHLP05, KP07a, IdRT08]. We extend existing literature by going into details regarding the treatment of devices and the handling of timer interrupts.

In Section 11, the general device model from Section 6 is instantiated with a specification of a single automotive bus controller (ABC). In Section 12, we specify our OSEKtime-like operating [OSE08b] system OLOS from [Kna05] and show how to prove its correctness. Both, the local ABC specification from Section 11 and the local OLOS specification were first reported in [Kna08].

In Part II, we argue about the complete distributed system. In Sections 13 and 14, we introduce our ABC implementation that is based on boolean gates (being inspired by [Pau05]). Then we detail the verification approach of our implementation [BBG$^+$05, KP07b]. In particular, we deal with the formalization of the scheduling correctness which we have already reported in [ABK08b]. We combine arguments regarding asynchronous bit-level transmission with the arguments regarding the clock synchronization at gate level. In Section 15, the ABC implementation is connected to the processor implementation.

In Section 16, we specify the desired inter-ECU communication behavior in DISTR [Kna08], a synchronous parameterized model that uses the specification of a single ABC from Section 11. By instantiating DISTR with a hardware model, as done in Section 17, a distributed hardware (DH) model is obtained. We prove the correctness of the implementation with respect to this DH model.

As the notion of time is indicated to the processor by means of interrupts, the point of view on the time might differ slightly between the processor and the ABC implementation due to the interrupt sampling policy of the processor. Thus, in Section 18, we introduce a distributed ISA (DISA) model representing the processor's point of view.

In the correctness proof of the DISA model we run into a problem regarding timer interrupts generated by the ABC. While the occurrence of interrupts is well-defined at gate level, the instructions being interrupted at the ISA level can inherently not be determined at the ISA level alone. The execution time of an instruction depends on cache hits and cache misses, but the memory hierarchy is not visible at the ISA level. To solve this problem we combine classical program correctness proofs at the ISA level, worst case execution time (WCET) analysis at the gate level, and a processor correctness proof into a pervasive correctness proof for our real-time system from the gate level to the ISA level.

By instantiating the DISTR model from Section 16 with the other local models from Part I we obtain the corresponding distributed model stack: the distributed CVM (DCVM) in Section 19 and the distributed OLOS (DOLOS) in Section 20.

To prove the correctness of all remaining distributed models we can reuse the local correctness theorems from Part I to a large extent. This is due to the definition of the DISTR model. Note that we were able to extend the arguments from [KP07a] by a more detailed analysis of the interrupt sampling, of all distributed models themselves, and of the propagation of the WCET throughout the distributed model stack.

Finally, in Section 21, we review the automata-theoretic model called communicating OLOS automata (COA) that completely abstracts from the devices. We report on our formal correctness proof of the DISTR model being instantiated with our local OLOS model with respect to the COA model that serves as our top-level specification, see also [Kna08]. Note that this proof has been formalized in Isabelle/HOL [NPW02].

We conclude in Section 22 and point out some future work in Section 23. In Section 24, the important abbreviations for all configurations and transition functions that are used throughout the thesis are summarized.

2 Related Work

Pervasive verification or *systems verification*, i.e. the verification of a system over several layers of abstraction, was introduced in the context of the CLI stack project [BHMY89]. However, the application of such verification-techniques to an industrial scenario without setting too harsh restrictions, e.g. on the programming languages, poses a "grand challenge" according to Moore [Moo03].

Single system layers were addressed in several other attempts. Usually the micro kernel layer is being considered. Recent candidates in this category are L4.verified [HEK+07, EKD+07], VFiasco [HTS02, HT05], and Eros [SW00]. All three projects have established semantics for C variants and have verified some properties on source-code level. However, hardware specific parts as well as the communication with I/O devices have been left out in the argumentation so far. All properties that are considered are purely local in the sense that they argue about a single instance of the system only.

Within the Flint project [NYS07], a verification framework for assembly code was developed. Using this framework and a formalization of a subset of the x86 instruction set, the correctness of context-switching code was formally proven.

There have also been severe efforts to argue about communication systems themselves. In particular, the formal verification of clock synchronization in timed systems has a long history [LMS85, Sha92, PSvH99]. But almost all approaches focused on algorithmic correctness, rather than on concrete system or even hardware correctness. As an exception, Bevier and Young [BY91] described the verification of a low-level hardware implementation of the "Oral Message" algorithm. The presented hardware model is quite simplified, as synchronous data transmission is assumed.

A formal proof of a clock-synchronization circuit was reported by Miner [MJ96]. Based on abstract state machines, a correctness proof of a variant of the Welch-Lynch algorithm [WL88] has been carried out in PVS [ORS92]. However, the algorithm was only manually translated to a hardware specification, which was finally refined semi-automatically to a gate-level implementation. No formal link between both is reported and the low-level bit transmission is not covered in the formal reasoning at all. Note that we explicitly show how to integrate such low-level results into the argumentation.

Rushby proposes the separation of the verification of timing-related properties (as clock synchronization) and protocol specifications [Rus99]. A set of requirements is identified which an implementation of a scheduler (e.g. in hardware) has to obey. In short, (i) clock synchronization and (ii) a round offset large enough to compensate the maximum clock drift between synchronization events are assumed. The central result

is a formal and generic PVS simulation proof between the real-time system and its lock-step and synchronous specification. However, the required assumptions have not been discharged for concrete hardware.

Rushby gives an overview of the formal verification of the Time-Triggered Architecture [SHS+97] in [Rus02] and also formally proves the correctness for some key algorithms, e.g. a clock synchronization algorithm based on the Welch-Lynch algorithm [WL88]. Nevertheless, this remains isolated work. Even Rushby himself states that "some of these algorithms pose formidable challenges to current techniques and have been formally verified only in simplified form or under restricted fault assumptions".

A proof of the Biphase-Mark protocol was proposed by Brown and Pike [BP06]. Their models include metastability but verification is only done at specification level, rather than at the concrete hardware. The models were extracted manually.

Pike [Pik07] corrects and extends Rushby's work and instantiates the new framework with SPIDER, a *fly-by-wire* communication bus used by NASA. His model was extracted from the hardware design by hand, too. Neither of these approaches proved the correctness of any gate-level hardware.

Assuming correct clock synchronization, Zhang verified properties of the FlexRay bus guardian [Zha06]. He did not deal with any hardware implementation.

Serial interfaces were subject to formal verification in the work of Berry *et al.* [BKS03]. They specified a universal asynchronous receiver transmitter (UART) model in a synchronous language and proved a set of safety properties regarding FIFO queues. Based on that a hardware description can be generated and run on a FPGA. However, data transmission was not analyzed.

Neither of these approaches covers all aspects of the whole distributed system. Note that due to the differences in the formalisms being used it would be very cumbersome to combine the approaches into a single unified theory.

Part I
Single Electronic Control Unit (ECU)

3 Notation

Given a bit-string $a = a[n-1:0] \in \mathbb{B}^n$ we denote the natural number with binary representation a by $\langle a \rangle_n$, where $\langle . \rangle_n : \mathbb{B}^n \to \mathbb{N}$:

$$\langle a \rangle_n = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

We use the shorthand $\langle a \rangle$ instead of $\langle a \rangle_n$ if the length of the bit-string is clear from the context.

For numbers $x \in \{0, \ldots, 2^n - 1\}$ the binary representation of length n is the bit-string $bin_n(x)$ where $bin_n : \mathbb{N} \to \mathbb{B}^n$ is defined by:

$$bin_n(x) = y \Leftrightarrow \langle y \rangle = x$$

We denote the n-bit binary addition by $+_n$ where $+_n : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$ is defined by:

$$a +_n b = bin_n(\langle a \rangle_n + \langle b \rangle_n \bmod 2^n)$$

Bit string concatenation is denoted by $\circ$. For bit-strings x and natural numbers $n \geq 1$ we denote by x^n the bit-string obtained by concatenating x exactly n times with itself:

$$\begin{aligned} x^1 &= x \\ x^n &= x^{n-1} \circ x \end{aligned}$$

Throughout the thesis we define various configurations and transition functions on these configurations. Given a configuration c and its transition function δ_{C}, we call the configuration after the application of δ_{C} the next configuration and denote it by c':

$$c' = \delta_{\mathrm{C}}(c)$$

Transition functions are defined by their effect on c, i.e. we define how to get c' from c. We stick to the following notation. An update of a function f at position a with the value x is denoted by:

$$f'(a) = x$$

Typically, configurations are records, e.g. $c.b$ denotes the record component b in configuration c. A record update of the component $c.b$ with the value y is denoted by:

$$c'.b = y$$

To shorten notation we only write down configuration components that are indeed changed. All unmentioned components remain unchanged unless explicitly stated otherwise.

Memories m are defined using functions. The content of a memory m at address a is denoted by $m(a)$. For 32-bit addresses a and natural numbers $x \geq 1$ we denote by $m_x(a)$ the concatenation of x memory entries starting at address a in little endian order:

$$\begin{array}{lcl} m_1(a) & = & m(a) \\ m_x(a) & = & m_{x-1}(a +_{32} 1) \circ m(a) \end{array}$$

Updates a whole memory regions are denoted by:

$$m'_x(z) = u$$

Note that x must coincide with the number of memory entries in u.

4 Instruction Set Architecture (ISA)

Every processor can execute a specific set of instructions. The set of all these instructions and their effects on the processor are called the instruction-set architecture (ISA).

In this section we introduce the ISA for the DLX processor (see [MP00, HP96]). The DLX is a 32-bit reduced instruction set computing (RISC) processor with 32 general purpose registers (GPR) and a byte-adressable memory.

4.1 ISA Configuration

Although a 32-bit architecture with a byte-addressable memory can address up to 2^{32} memory bytes, the available build-in memory is often much smaller in reality.

Let the available number of bytes be given by nb. The set of available memory addresses Ma contains all binary addresses smaller than nb starting from address 0:

$$Ma = \{a \mid 0 \leq \langle a \rangle < nb\}, \text{ where } 0 < nb \leq 2^{32}$$

To simplify the argumentation regarding word accesses and paging later on we require that nb is a multiple of $2^{12} = 4K$.

An ISA configuration isa has the following components:

- a program counter $isa.pc \in \mathbb{B}^{32}$,
- a delayed program counter $isa.dpc \in \mathbb{B}^{32}$ used to specify the delayed branch mechanism (for details see [MP00]),
- a general purpose register file $isa.gpr : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$ containing 32 registers each 32 bit wide, and
- a byte addressable memory $isa.m : Ma \rightarrow \mathbb{B}^8$.

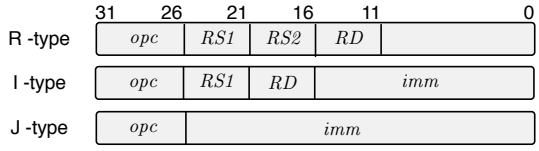

Figure 1: DLX Instruction Types

4.2 Instruction Decoding

The instruction executed in configuration isa is denoted by $I(isa)$. It is the memory word addressed by the delayed PC:

$$I(isa) = isa.m_4(isa.dpc)$$

Instruction decoding is formalized by predicates on the instruction word $I(isa)$. The six high-order bits of the instruction word constitute the opcode opc:

$$opc(isa) = I(isa)[31:26]$$

In some cases it suffices to inspect the opcode to decode an instruction. For instance, the current instruction is a load word lw instruction if the opcode is 100011:

$$lw(isa) \Leftrightarrow opc(isa) = 100011$$

DLX instructions are divided in three instruction types as shown in Figure 1. The type of an instruction defines how the rest of the instruction word is interpreted. An occurrence of R-type instructions is for instance specified by:

$$rtype(isa) \Leftrightarrow opc(isa) = 000000$$

We denote I-type and J-type instructions by $itype(isa)$ and $jtype(isa)$. Their corresponding definitions are stated in [MP00]. Depending on the instruction type certain fields might have different positions within the instruction word. We illustrate this by giving some of the definitions as examples.

The operand $RS1$ denotes the first register source. It is given by:

$$RS1(isa) = I(isa)[25:21]$$

The operand RD denotes the register destination and is given by:

$$RD(isa) = \begin{cases} I(isa)[20:16] & itype(isa) \\ I(isa)[15:11] & \text{otherwise} \end{cases}$$

The immediate constant imm is given by:

$$imm(isa) = \begin{cases} I(isa)[15:0] & itype(isa) \\ I(isa)[25:0] & \text{otherwise} \end{cases}$$

We compute the effective address ea of a load or store operation by adding up the content of the register addressed by the $RS1$ operant $isa.gpr(RS1(isa))$ and the sign extended immediate constant $imm(isa)$.

The addition is performed modulo 2^{32} with two's complement arithmetic. We define the sign extension of the immediate constant by:

$$sxt(imm(isa)) = \begin{cases} imm(isa)[15]^{16} \circ imm(isa) & itype(isa) \\ imm(isa)[26]^{5} \circ imm(isa) & \text{otherwise} \end{cases}$$

This operation turns the immediate constant into a 32-bit constant while preserving the value as a two's complement number (see [MP00] for details). The effective address is thus defined by:

$$ea(isa) = isa.gpr(RS1(isa)) +_{32} sxt(imm(isa))$$

This definition is possible since n bit two's complement numbers and n bit binary numbers have the same value modulo 2^n. For details see Chapter 2 of [MP00].

4.3 ISA Transition Function

Using the above definitions we define the next configuration isa' as the configuration after the execution of $I(isa)$. This formalizes the instruction set in terms of a transition function δ_{isa}:

$$isa' = \delta_{isa}(isa)$$

In the definition of isa' we split cases depending on the instruction to be executed. As an example we specify the next configuration for a load word and a store word instruction.

As the main effect of a load word instruction lw, the general purpose register addressed by the RD field is updated with the memory word addressed by the effective address ea. Furthermore, the PC is incremented by four using 32-bit binary arithmetic and the old PC is copied into the delayed PC:

$$lw(isa) \Rightarrow \begin{cases} isa'.gpr(RD(isa)) & = isa.m_4(ea(isa)) \\ isa'.pc & = isa.pc +_{32} bin_{32}(4) \\ isa'.dpc & = isa.pc \end{cases}$$

This update of the program counters is identical for all instructions except control instructions.

As the main effect of a store word instruction sw, the general purpose register content addressed by RD is copied into the memory word addressed by ea:

$$sw(isa) \Rightarrow \begin{cases} isa'.m_4(ea(isa)) & = isa.gpr(RD(isa)) \\ isa'.pc & = isa.pc +_{32} bin_{32}(4) \\ isa'.dpc & = isa.pc \end{cases}$$

The entire definition of δ_{isa} is obtained by formalizing all instructions in the above formalism. Since this formalization is not important for our work, we do not give it here but refer to a similar definition in [MP00].

4.4 Extended ISA Configuration

Interrupts provide the means to break the execution of a sequential program. As we will see later on in Section 10.3, they can be used by applications to request special services from an operating system.

In order to define the interrupt semantics at the ISA level, we extend the ISA configuration isa by a special purpose register file (SPR):

- the special purpose register file $isa.spr : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$ contains 32 single registers each 32 bit wide.

We use the following abbreviations to refer to particular registers of the special purpose register file (see [MP00]):

- the status register $isa.sr = isa.spr[0]$ stores an interrupt mask,
- the exception cause register $isa.eca = isa.spr[bin_5(2)]$ stores the masked cause of an interrupt, and
- the exception data register $isa.edata = isa.spr[bin_5(5)]$ stores additional information regarding the current interrupt.

4.5 Dealing with Interrupts

Interrupts are triggered by interrupt event signals that might be internally generated (like illegal instruction, misalignment, and overflow) or externally generated (like reset and timer interrupt). Interrupts are numbered using indices $j \in \{0, \ldots, 31\}$. We classify the set of these indices in two categories:

- maskable or not maskable interrupts; the set of indices of maskable interrupts is denoted by M.
- external or internal interrupts; the set of indices of external interrupts is denoted by E.

We denote external event signals by $eev[j]$ with $j \in E$ and we denote internal event signals by $iev[j]$ with $j \notin E$. External event signals are gathered into a vector eev and internal event signals into a vector iev.

Formally these signals must be treated in a very different way. Whether an internal event signal $iev[j]$ is activated in configuration isa is determined by the configuration only.

Let $j = 1$ be used for the illegal instruction interrupt and let $LI \subset \mathbb{B}^{32}$ denote the set of bit patterns not coding an instruction of the ISA. Then the illegal instruction interrupt is given by:

$$iev(isa)[1] \Leftrightarrow I(isa) \notin LI$$

While the internal event signals are functions of the current processor configuration isa, external interrupts are external inputs to the transition function.

Before we can argue about the interrupt handling we need some more auxiliary functions. The cause vector ca of all event signals is a function of the processor configuration isa and the external input eev:

$$ca(isa, eev)[j] = \begin{cases} eev[j] & j \in E \\ iev(isa)[j] & \text{otherwise} \end{cases}$$

The masked cause vector mca is computed from ca using the interrupt mask $isa.sr$. An interrupt j is masked out if it is maskable and $sr[j] = 0$:

$$mca(isa, eev)[j] = \begin{cases} ca(isa, eev)[j] \wedge isa.sr[j] & j \in M \\ ca(isa, eev)[j] & \text{otherwise} \end{cases}$$

The jump to interrupt service routine $JISR$ signal is turned on if any of the masked cause bits equals 1:

$$JISR(isa, eev) = \bigvee_j mca(isa, eev[j])$$

It is important to know the smallest index of an active bit in the mca since several interrupt signals might become active simultaneously. This index is called the interrupt level il. It specifies the interrupt of highest priority that is handled immediately:

$$il(isa, eev) = \min\{j \mid mca(isa, eev)[j] = 1\}$$

4.5.1 Trap Instructions

The DLX instruction set contains a special $trap$ instruction that allows the programmer to generate an internal interrupt. It is of J-type format with opcode 111110. We assign the trap instruction the internal event number 5:

$$iev(isa)[5] \Leftrightarrow opc(isa) = 111110$$

A trap interrupt needs to be handled if this event signal is the active signal with highest priority:

$$trap(isa, eev) \Leftrightarrow il(isa, eev) = 5$$

4.6 ISA Transition Function with Interrupts

We extend the ISA transition function δ_{isa} to deal with interrupts:

$$isa' = \delta_{isa}(isa, eev)$$

We case split on $JISR$. If no interrupt needs to be handled, i.e. $JISR$ does not hold, we use the simple transition function from Section 4.3. The special purpose register file remains unchanged:

$$\neg JISR(isa, eev) \Rightarrow isa' = \delta_{isa}(isa)$$

Otherwise the PCs are forced to point to the start addresses of the interrupt service routine (ISR); we assume it starts at (binary) address 0:

$$JISR(isa, eev) \Rightarrow \begin{cases} isa'.dpc & = bin_{32}(0) \\ isa'.pc & = bin_{32}(4) \end{cases}$$

All maskable interrupts are masked and the masked cause register is saved into the exception cause register:

$$JISR(isa, eev) \Rightarrow \begin{cases} isa'.sr & = bin_{32}(0) \\ isa'.eca & = mca(isa, eev) \end{cases}$$

By $ed(isa)$ we denote the sign-extended immediate constant in case of a trap interrupt and the effective address otherwise:

$$ed(isa, eev) = \begin{cases} sxt(imm(isa)) & trap(isa, eev) \\ ea(isa) & \text{otherwise} \end{cases}$$

Auxiliary data for the interrupt handling (in form of ed) is stored in the exception data register $edata$:

$$isa'.edata = ed(isa, eev)$$

Note that in case of a trap the $edata$ registers keeps the sign-extended immediate constant that was specified by the programmer as part of the instruction word.

This summary suffices for our purposes. A complete definition of the interrupt mechanism in a similar formalism is given in Chapter 5 of [MP00].

5 Processor Hardware (H)

The ISA defined above constitutes the specification for the behavior of a processor. In this section we show how to implement a processor that provably implements this ISA.

5.1 Hardware Configuration

The processor is implemented using a standard digital hardware model. A hardware configuration h consists of:

- 32 bit registers, one for each specification register (comp. Sections 4.1 and 4.4), i.e. $h.pc, h.dpc \in \mathbb{B}^{32}$, $h.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$, and a multi-port register file $h.spr$ as defined in [MP00],
- an additional set of implementation specific registers, and
- an $(a \times d)$-random access memory (RAM), i.e. $h.m : \mathbb{B}^a \to \mathbb{B}^d$.

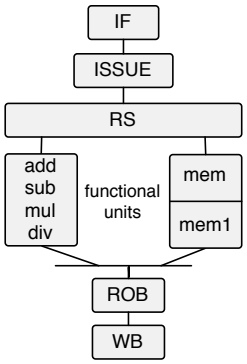

Figure 2: Processor Pipeline

5.2 Hardware Transition Function

Registers and RAMs are connected by Boolean circuits. We denote the value of a signal s in configuration h by $s(h)$. The hardware transition function δ_{h} depends on an external hardware event vector $heev$. It maps a hardware configuration h to the hardware configuration $h' = \delta_{\mathrm{h}}(h, heev)$, i.e. the configuration after the next clock cycle. For registers $h.r$ with clock enable signal rce and input rin we define:

$$h'.r = \begin{cases} rin(h) & rce(h) \\ h.r & \text{otherwise} \end{cases}$$

A multi-port register file which allows the simultaneous update of several register is defined in [MP00].

For a RAM $h.m$ with address signal $addr$, data input din, and the write signal w we define:

$$h'.m(x) = \begin{cases} din(h) & x = addr(h) \wedge w(h) \\ h.m(x) & \text{otherwise} \end{cases}$$

Hardware computations are defined in the usual way as sequences of configurations. Hardware computations must satisfy for all cycles t:

$$h^{t+1} = \delta_{\mathrm{h}}(h^t, heev^t)$$

A superscript t in this model is read as 'during cycle t'.

In general, processor correctness theorems state that hardware defined in a hardware model simulates in some sense the ISA. In the following sections we state more precisely in which sense.

5.3 Scheduling Functions

The processor correctness proof considered here is based on the concept of scheduling functions s as in [KMP00, MP00]. The hardware of pipelined processors, supporting

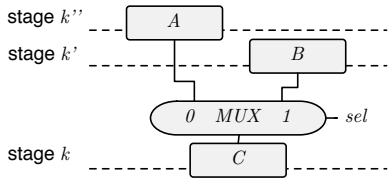

Figure 3: Illustration of Scheduling Functions

out-of-order execution [Kro01], consists of many stages as sketched in Figure 2, e.g. instruction fetch (IF) stage, issue stage, reservation stations (RS), reorder buffer (ROB), write back (WB) stage.

Stages can be full or empty due to pipeline bubbles. The hardware keeps track of this with the help of full bits $full_k$ for each stage k as defined in [MP00]. The value of the full bit of stage k in cycle t is denoted by $full_k(h^t)$. To shorten notation we simply write $full_k^t$. Note that the instruction fetch stage IF is always full, since new instructions can be fetched at any time, i.e. $\forall t.\ full_{IF}^t = 1$.

For stages k that are full during hardware cycle t, i.e. $full_k^t$ holds, the value of the scheduling function $s(k,t)$ is the index i of the instruction that is in stage k during cycle t. If the stage is not full, $s(k,t)$ is the index of the instruction that was in stage k in the last cycle before t when the stage was full. Initially $s(IF,0) = 0$ holds.

In the definition of scheduling functions we use a simple idea: We imagine the hardware having registers that can hold integers of arbitrary size. We augment each stage with such a register and use it to store the index of the instruction currently being executed in that stage.

Note that these indices are computed exactly as the tags in a Tomasulo scheduler. The only difference is that the indices have unbounded size because we count up to arbitrarily large indices. In real hardware this is not possible and not necessary. Nevertheless, in this abstract mathematical model there is no problem to do this.

Each stage k of the processors under consideration has an update enable signal ue_k. Stage k gets new data in cycle t if the update enable signal ue_k equals one in cycle $t-1$. We fetch instructions in order and define for the instruction fetch stage IF:

$$s(IF,t) = \begin{cases} s(IF,t-1)+1 & ue_{IF}^{t-1} \\ s(IF,t-1) & \text{otherwise} \end{cases}$$

Other stages k can get data belonging to a new instruction from one or more stages. Examples for stages with more than one predecessor are: cycles in the data path of a floating point unit performing iterative division, or the producer registers connected to the common data bus of a Tomasulo scheduler. In this situation we define for each stage k a predicate $trans(k',k,t)$ indicating that in cycle t data is transmitted from stage k' to stage k. In the example in Figure 3 we use the select signal sel of the multiplexer MUX and define:

$$trans(k',k,t) = ue_k^t \wedge sel^t$$

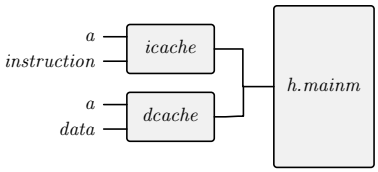

Figure 4: Memory System

If $trans(k', k, t-1)$ holds for some k' we set $s(k,t) = s(k', t-1)$ for that k'. Otherwise $s(k,t) = s(k, t-1)$.

5.4 Processor Correctness Theorem

The simulation relation $isasim(h, isa)$ between some hardware configuration h and some ISA configuration isa requires that specification registers $r \in \{pc, dpc, gpr, spr\}$ have identical values:

$$h.r = isa.r$$

We would like to define something similar for the memory. However, this cannot work, since the user visible processor memory is simulated in the hardware by a memory system consisting among others of an instruction cache $icache$, a data cache $dcache$, and a main memory $mainm$ as shown in Figure 4.

There exists a non-trivial function $m(h) : Ma \to \mathbb{B}^8$ specifying the memory simulated by the memory system. This function can be defined in the following way [Pau08]: Imagine we have an abstract cache ac. This abstract cache stores content of the size of the main memory $ac.con : Ma \to \mathbb{B}^8$ and a valid bit $ac.v : Ma \to \mathbb{B}$.

We use two abstract caches to specify the memory that is simulated by the memory system in the hardware configuration h, one for data $dac(h)$ and one for instructions $iac(h)$. Let $ihit(h,a)$ denote a hit in the instruction cache for an address a and let $dhit(h,b)$ denote a hit in the data cache for an address b. Then the valid bits are defined as follows:

$$\begin{aligned} dac(h).v(a) &= dhit(h,a) \\ iac(h).v(b) &= ihit(h,b) \end{aligned}$$

The content fields simply map addresses a to the content of the corresponding hardware caches at address a.

Note that in the processor under consideration the caches snoop on each other. Hence the data of address a is only in at most one cache [BJK+03, Bey05]. Thus we define:

$$m(h)(a) = \begin{cases} iac(h).con(a) & iac(h).v(a) \\ dac(h).con(a) & dac(h).v(a) \\ h.mainm(a) & \text{otherwise} \end{cases}$$

Using this definition we additionally require in the simulation relation $isasim(h, isa)$:

$$m(h)(a) = isa.m(a)$$

In a pipelined machine this simulation relation almost never holds. In one cycle different hardware stages k usually hold data from different ISA configurations; after all this is the very idea of pipelining. There is however an important exception when the pipeline is drained and thus all hardware stages except the instruction fetch stage are empty:

$$drained(h^t) \Leftrightarrow \forall k \neq IF.\ \neg full_k^t$$

This happens to be the case after interrupts, in particular initially after reset.

First, we ignore the interrupt event signals (which brings us formally back to an ISA computations defined by $isa^{i+1} = \delta_{isa}(isa^i)$).

Each user visible register $isa.r \in \{pc, dpc, gpr, spr\}$ of the processor has a counter Part $h.r$ belonging to some stage $k = stage(r)$ of the hardware. We have to show by induction on t that for all registers r in stage $k = stage(r)$, the value of the hardware register r in cycle t equals the value of the ISA register r for the instruction scheduled in stage k in cycle t:

$$h^t.r = isa^{s(k,t)}.r$$

For the memory we have to consider the memory unit of the processor consisting of two stages mem and $mem1$. Stage mem contains hardware for the computation of the effective address. The memory $m(h^t)$ simulated by the memory hierarchy of the hardware in cycle t is identical with the ISA memory $isa^{s(mem1,t)}.m$ for the instruction scheduled in stage $mem1$ in cycle t:

$$m(h^t)(a) = isa^{s(mem1,t)}.m(a)$$

We combine both statements in a processor correctness theorem.

Theorem 1 (Processor Correctness) *Let the pipeline be drained and let the simulation relation hold initially, i.e.* $drained(h^0)$ *and* $isasim(h^0, isa^0)$*. Then for all cycles* t *and for all registers* $r \in \{pc, dpc, gpr, spr\}$ *in stage* $k = stage(r)$*:*

$$\begin{aligned} h^t.r &= isa^{s(k,t)}.r \\ m(h^t)(a) &= isa^{s(mem1,t)}.m(a) \end{aligned}$$

The theorem is proven by induction on the hardware cycle t using additional statements on the implementation registers. Furthermore, restrictions for the software to be executed by the processor are needed, as detailed below.

For complex processors this requires hundreds of pages of paper and pencil proofs (see [MP00]). A formal correctness proof is described in [Bey05, BJK+05].

5.4.1 Synchronization Conditions

If the hardware implementation of a physical machine is pipelined or if instructions are executed out of order the following situation might occur: An instruction $I(isa^i)$ that is in the memory stage may modify a later instruction $I(isa^j)$ for $j > i$ after it has been fetched. This situation is called a read after write (RAW) hazard. Instruction $I(isa^i)$ may overwrite the instruction itself or change the operating mode.

On a RAW hazard an instruction fetch (in particular a translated fetch implemented by a memory management unit) would not work correctly. It is possible to detect such data dependencies in hardware and to roll back the computation if necessary.

Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(isa^j)$ denote the address of instruction $I(isa^j)$. If $I(isa^i)$ writes to address $iaddr(isa^j)$, then an intermediate instruction $I(isa^k)$ for $i < k < j$ must drain the pipe.

5.5 Dealing with Interrupts

When dealing with interrupts the hardware gets external inputs called the hardware external event signals $heev$. Their value in hardware cycle t is denoted by $heev^t$. Based on these we construct a sequence of external ISA event signals eev^i such that the ISA computation satisfying $isa^{i+1} = \delta_{isa}(isa^i, eev^i)$ simulates a hardware computation $h^{t+1} = \delta_{\mathrm{h}}(h^t, heev^t)$.

For non-memory operations the processor hardware samples external event signals in the write back stage WB as detailed in [Tve08]. Since WB is the last stage in the pipeline it cannot be stalled. Thus for every instruction i there is exactly one hardware cycle t when the instruction is written back:

$$t = WB(i) \Leftrightarrow s(WB, t) = i \wedge full^t_{WB}$$

In Part II of this thesis we will argue that in the distributed system under consideration the external interrupts occur only if the processor executes non-memory operations. Thus the external ISA event signal observed by instruction i is:

$$eev^i = heev^{WB(i)}$$

Note that a hardware event signal $heev^t$ is not visible to the ISA computation if the write back stage in cycle t is empty. With this new definition of the ISA-/hardware computation Theorem 1 still holds. More details regarding a formal processor correctness proof dealing with external event signals are given in [Bey05, Dal06].

Later on we argue about hardware cycles in which the pipeline is drained. To do so, we use the signal $JISR(h, heev)$ that indicates if an interrupt needs to be handled in the given hardware configuration. The predicate is defined by applying the definitions from Section 4.6 to hardware configurations. Note that once the $JISR$ predicate holds, the pipeline is $drained$ in the next hardware cycle:

$$JISR(h^t, heev^t) \Rightarrow drained(h^{t+1})$$

6 Devices (D)

In this section we introduce a framework for the integration of a device into the model stack. This framework will later on be instantiated with an automotive bus controller (see Sections 11 and 13).

The communication between the processor and the device is done via memory mapped I/O. The device is assigned a set of device addresses. These addresses, called

I/O ports, are mapped into the processors memory above the maximal physical byte address nb. Via read and write operations to these I/O ports the processor can communicate with the device.

In the remainder of this section we show how to integrate a device into the processor implementation and into the ISA.

6.1 Device Configuration

In general, a device has more state than is visible through the I/O ports. A device configuration d has the following components:

- a word addressable port RAM $d.m$ where the number of the addresses is given by the page size K, as defined in Section 4:

$$d.m : \mathbb{B}^{10} \rightarrow \mathbb{B}^{32}$$

- a boolean interrupt flag $d.int$ indicating if an interrupt was generated but has not been cleared, yet.
- some device specific 'internal' state $d.z$.

We assign the external event 13 to the device. As we are dealing with a single device only, the external event vector eev is computed as follows:

$$eev(d) = 0^{13} \circ d.int \circ 0^{18}$$

6.2 Integration of a Device in General

The I/O ports of a device can be divided into three categories:

- control ports are only written from the processor,
- status ports are only read by the processor, and
- data ports can be written or read from both the processor and the outside world.

Inputs from and outputs to the outside world are device specific: network devices have inputs and outputs, monitors produce only outputs, keyboards take only inputs, disks neither produce outputs nor consume external inputs. Thus, in general, the classical synchronization issues of shared memory arise.

In the following section we show how to integrate a device into the processor implementation. Therefore we extend the transition function by an external input from the outside world. We use a device specific $stable$ predicate which indicates if all the I/O ports are not altered by the outside world and thus have memory semantics. In Section 15 we will instantiate the device with our concrete automotive bus controller (ABC) implementation.

In Section 6.5, we define the semantics of a purely stable device at the gate level. In the remaining sections of Part I we will deal with a stable device only. The resulting

theorems will be applied later on in Part II for a bounded interval in which the real device is indeed stable.

We justify the usage of the stable device semantics in Section 17. There we will prove a simulation theorem between our possibly unstable device implementation and an extended semantics which is based on the stable device semantics from 6.5. Note that our device implementation behaves most of the time like a stable device. Only in well-defined cycles the device alters the data-ports. In the interval between those cycles the device is stable and we can use the semantics developed throughout Part I in our argumentation.

6.3 Integration of a Device into the Processor Implementation

We integrate a device into the processor implementation by combining the processor configuration with the device configuration. We denote the combined configuration by $impl$. It has the following components:

- a processor configuration $impl.p$ being a hardware configuration h as defined in Section 5, and
- a device configuration $impl.d$ as defined above.

The device takes inputs from and produces output to the processor side and the outside world. Inputs from the processor side are like inputs for the RAM and consist of: data input $din(impl.p)$, address $adr(impl.p)$, a write signal $dw(impl.p)$ and a read signal $dr(impl.p)$. Outputs to the processor side consists of data output $dout(impl.d)$ and an external interrupt signal. Inputs to and outputs from the processor side are device specific.

Let address $adr(impl.p)$ be the address of a memory operation in analogy to the effective address ea in the ISA.

The device is mapped into the processors memory starting at some base address ba which we assume to be $1^{20}0^{12}$. Using an address decoder the memory system decides if a read or write access to the memory is a device access. The read signal $dr(impl.p)$, which indicates a read-access to the device, is activated if the operation is a load word operation $lw(impl.p)$ and the memory address is greater or equal to the base address ba. The latter is implemented by an equality check of the upper most bits of the address:

$$dr(impl.p) \Leftrightarrow (lw(impl.p) \wedge adr(impl.p)[31:12] = 1^{20})$$

The write signal $dw(impl.p)$, which indicates a write-access to the device, is activated if the operation is a store word operation $sw(impl.p)$ and the memory address is greater or equal to the base address ba.

$$dw(impl.p) \Leftrightarrow (sw(impl.p) \wedge adr(impl.p)[31:12] = 1^{20})$$

Using these two signals we define a predicate $daccess(impl)$ which indicates if the processor is accessing the device:

$$daccess(impl) = (dw(impl.p) \vee dr(impl.p))$$

In case of a device access, the lower bits of the memory address code the port, i.e. the word-address within the address range of the device:

$$port(impl) = adr(impl.p)[11:2]$$

Note that the cache system must be designed in a way that it does not cache accesses to I/O ports.

6.4 Implementation Semantics

Next we define the transition function δ_{impl} for the combined configuration $impl$. In addition to the configuration itself it takes a device specific input x from the outside world:

$$impl' = \delta_{impl}(impl, x)$$

Being at the gate level, the processor implementation and the device are clocked in lock-step.

If the processor does not access the device, i.e. $\neg daccess(impl)$, the processor configuration is updated according to the old transition function δ_{h} from Section 5.2:

$$\neg daccess(impl) \Rightarrow impl'.p = \delta_{\mathrm{h}}(impl.p)$$

To define the transition function δ_{impl} in case of a device access, we use the predicate $stable(impl)$ which indicates if:

- the I/O ports of the ABC implementation have memory semantics. Thus for a read operation from the device it holds that:

  $$daccess(impl) \wedge dr(impl.p) \Rightarrow dout(impl.d) = impl.d.m(port(impl))$$

  For a write operation to the device it holds that:

  $$daccess(impl) \wedge dw(impl.p) \Rightarrow$$
  $$impl'.d.m(x) = \begin{cases} din(impl.p) & x = port(impl) \\ impl.d.m(x) & \text{otherwise} \end{cases}$$

- a write to a special device address, called *command port*, has the side-effect that the interrupt flag of the device is cleared. We assume w.l.o.g. that this port has the device address cmd:

  $$daccess(impl) \wedge dw(impl.p) \wedge (port(impl) = cmd) \Rightarrow impl'.d.int = 0$$

In case of a device access, i.e. $daccess(impl)$, the transition function δ_{impl} is defined as follows: if the device is being accessed by a read operation and the device is stable, the output $dout(impl.d)$ from the device to the processor is the content of the I/O port being addressed:

$$daccess(impl) \wedge dr(impl.p) \wedge stable(impl) \Rightarrow$$
$$dout(impl.d) = impl.d.m(port(impl))$$

The processor component is updated using the output $dout(impl.d)$ from the device.

If the device is being accessed by a write operation and the device is stable, the ports behave like a RAM:

$$daccess(impl) \wedge dw(impl.p) \wedge stable(impl) \Rightarrow$$
$$impl'.d.m(x) = \begin{cases} din(impl.p) & x = port(impl) \\ impl.d.m(x) & \text{otherwise} \end{cases}$$

Furthermore, if the device is stable, a write to the command port cmd has the side-effect that the interrupt flag is cleared:

$$daccess(impl) \wedge dw(impl.p) \wedge stable(impl) \wedge (port(impl) = cmd) \Rightarrow$$
$$impl'.d.int = 0$$

If the device is not stable, the ports might be read or modified by the device side in a device specific way using the external input x. The effect of writing to a non-stable device is left undefined. The processor side learns about changes in the stable predicate by an interrupt generated by the device.

Note that according to the above definitions the device is unusually fast: it updates a port in a single cycle of the processor hardware. Devices are usually slower and thus require a busy signal indicating if a read or write access is in progress. The above definitions could be extended in this way using a memory protocol like the one from [MP00].

6.5 Implementation Semantics for a Stable Device

In this section we show how to integrate a stable device into the processor hardware. This device simply behaves like an additional memory.

We denote the combined configuration by hd. It has the following components:

- a hardware configuration $hd.p$ as defined in Section 5, and
- a device configuration $hd.d$.

Note that the configuration hd has basically the same components as the configuration $impl$. However, the transition function for hd does explicitly model the behavior of a stable device. In Section 17 we show that in bounded cycle intervals the hd configuration and transition function can be used to argue about the implementation configuration $impl$.

Next we define the transition function δ_{hd} for the combined configuration hd. Note that this transition function does not take any external input:

$$hd' = \delta_{hd}(hd)$$

If the processor does not access the device, i.e. $\neg daccess(hd)$, we reuse the old transition function δ_{h} from Section 5.2. The device remains unchanged:

$$\neg daccess(hd) \Rightarrow \begin{cases} hd'.p = \delta_{\mathrm{h}}(hd.p) \\ hd'.d = hd.d \end{cases}$$

If the device is being accessed by a read operation, the output $dout(hd.d)$ from the device to the processor is the content of the I/O port being addressed:

$$daccess(hd) \wedge dr(hd.p) \Rightarrow dout(hd.d) = hd.d.m(port(hd))$$

The processor component is updated using the data output $dout(hd.d)$ from the device.

If the device is being accessed by a write operation, the ports behave like a RAM:

$$daccess(hd) \wedge dw(hd.p) \Rightarrow hd'.d.m(x) = \begin{cases} din(hd.p) & x = port(hd) \\ hd.d.m(x) & \text{otherwise} \end{cases}$$

A write to the command port, has the side-effect that the interrupt flag of the device is cleared:

$$daccess(hd) \wedge dw(hd.p) \wedge (port(hd) = cmd) \Rightarrow hd'.d.int = 0$$

Note that if the stable predicate $stable$ holds, the δ_{hd} transition function can be used instead of the δ_{impl} transition function:

Lemma 1 (Stable Semantics) *If the stable predicate $stable(impl)$ holds, the δ_{hd} semantics corresponds to the δ_{impl} semantics:*

$$\forall x.\ stable(impl) \Rightarrow \begin{cases} (\delta_{impl}(impl,x)).p & = & (\delta_{hd}(impl)).p \\ (\delta_{impl}(impl,x)).d.m & = & (\delta_{hd}(impl)).d.m \\ (\delta_{impl}(impl,x)).d.int & = & (\delta_{hd}(impl)).d.int \end{cases}$$

This lemma is proven using the definition of the $stable$ predicate from Section 6.4.

6.6 Integration of a Stable Device into the ISA

We integrate a stable device into the ISA by combining the two configurations. The combined configuration $isad$ has the following components:

- an ISA configuration $isad.p$ as defined in Section 4.4, and
- a device configuration $isad.d$ as defined above.

The predicate $daccess(isad)$ indicates if the processor is accessing the device. It is true if the processor performs a store word or a load word operation and the effective address equals or is greater than the base address ba which is in our case $1^{20}0^{12}$:

$$daccess(isad) = (lw(isad.p) \vee sw(isad.p)) \wedge (ea(isad.p)[31:12] = 1^{20})$$

If the predicate $daccess(isad)$ holds, the lower bits of the effective address code the port address $port(isad)$, i.e. the word-address within the address range of the device:

$$port(isad) = ea(isad.p)[11:2]$$

Using these two functions, the transition function δ_{isad} for the combined configuration $isad$ is defined as follows:

$$isad' = \delta_{isad}(isad)$$

This transition function will only be applied for intervals in which the device is stable and thus behaves like an ordinary RAM.

If the device is not accessed or an interrupt needs to be handled, the old transition function from Section 4.6 is reused. The state of the device is not altered:

$$\neg daccess(isad) \vee JISR(isad.p, eev(isad.d)) \Rightarrow \begin{cases} isad'.p & = & \delta_{isa}(isad.p, eev(isad.d)) \\ isad'.d & = & isad.d \end{cases}$$

If the processor is accessing the device via a memory operation and no interrupt needs to be handled, an extended load word or store word semantics is used. In case of a load word from a device address the return destination register RD is updated with the content of the accessed device memory:

$$daccess(isad) \wedge \neg JISR(isad.p, eev(isad.d)) \wedge lw(isad.p) \Rightarrow isad'.p.gpr(RD(isad.p)) = isad.d.m(port(isad))$$

In case of a store word to a device address, the accessed ports behave like a RAM:

$$daccess(isad) \wedge \neg JISR(isad.p, eev(isad.d)) \wedge sw(isad.p) \Rightarrow isad'.d.m(x) = \begin{cases} isad.p.gpr(RD(isad.p)) & x = port(isad) \\ isad.d.m(x) & \text{otherwise} \end{cases}$$

A write to the command port cmd has the side-effect that the interrupt flag is cleared:

$$daccess(isad) \wedge \neg JISR(isad.p, eev(isad.d)) \wedge sw(isad.p) \wedge (port(isad) = cmd) \Rightarrow isad'.d.int = 0$$

6.7 Processor Correctness Theorem with Devices

Next, we extend the processor correctness (Theorem 1) to deal with a stable device. Although we will quantify in the following theorem over all cycles t it will only be applied for a bounded computation in which the device indeed behaves like an ordinary RAM (see Section 17).

In the hardware the device is placed parallel to the normal memory system in stage $mem1$, thus we can use the same scheduling functions as for the memory.

By extending the simulation relation from Section 5.4 we get a new simulation relation $isadsim(hd, isad)$. We additionally require that the device configuration be the same:

$$isasim(hd.p, isad.p)$$
$$hd.d = isad.d$$

The processor correctness theorem dealing with devices is stated as follows:

Theorem 2 (Processor Correctness with Devices) *Let the pipeline be drained and let the simulation relation hold initially, i.e.* $drained(hd.h^0)$ *and* $isadsim(hd^0, isad^0)$.

Then for all hardware cycles t*, for all processor stages* k *and for all specification registers* $r \in \{pc, dpc, gpr, spr\}$ *with* $stage(r) = k$*:*

$$\begin{aligned} hd^t.p.r &= isad^{s(k,t)}.p.r \\ m(hd^t.p) &= isad^{s(mem1,t)}.p.m \\ hd^t.d &= isad^{s(mem1,t)}.d \end{aligned}$$

The theorem is proven by induction on the hardware cycle t using additional statements on the implementation registers as well as the software conditions from Section 5.4.1. Since we are using the same device model for the ISA level as for the gate level, the external event signals correspond to the translation from Section 5.5:

$$(eev(isad.d))^i = (eev(hd.d))^{WB(i)}$$

In [Tve08] Sergey Tverdyshev will report on the formal results regarding the integration of devices into the processor design. Note that details regarding the coupling of the processor with a harddisk as well as with a universal asynchronous receiver transmitter (UART) have already been reported in [HIP05, AHK+07].

7 Memory Management (MM)

Physical machines consist of a processor operating on fast physical memory and on slow swap memory. In reality computer systems are shipped with less physical memory than could be addressed by the architecture. This is due to the fact that physical memory is expensive. Instead, the much cheaper (and slower) swap memory, e.g. in form of a hard-disk, is used to provide the application programmer the illusion of a huge physical memory.

By using the existing physical memory as a write-back cache for the swap memory this illusion is kept alive (see [GHLP05, Hil05]). Depending on the memory usage of the applications, huge amount of data needs to be moved between swap and main memory. This *swapping* is done transparent for the applications, however, due to the slowness of the swap memory compared to the main memory, huge latencies occur.

These latencies make swapping not feasible for real-time operating systems. In most cases these applications implement quite simple tasks like polling a crash sensor or controlling a GPS navigation system. In practice the applications are optimized in two ways: Low memory consumption and run-time.

We require that all applications to be run under the real-time operating system fit in the physical memory. No swap memory is used. Nevertheless, we use the concepts that are applied in physical machines to ensure process separation.

In the following sections we introduce these concepts and sketch the corresponding correctness theorem.

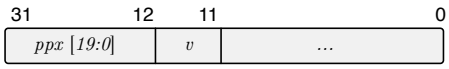

Figure 5: Page Table Entry

7.1 Physical Machine Configuration

We extend the special purpose register file from the ISA configuration isa from Section 4.1 by some additional registers:

- the page table origin pto containing the base address of the page table that is used for address translation (see below); we use the abbreviation $isa.pto$.
- the page table length ptl indicating the length of the pagetable; we use the abbreviation $isa.ptl$.
- the mode register $mode$ indicating the current operating mode; we use the abbreviation $isa.mode$.

7.2 Address Translation, Physical and Virtual Machines

The behavior of a physical machine is determined by the mode register: In system mode, i.e. if $isa.mode = 0^{32}$, the physical machine operates like the basic processor model from Section 4. In user mode, i.e. if $isa.mode \neq 0^{32}$, the physical machine with appropriate software (see Section 5.4.1) simulates the basic processor model using page tables for address translation.

The simulated machine is called a *virtual machine*. The addresses used by the virtual machines are called virtual addresses. We keep the notation isa for configurations of the physical machine and we denote configurations of the virtual machine by vm. Virtual addresses $vadr$ are split into a page index $vadr.px = vadr[31:12]$ and a byte index $vadr.bx = vadr[11:0]$. Thus the page size is $2^{12} = 4K$ bytes (compare Section 4.1).

In user mode an access to memory address $vadr$ is subject to an address translation. It either causes a page fault or it is redirected to the translated physical memory address $pma(isa, vadr)$ as defined below.

The result of this address translation depends on the content of the *page table*, a region of the physical memory starting at address $isa.pto \cdot 4K$ with $isa.ptl$ entries. Page table entries have a length of four bytes. The page table entry address for virtual address $vadr$ is defined by:

$$ptea(isa, vadr) = isa.pto \cdot 4K + 4 \cdot vadr.px$$

and the page table entry of $vadr$ is defined by

$$pte(isa, vadr) = isa.m_4(ptea(isa, vadr))$$

Here we use only two parts of a page table entry as depicted in Figure 5: The physical page index $ppx(isa, vadr) = pte(isa, vadr)[31:12]$ and the valid bit $v(isa, vadr) = pte(isa, vadr)[11]$.

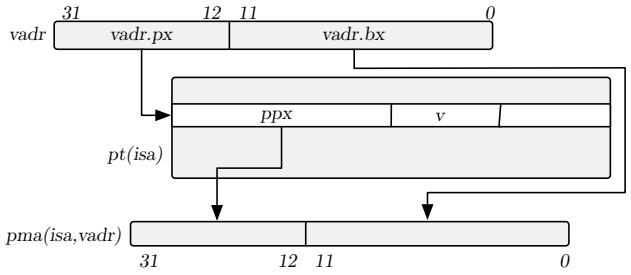

Figure 6: Address Translation

Being in user mode and accessing memory address $vadr$, a page fault signals if the page index exceeds the page table length, $vadr.px \geq isa.ptl$ or if the page table entry is not valid, $v(isa, vadr) = 0$.

If no page fault is generated, the access is performed on the (translated) physical memory address $pma(isa, vadr)$. It is defined as the concatenation of the physical page index and the physical byte index (see Figure 6):

$$pma(isa, vadr) = ppx(isa, vadr) \circ vadr.bx$$

A real-time operating system must make sure that no page-fault is generated resulting in additional constraints for the applications.

7.3 Physical Machine Transition Function

Hence, the transition function of the physical machine corresponds, with slight extensions, to the simple ISA transition function from Section 4.3. The instruction $I(isa)$ fetched in configuration isa is for instance defined as follows: if $isa.mode = 0$ then $I(isa) = isa.m_4(isa.dpc)$, otherwise, provided that no page-fault is generated, $I(isa) = isa.m_4(pma(isa, isa.dpc))$. The effected address ea of a load word lw and a store word sw operation is translated accordingly.

7.4 Virtual Memory Correctness Theorem

Let nb be the maximal physical byte address. The simulation relation $B(vm, isa)$ indicates if a (user mode) physical machine configuration isa encodes the virtual machine configuration vm. Essentially, $B(vm, isa)$ is the conjunction of the following two conditions:

- for each of the $nb/4K$ pages of virtual memory (nb is assumed to be a multiple of $4K$ see Section 4.1) there is a page table entry in the physical machine, i.e. $nb/(4K) = isa.ptl + 1$.
- the content of the virtual memory addressed by $vadr$ is stored in the physical memory at address $pma(isa, vadr)$:

$$vm.m(vadr) = isa.m(pma(isa, vadr))$$

The simulation theorem for a single virtual machine is stated as follows:

Theorem 3 (Virtual Machine Correctness) *Let the simulation relation hold initially, i.e.* $B(vm^0, isa^0)$.

For all computations of the virtual machine there is a computation of the physical machine and there are step numbers $s(i)$ *for the physical machine such that for all steps* i*:*

$$B(vm^i, isa^{s(i)})$$

The theorem is proven by induction on the steps i. Note that additional software constraints (compare Section 5.4.1) are required to prove the above theorem.

If isa^j is in user mode and $I(isa^i)$ for $j > i$ writes to $ptea(isa^j, isa^j.dpc)$, an intermediate instruction $I(isa^k)$ for $i < k < j$ must drain the pipe. The mode can only be changed to user mode by an rfe (return from exception) instruction (and the hardware guarantees that rfe instructions drain the pipeline).

8 The Programming Language C0

In this section we introduce the high-level programming language C0 by summarizing the results from [LP08]. C0 is roughly speaking PASCAL with C syntax.

8.1 C0 Types Expressions and Statements

In C0 types are either elementary ($bool$, int, ...), or pointer types, or aggregates ($array$ or $struct$). A type is called simple if it is an elementary type or a pointer type.

We define the (abstract) size of types for simple types t by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \ldots, n_s : t_s\}) = \sum_i size(t_i)$.

Values of variables with simple types are called *simple values*. Variables of aggregate type are called *aggregate values*, which are represented as a flat sequence of simple values.

Variable names and literals are expressions. Given expressions e and e' and a name n, then array accesses $e[e']$, struct accesses $e.n$, pointer dereferencing $^*e$, and the 'address-of' operation $\&e$ are also expressions. Additionally, C0 supports the usual unary and binary operators.

Pointer arithmetic is forbidden. In C0 expressions do not have side effects. Function calls as part of expressions are not allowed.

Given a type t and expressions e, e' and e_i, then the empty statement $skip$, assignments $e = e'$, memory allocation $e = new(t)$, function calls $e = f(e_1, \ldots, e_m)$ and function returns $return\ e$ are statements.

Given an expression e and statements s and s', then while loops $while\ e\ do\ s$, if conditionals $if\ e\ do\ s\ else\ s'$, and sequential composition $(s; s')$ are also statements. In each function body the last statement must be the only $return$ statement of this body.

8.2 C0 Machine Configuration

The state of a C0 program is modeled by a C0 machine configuration co. To define the latter a relatively explicit, low level memory model in the style of [Nor98] is used. Memory frames mf are records with the following components:

- the number $mf.n$ of variables in mf,
- a function $mf.name$ mapping variable numbers $i \in [0 : mf.n-1]$ to their names (not used for variables on the heap),
- a function $mf.ty$ mapping variable numbers to their type; we define the size of a memory frame $msize(m)$ as the number of simple values stored in it:

$$\mathit{mfsize}(mf) = \sum_{i=0}^{mf.n-1} \mathit{size}(mf.ty(i))$$

- a content function $mf.ct$ mapping indices $0 \leq i < \mathit{mfsize}(mf)$ to the corresponding simple values.

A C0 machine configuration co is a record with the following components:

- the program rest $co.pr$ being the sequence of C0 statements to be executed,
- the current recursion depth $co.rd$,
- the local memory stack $co.lms$. It maps numbers $0 \leq i \leq co.rd$ to memory frames, i.e. $co.lms : \mathbb{N} \to mf$. The global memory is $co.lms(0)$. We denote the top-most local memory frame of a configuration co by $\mathit{top}(co) = co.lms(co.rd)$.
- a heap memory $co.hm$ also being a memory frame.

Parameters of the configuration that do not change during a computation are:

- the type table $co.tt$ containing information about types used in the program, and
- the function table $co.ft$ containing information about the functions of a program. It maps function names fn to pairs $co.ft(\mathit{fn}) = (co.ft(\mathit{fn}).ty, co.ft(\mathit{fn}).body)$ where $co.ft(\mathit{fn}).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $co.ft(\mathit{fn}).body$ specifies the function body.

8.2.1 C0 Variables

A variable v of configuration co is a pair $v = (mf, i)$ where mf is a memory frame of co and $i < mf.n$ is the number of the variable in the frame. The type of a variable (mf, i) is given by $ty(mf, i) = mf.ty(i)$.

Subvariables $S = (mf, i)s$ are formed from variables (mf, i) by appending a selector $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting array element number j or the form $s_i = .n$ for selecting the struct component

with name n. If the selector s is consistent with the type of (mf, i), then $S = (mf, i)s$ is a *subvariable* of (mf, i). Selectors are allowed to be empty.

In C0, pointers p may point to subvariables $(mf, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(mf, i)s$. Component $mf.ct$ stores the current values $va(co, (mf, i)s)$ of the simple subvariables $(mf, i)s$ in canonical order. Values of aggregate variables x are represented in $mf.ct$ by sequences of simple values starting from the abstract base address $ba(x)$ of variable x.

With the help of visibility rules and bindings the definition of va, ty, and ba from variables and subvariables can be easily extended to expressions e.

8.3 C0 Machine Transition Function

Eventually we want to consider several programs running as user processes under an operating system. The computations of these programs are interleaved. Therefore we need a compiler correctness statement based on small-steps structured operational semantics [Win93, NN99].

Here we do not give the full definition of the (small-step) transition function δ_{co} mapping C0 configurations co to their successor configuration:

$$co' = \delta_{\text{co}}(co)$$

The interested reader may consult [Lei08]. However, as an example, we give a partial definition of the $while$ statement and the function call semantics.

The first statement of a statement list sl is denoted by $hd(sl)$ and the remaining statements by $tl(sl)$.

Assume the head of the current program rest is a $while$ loop, i.e. $hd(co.pr) = while(cond)\{a\}$. If the condition $cond$ is satisfied in configuration co, then the statement list a is executed before the condition is checked once again. Otherwise we continue with the tail of the old program rest:

$$hd(co.pr) = while(cond)\{a\} \Rightarrow co'.pr = \begin{cases} a;\ co.pr & \text{if } va(co, cond) = 1 \\ tl(co.pr) & \text{otherwise} \end{cases}$$

Assume the head of the current program rest is a call of function *fn* with parameters $e_1, \ldots, e_n$ assigning the function's result to variable v, formally $hd(co.pr) = (v = f(e_1, \ldots, e_n))$. In the new program rest, the recursion depth is incremented and the function-call statement is replaced by the body of the function f taken from the function table $co.ft$. Furthermore, the values of all parameters e_i are stored in the new top local memory frame $top(co')$ by updating its content function at the corresponding positions:

$$hd(co.pr) = (v = fn(e_1, \ldots, e_n)) \Rightarrow$$
$$\begin{cases} co'.rd & = co.rd + 1 \\ co'.pr & = (co.ft(fn).body; tl(co.pr)) \\ top(co').ct_{size(ty(co,e_i))}(ba(co', e_i)) & = va(co, e_i) \quad \text{for } 1 \le i \le n \end{cases}$$

8.4 Compiler Correctness Theorem

The compiler correctness theorem uses the simulation relation $cosim(aba)(co, isa)$ between a C0 machines configuration co and an ISA configuration isa that runs the compiled program. The relation is parameterized by an allocation function aba mapping subvariables S of the C0 machine to their allocated base addresses $aba(co, S)$ in the ISA machine. The allocation function may change during a computation in two situations:

- if the recursion depth and thus the set of local variables changes due to function calls and function returns, and
- if reachable variables are moved on the heap during garbage collection (not yet implemented).

Notice however that in the first case only the range of the allocation function is changed. For C0 configurations co and local or global (sub) variables x the allocated base address $aba(x, co)$ depends only on co.

The simulation relation $cosim(aba)(co, isa)$ consists essentially of the following five conditions:

1. value consistency $v\text{-}cosim(aba)(co, isa)$: This condition states that reachable elementary subvariables x have the same value in the C0 machine and in the ISA machine. Let $asize(x)$ be the number of bytes needed to store a value of type $ty(x)$. Then we require $isa.m_{asize(x)}(aba(co, x)) = va(co, x)$.

2. pointer consistency $p\text{-}cosim(aba)(co, isa)$: This predicate requires for reachable pointer variables p pointing to a subvariable y that the value stored at the allocated address of variable p in the ISA machine is the allocated base address of y, i.e. $isa.m_4(aba(co, p)) = aba(co, y)$. This induces a subgraph isomorphism between the reachable portions of the heap in the C0 machine and the memory in the ISA.

3. control consistency $c\text{-}cosim(co, isa)$: This condition states that the delayed PC of the ISA configuration (used to fetch instructions) points to the start of the translated code of the program rest $co.pr$ of the C0 machine. Let the address of the first assembly instruction that is generated for a statement s be denoted by $caddr(s)$. Then we require $isa.dpc = caddr(hd(co.pr))$ and $isa.pc = isa.dpc + 4$.

4. code consistency $code\text{-}cosim(co, isa)$: This condition requires that the compiled code of the C0 program be stored in the ISA configuration isa starting at the code start address $cstart$. Thus it requires that the compiled code be not changed during the computation of the physical machine. We thereby forbid self modifying code.

5. stack consistency $s\text{-}cosim(co, isa)$: This is a technical condition about stack pointers, heap pointers etc. which does not play an important role here. For more details see [Lei08].

Based on the simulation relation $cosim(aba)(co, isa)$ we state the compiler correctness theorem.

Theorem 4 (Compiler Correctness) *Let the C0 machine configuration and the ISA configuration be consistent initially, i.e.* $cosim(aba^0)(co^0, isa^{s(0)})$.

For every C0 machine computation there is an ISA computation, step numbers $s(i)$*, and a sequence of allocation functions* aba^i *such that for all steps* i*:*

$$cosim(aba^i)(co^i, isa^{s(i)})$$

A formal proof of this statement for a non optimizing compiler, formalized and carried out in Isabelle-HOL [NPW02], is reported in [Lei08]. There exists an implementation of the same compilation algorithm written in C0. A formal proof that the C0 implementation simulates the Isabelle-HOL implementation is reported in [Pet07].

To solve the bootstrap problem [Ver08a] the C0 version of the compiler can be translated by an existing compiler into DLX code. The fact that the target DLX code simulates the source code can be shown using translation validation [PSS98]. Further solutions are discussed in [Lei08].

9 Inline Assembly Code ($C0_A$)

Recall that processor registers, I/O ports and user processes are not visible in the C0 variables of an operating system kernel written in C0. Hence we extend the language and permit sequences u of inline assembly instructions, as done in [KP07a, ST08].

We extend the language C0 by a statement of the form $asm(u)$ and call the resulting language $C0_A$. In $C0_A$ the use of inline assembly code is restricted as follows:

- only a certain subset of DLX instructions is allowed (e.g. no load or store of bytes or half words, only *relative* jumps).
- the target address of store word instructions must be outside the code and data regions of the $C0_A$ program or it must be equal to the allocated base address of a subvariable of the $C0_A$ program with type int or $unsigned\ int$ (this implies that inline assembly code cannot change the stack layout of the $C0_A$ program).
- certain registers (e.g. the stack pointer) must not be changed.
- the last assembly instruction in u must not be a jump or branch instruction.
- the execution of u must terminate.
- the target of jump and branch instructions must not be outside the code of u.
- the execution of u must neither generate misalignment nor illegal instruction interrupts.

To argue about the correctness of $C0_A$ programs we must define the semantics of the newly introduced statement. A store word instruction of inline assembly code can overwrite a C0 variable x. Hence we have to specify the effect of such a store

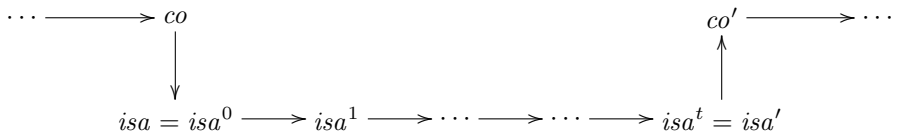

Figure 7: Execution of Inline Assembly Code

instruction on the value of x in the C0 machine configuration. This is easily done with the help of the allocated base address function aba that was introduced in the previous section.

Thus consider a $C0_A$ configuration co with program rest $co.pr = asm(u); r$. We will use the ISA semantics to model the execution of the inline assembly portion.

Using the compiler correctness statement from Section 8.4 we obtain an ISA configuration isa which is consistent with the given C0 machine configuration co, i.e. $cosim(aba)(co, isa)$. In addition to the allocated base address function aba the configuration isa is taken as an input parameter for the $C0_A$ transition function δ_{co_A}.

As depicted in Figure 7, the execution of the inline assembly sequence u leads to a physical machine computation $(isa = isa^0, \ldots, isa^t = isa')$ with $isa'.dpc = caddr(tl(co.pr))$ and $isa'.pc = isa'.dpc + 4$ by the restrictions on inline assembly code.

Let $j < t$. If the store word predicate $sw(isa^j)$ holds, the instruction executed in configuration isa^j updates the memory word at address $ea(isa^j)$ with the value $v = isa^j.gpr(RD(isa^j))$ as defined in Section 4.

Given that there exist one or more store word instructions in the sequence u writing to the same effective address $ea(isa)$, then the predicate $lastsw(u, isa)$ indicates the last instruction in the sequence u doing so. If the effective address is equal to the allocated base address of some C0 variable x, then the corresponding variable is updated in the new C0 machine configuration co' such that $va(co', x) = v$:

$$(hd(co.pr) = asm(u)) \wedge lastsw(u, isa^j) \wedge (ea(isa^j) = aba(co, x)) \Rightarrow va(co', x) = isa^j.gpr(RD(isa^j))$$

The result of the $C0_A$ transition function is defined by $(co', isa') = \delta_{co_A}(aba)(co, isa)$. Note that this definition keeps the two configurations consistent:

Lemma 2 ($C0_A$ Correctness) *If the current program rest of the C0 machine configuration co starts with an inline assembly statement $asm(u)$, then the compiler consistency is preserved after the execution of u:*

$$cosim(aba)(co, isa) \Rightarrow cosim(aba)(\delta_{co_A}(aba)(co, isa))$$

The proof of this lemma uses the restrictions on the inline assembly code. More details will be reported in [Tsy08].

10 Communicating Virtual Machines (CVM)

In this section we introduce communicating virtual machines (CVM). It is a model of a generic operating system kernel interacting with a fixed number of user processes [GHLP05, KP07a, IdRT08].

The kernel that is presented here is not preemptive, i.e. its computation can only be interrupted by a reset.

CVM uses the C0 language semantics to model computations of the (abstract) kernel and virtual machines to model computations of user processes. It is a pseudo-parallel model in the sense that in every step of computation either the kernel or one user process can progress.

From a kernel implementor's point of view, CVM encapsulates the low-level functionality of a microkernel and provides access to it as a library of functions, the so-called CVM primitives.

In the following sections we define CVM configurations, CVM computations, and show how abstract kernels implement system calls as regular C0 function calls. The implementation of the CVM model itself including the corresponding simulation theorem is presented later on in Section 10.4.

Note that this is the first detailed report on the CVM that includes the treatment of CVM device primitives.

10.1 CVM Configuration

The number of user processes running under CVM is denoted by np. Thus a process number pn is given by:

$$pn \in Pn = [1 : np]$$

A CVM configuration cvm is split into a processor component $cvm.p$ and a device component $cvm.d$. The processor component $cvm.d$ consists of:

- user processes are modeled by virtual machines (see Section 7). The configuration of all user processes is combined in a mapping $cvm.p.vm : Pn \to vm$.

  To model dynamic memory allocation each user process has its individual page table length $cvm.p.vm(u).ptl \in \mathbb{B}^{20}$.

  Furthermore, each user process has its individual status register $cvm.p.vm(u).sr$.

- a C0 machine configuration $cvm.p.co$ represents the so-called *abstract kernel*.

- the component $cvm.p.cp$ indicates the current process, i.e. $cvm.p.cp = 0$ means that the kernel is running while $cvm.p.cp = u$ where $u \in Pn$ means that user process u is running.

The device component $cvm.d$ is a stable device configuration as defined in Section 6.1.

We require the kernel configuration, in particular its initial configuration, be in a certain form:

- some special functions $f \in CVMP$, the CVM primitives, must be declared only, i.e. their body must be empty; their arguments and effects are described below.
- a special function called $kdispatch$ must be declared. Further details regarding this function are given below.

10.2 CVM Transition Function

The CVM transition function δ_{cvm} takes a CVM configuration cvm and returns an updated configuration:

$$cvm' = \delta_{cvm}(cvm)$$

In the definition we split cases on the current process component $cvm.p.cp$.

Note that this transition function is only used for a bounded computation in which the device behaves like an ordinary RAM and does neither take external input nor produce any external output (see Section 17).

10.2.1 User Transition

If the current process component $u = cvm.p.cp$ is $u \neq 0$ then user process $vm(u)$ does a step:

$$cvm'.vm(u) = \delta_{isa}(cvm.p.vm(u))$$

If no interrupt needs to be handled then user process $vm(u)$ keeps running:

$$\neg JISR(cvm.p.vm(u), eev(cvm.d)) \Rightarrow cvm'.cp = u$$

Otherwise the execution of the abstract kernel starts. Recall from Section 4.6 on interrupt semantics that in case of an interrupt the masked cause register is saved into the exception cause register eca and that the data necessary for handling the exception is stored in the register $edata$.

The entry point in the kernel is the function $kdispatch$ that is called with the exception cause $eca(cvm) = mca(cvm.p.vm(u), eev(cvm.d))$ and the exception data $edata(cvm) = ed(cvm.p.vm(u), eev(cvm.d))$ as parameters. The current process component and the kernel's recursion depth are set to zero:

$$\begin{aligned} cvm'.cp &= 0 \\ cvm'.p.co.rd &= 0 \\ cvm'.p.co.pr &= (v = kdispatch(eca(cvm), edata(cvm))) \end{aligned}$$

10.2.2 Kernel Transition

Initially (after power-up) and after an interrupt, the kernel execution starts with a call of the function $kdispatch$. The kernel is executed if $cvm.p.cp = 0$ holds. Then we further case-split on the head of the kernels program rest:

If the head of the program rest is a normal C0 statement, i.e. $hd(cvm.p.co.pr) \neq (v = f(e_1, \ldots, e_n); r)$ where $f \in CVMP$, it is executed using the regular C0 semantics from Section 8.3:

$$cvm'.co = \delta_{\text{co}}(cvm.p.co)$$

Otherwise the CVM primitive $hd(cvm.p.co.pr) = (v = f(e_1, \ldots, e_n); r)$ is executed where $f \in CVMP$. Below we define the semantics of a few selected primitives extending existing literature [ST08]. We ignore any preconditions or border cases[1]:

- the *start* primitive $v = start(e)$ hands control over to the user process specified by the current value of expression e:

  $$cvm'.cp = va(cvm.p.co, e)$$

  By this definition, the kernel stops execution and is only restarted again on the next interrupt (with a fresh program rest as described above).

- the *alloc* primitive $v = alloc(u, x)$ increases the memory size of user process $U = va(cvm.p.co, u)$ by $X = va(cvm.p.co, x)$ pages:

  $$cvm'.vm(U).ptl = cvm.p.vm(U).ptl + X$$

  The new pages are cleared:

  $$\forall a \in [(cvm.p.vm(U).ptl \cdot 4K) : (cvm.p.vm(U).ptl + X) \cdot 4K - 1]. \\ cvm'.p.vm(U).m(a) = 0^8$$

- the *free* primitive $v = free(u, x)$ freeing $X = va(cvm.p.co, x)$ pages of user process $U = va(cvm.p.co, u)$ is defined in a similar way.

- the *copy* primitive $v = copy(u_1, a_1, u_2, a_2, d)$ copies a memory region between user processes $U_1 = va(cvm.p.co, u_1)$ and $U_2 = va(cvm.p.co, u_2)$. The start addresses in the memory of the source process U_1 and the destination process U_2 are given by $A_1 = va(cvm.p.co, a_1)$ and $A_2 = va(cvm.p.co, a_2)$ respectively. The number of words to be copied is given by $D = va(cvm.p.co, d)$:

  $$cvm'.p.vm(U_2).m_{4 \cdot D}(A_2) = cvm.p.vm(U_1).m_{4 \cdot D}(A_1)$$

- the *input* primitive $v = input(u, a)$ updates the value of v with the memory content of user process $U = va(cvm.p.co, u)$ starting at address $A = va(cvm.p.co, a)$. The number of words to be copied is given by the size of the type of v denoted by $D = size(ty(v))$:

  $$va(cvm'.p.co, v) = cvm.p.vm(U).m_{4 \cdot D}(A)$$

- the *output* primitive $v = output(x, u, a)$ copies the value of x into the memory of user process $U = va(cvm.p.co, u)$ starting at address $A = va(cvm.p.co, a)$. The number of words to be copied is given by the size of the type of x denoted by $D = size(ty(x))$:

  $$cvm'.p.vm(U).m_{4 \cdot D}(A) = va(cvm.p.co, x)$$

[1] Note that there exist a lot of those but in their very nature each one is quite simple (compare [ST08]).

- the *input device* primitive $v = inputDev(a)$ updates the value of v with the content of the device memory starting at address $A = va(cvm.p.co, a)$. The number of words to be copied is given by the size of the type of v denoted by $D = size(ty(v))$:

$$va(cvm'.p.co, v) = cvm.d.m_D(A)$$

- the *output device* primitive $v = outputDev(x, a)$ copies the value of x into the memory of the device starting at address $A = va(cvm.p.co, a)$. The number of words to be copied is given by the size of the type of x denoted by $D = size(ty(x))$:

$$cvm'.d.m_D(A) = va(cvm.p.co, x)$$

  Note that due to the semantics of the device (see Section 6) the output device primitive might have side effects on the interrupt flag in the device.

- the primitive $v = getgpr(r, u)$ updates the value of v with the content of the general purpose register $R = va(cvm.p.co, r)$ of process $U = va(cvm.p.co, u)$:

$$va(cvm'.p.co, v) = cvm.p.vm(U).gpr(R)$$

- the primitive $v = setgpr(x, r, u)$ writes the current value of x into the general purpose register $R = va(cvm.p.co, r)$ of user process $U = va(cvm.p.co, u)$:

$$cvm'.p.vm(U).gpr(R) = va(cvm.p.co, x)$$

- the primitive $v = wait()$ does nothing but wait for an interrupt at which it reinitializes the kernel as defined in Section 10.2.1:

$$cvm'.p.co.pr = \begin{cases} cvm.p.co.pr & \text{if } \neg(\exists u.\ JISR(cvm.p.vm(u), eev(cvm.d))) \\ (v = kdispatch(eca(cvm), edata(cvm))) & \text{otherwise} \end{cases}$$

  Note that, as no user process is scheduled no internal interrupt can be generated. By this definition the CVM waits until the generation of an external interrupt.

After an interrupt the $kdispatch$ function is invoked in the kernel. Depending on the interrupt itself, the corresponding interrupt handler is called (as defined bellow). Furthermore, the body of the $kdispatch$ function implements the scheduling policy of the operating system. It is executed after the return from the interrupt handler. Eventually the *start* primitive must be invoked which again schedules a user process. This user process computes until the next interrupt is generated.

10.3 Binary Kernel Interface

Before going into details regarding the CVM implementation we show how user processes can invoke services provided by the operating system in form of system calls.

A user process can invoke a system call using the trap instruction that causes an internal interrupt (compare Section 4.5.1). If the kernel provides k trap handlers, then the user can specify the handler to be invoked using the immediate constant i being part of the trap instruction, where $\langle i \rangle \in [0 : k-1]$.

The kernel call definition function kcd maps immediate constants i to names of functions declared in the abstract kernel. Thus $kcd(i)$ is simply the name of the C0 function handling a trap with immediate constant i.

For each i, let $npar(i)$ be the number of parameters of function $kcd(i)$. We require that user processes pass the parameters for function $kcd(i)$ in general purpose registers $gpr[1 : npar(i)]$. With the specification of the functions $kcd(i)$ and $npar(i)$ the binary interface to the kernel is defined.

10.3.1 Binary Kernel Interface Implementation

To handle system calls the kernel maintains a variable CUP keeping track of the user process that is currently running or that has been running before the kernel execution started:

$$cvm.p.cp \neq 0 \Rightarrow va(cvm.p.co, CUP) = cvm.p.cp$$

Assume $cvm.p.cp = u$ where $u \neq 0$ and user process $cvm.p.vm(u)$ executes the trap instruction with immediate constant i. The trap instruction activates internal event signal $iev(5)$ as described in Section 4.5.1. Furthermore, assume that no interrupts with higher priority (lower index) are active simultaneously. Then the masked cause vector $0^{26}10^5$ is saved into the exception cause register $eca[31 : 0]$ and parameter i is saved into the exception data register $edata$:

$$\begin{array}{rcccl} eca & = & mca(cvm'.vm(u)) & = & 0^{26}10^5 \\ edata & = & ed(cvm'.vm(u)) & = & i \end{array}$$

According to the CVM semantics the abstract kernel is started with the function call $kdispatch(eca, edata)$. By a case split on eca the $kdispatch$ function conclude that a trap needs to be handled. Hence it invokes the function $f(e_1, \ldots, e_{npar(i)})$, where $f = kcd(i)$ using the parameters computed by the assignment $e_i = getgpr(i, CUP)$.

The effect of a system call is summarized in the following lemma: The intended handler is called with the intended parameters.

Lemma 3 (System Call Semantics) *Let cvm be the CVM configuration before the kernel is entered and let cvm'' be the CVM configuration after the execution of the kcd function. Then we can derive from the semantics of CVM and C0:*

$$\begin{array}{rcll} cvm''.co.rd & = & cvm.p.rd + 1 & = 1 \\ cvm''.co.pr & = & cvm.p.co.ft(f).body; r & \textit{for some } r \\ top(cvm'').ct(j) & = & cvm.p.vm(u).gpr(j) & \textit{for all } j \in [1 : npar(i)] \end{array}$$

This lemma formalizes the idea that an interrupt is like a function call of the handler. Comparing with the C0 semantics in Section 8 we see that the trap instruction indeed formally causes a function call of the handler. The function call is however remote, because it is executed by a process (the abstract kernel) different from the calling user process (a virtual machine).

10.4 CVM Implementation

So far we have argued mathematically only about the configurations of the abstract kernel $cvm.p.co$. Now we also argue about its source code that we denote by sak. The source code of the so-called *concrete kernel*, denoted by sck, is obtained by linking sak with the source code of some CVM implementation $scvm$ using a link operator ld:

$$sck = ld(sak, scvm)$$

Note that sak is a pure C0 program, whereas $scvm$ and sck are $C0_A$ programs.

The function table of the linked program sck is constructed from the function tables of the input programs as follows. For functions present in both programs, *defined functions* (with a non-empty body) take precedence over *declared functions* (with an empty body). We do not formally define the ld operator here; it may only be applied under various restrictions concerning the input programs, e.g. the names of global variables of both programs must be distinct, function signatures must match, and no function may be defined in both input programs. More details will be given in [IdR08].

We require that the abstract kernel sak defines $kdispatch$ and declares all CVM primitives while the CVM implementation $scvm$ defines the primitives and declares $kdispatch$.

10.5 Concrete Kernel Simulation Relation

We define a relation $kcosim(kalloc)(co, cc)$ stating that the abstract kernel configuration co is coded by the concrete kernel configuration cc. The function $kalloc$ maps subvariables x of the abstract kernel configuration co to subvariables $kalloc(x)$ of the concrete kernel configuration cc.

Linking is less complex than compiling. The definition of the $kcosim$ relation has only three parts:

1. elementary variable consistency $e\text{-}kcosim(kalloc)(co, cc)$: All reachable elementary (sub) variables x of the abstract kernel configuration co and the values of x in the concrete kernel cc coincide:

$$va(co, x) = va(cc, x)$$

2. pointer consistency $p\text{-}kcosim(kalloc)(co, cc)$: The function $kalloc$ is a graph isomorphism between reachable portions of the heaps. For all reachable pointer variables p of abstract kernel configuration co, pointing to subvariable v, the following holds:

$$(va(co, p) = v) \Rightarrow va(cc, p) = kalloc(v)$$

3. code consistency $c\text{-}kcosim(kalloc)(co, cc)$: The program rest of the concrete kernel is a prefix of the program rest of the abstract kernel (see [KP07a, IdRT08]).

10.5.1 Concrete Kernel Data Structures

The CVM implementation maintains data structures for the simulation of the virtual machines, i.e. for the support of multiprocessing. These include:

- an array of process control blocks $pcb[u]$ for the kernel ($u = 0$) and the user processes ($u \in Pn$). Process control blocks are structs with components $pcb[u].R$ for every processor register R of the *physical* machine.
- a single integer array $ptarray$ on the heap holds the page tables of all user processes in the order of the process numbers u. The function $ptbase(u)$ defines the start index of the page table for process u:

  $$ptbase(u) = \sum_{j<u}(pcb[j].ptl + 1)$$

  Virtual addresses are split into the page index $vadr.px = vadr[31 : 12]$ and the byte index $vadr.bx = vadr[11 : 0]$. Since the C0 array $ptarray$ is indexed by words and not by bytes we can define the page table entry for a virtual address $vadr$ of process u as the following C0 expression (compare 7.2):

  $$pte(u, vadr) = ptarray[ptbase(u) + vadr.px]$$

  The physical memory address and the valid bit are then defined by the following C0 expression:

  $$\begin{aligned} pma(u, vadr) &= pte(u, vadr)[31 : 12] \circ vadr.bx \\ v(u, vadr) &= pte(u, vadr)[11] \end{aligned}$$

  We require that the compiler computes the allocated base address of the array $ptarray$ as a multiple of the page size $4K$.
- data structures (in the simplest case doubly-linked lists) for the management of physical and swap memory (including victim selection for page faults).
- the variable CUP keeping track of the current user process and thus encoding the $cvm.p.cp$ component (unless the kernel is running).

10.5.2 Entering System Mode after an Interrupt

When the mode bit in the concrete kernel flips from user to system mode, the program rest is initialized with $init_1; init_2$. In all cases except for a reset, the first Part $init_1$ will write all processor registers R to the process control block $pcb[CUP].R$ of the user process CUP that was interrupted. Furthermore, it will restore the registers of the kernel from process control block $pcb[0]$.

In the second Part $init_2$, the CVM implementation detects if the interrupt was due to a page fault or to other causes. Page faults are handled silently without calling the abstract kernel. For other interrupts, the $kdispatch$ function is called with the parameters obtained from the C0 array $pcb[CUP]$, i.e.:

$$kdispatch(pcb[CUP].eca, pcb[CUP].edata)$$

10.5.3 Leaving System Mode

An invocation of the $start$ primitive will switch to user mode again. The primitive is implemented using inline assembly code. The physical processor registers are written to $pcb[0]$ to save the state of the concrete kernel. Then the physical processor registers for process CUP are restored from $pcb[CUP]$. Finally, a return from exception instruction rfe is executed.

10.5.4 Process Virtualization

To argue about multiple user processes, we slightly extend the B relation introduced in Section 7.4. The extended relation $B(u)(cvm, cc, isa)$ states that the user process $cvm.p.vm(u)$ of the CVM configuration cvm is coded by the concrete kernel configuration cc and the physical machine configuration isa:

1. processor registers of user process u are stored in the physical processor registers, if process u is running; otherwise they are stored in the corresponding process control block:

$$cvm.p.vm(u).gpr(r) = \begin{cases} isa.gpr(r) & cvm.p.cp = u \\ va(cc, pcb[u].gpr(r)) & \text{otherwise} \end{cases}$$

2. the memory content of user process u is stored in the physical memory at the corresponding physical memory address:

$$cvm.p.vm(u).m(a) = isa.m(pma(u, a))$$

10.5.5 Implementation of the CVM Primitives

The implementation of CVM primitives like $e = getgpr(u, r)$ and $e = setgpr(u, r, g)$ is straightforward using simple assignments:

$$e = pcb[u].gpr(r))$$
$$pcb[u].gpr(r) = g$$

For the CVM primitives $alloc$ and *free* the page table length of the process has to be increased or decreased and lots of page table entries in $ptarray$ above the portion of the modified user process have to be moved around in the page table array. Various other data structures concerning memory management have to be adjusted as well. Since the page tables are accessible as a C0 data structure, inline assembly code is only required to clear newly allocated physical pages.

Similarly, the implementation of the *copy*, *input*, and *output* primitive requires assembly code to copy data between the kernel, user processes and devices.

The primitives *input device* and *output device* are implemented using simple load word and store word operations to device addresses.

The primitive *wait* is somehow special. It is used to await the next timer interrupt. Therefore, the primitive implementation is split in the following parts:

$$init\_idle; \quad a : jump\ a; \quad a + 4 : noop$$

Within $init\_idle$ the timer interrupt, which is normally disabled during the kernel execution, is enabled by setting the status register to 0^{32}. Then an idle loop is entered which consist of a self-loop to address a and a $noop$ instruction[2]. The latter fills the delay slot that occurs due to the delayed branching (see Section 4.1). Once the idle loop is executed the configuration is not altered until an interrupt is generated. Further details regarding this primitive will be given in [Tsy08].

10.6 CVM Correctness Theorem

The correctness proof of the CVM deals simultaneously with computations in three computational models:

- the CVM consisting of a C0 machine and several virtual machines; configurations are denoted by cvm.
- an intermediate model for the $\mathrm{C0_A}$ computation of the concrete kernel; configurations are denoted by cc
- the physical machine model; configurations are denoted by $isad$

Although we quantify in the following theorem over all all steps i of the CVM computation, we will apply it only for a bounded computation in which the device indeed behaves like ordinary RAM (see Section 19).

Theorem 5 (CVM Correctness) *Let the CVM simulation relation hold initially, i.e. assume there exists a concrete kernel cc such that $kcosim(kalloc^0)(cvm^0.p.co, cc^0)$ as well as $cosim(aba^0)(cc^0, isad^0.p)$ and $B(u)(cvm^0, cc^0, isad^0.p)$ hold.*

For all steps i of the CVM computation, there is a concrete kernel configuration, a physical machine configuration, two sequences of allocation functions aba^i and $kalloc^i$ and two sequences of step numbers $s(i)$ and $t(i)$ such that:

- *the abstract kernel configuration $cvm^i.p.co$ is coded by the concrete kernel configuration $cc^{s(i)}$:*
$$kcosim(kalloc^i)(cvm^i.p.co, cc^{s(i)})$$
- *the concrete kernel configuration $cc^{s(i)}$ is coded by the physical machine configuration $isad^{t(i)}$. Recall that on the physical machine the compiled concrete kernel is executed:*
$$cosim(aba^i)(cc^{s(i)}, isad.p^{t(i)})$$
- *the configurations of the user-processes $cvm^i.vm(u)$ are coded by the concrete kernel configuration $cc^{s(i)}$ and the physical machine configuration $isad^{t(i)}$:*
$$B(u)(cvm^i, cc^{s(i)}, isad.p^{t(i)})$$

[2]If not supported by the ISA a $noop$ instructions can be faked, e.g. using an add immediate instruction with an immediate constant that equals 0.

- *the configuration of the device is the same.*

$$cvm^i.d = isad^{t(i)}.d$$

The theorem is proven by induction on the steps i of a CVM computation. Depending on the current process number $cvm^i.cp$ and the interrupts occurring, the proof uses compiler correctness (see Section 8.4), the correctness of the memory management mechanisms (see Section 7.4), and detailed arguments about inline assembly code based on the $C0_A$ semantics (see Section 9). A formal version of this theorem is reported in [IdRT08]. More elaborated versions will follow in [Tsy08, IdR08].

Note that the concrete kernel is only used as an intermediate model to modularized the argumentation regarding the CVM correctness. In the remainder of this thesis we will use the shorthand $cvmsim(isad, cvm)$ to refer to the CVM simulation relation.

11 Automotive Bus Controller (ABC)

Our real-time operating system includes drivers for a FlexRay-like interface called automotive bus controller (ABC). Thus we introduce a specification for a single ABC before we argue about the operating system itself in the next section. To do so, we specialize the stable device configuration from Section 6.1.

We are dealing with a time-triggered system [Rus99]. Time is divided into so-called rounds. Each round consists of ns many slots. Thus a slot number s is given by:

$$s \in Sn = [0 : ns - 1]$$

Given a slot number s the predecessor $s - 1$ and the successor $s + 1$ are computed modulo ns.

Messages msg, to be broadcast by the ABC, are bit-stings consisting of ml many bytes:

$$msg \in Msg = \mathbb{B}^{8 \cdot ml}$$

We require that ml is a multiple of 4 such that only complete words are being broadcast.

An ABC needs to know whether it is the sender in a given slot. This knowledge is represented by the local schedule called FlexRay table ft:

$$ft : Sn \to \mathbb{B}$$

It indicates for each slot number s in a round if the ABC is the sender in s or not. The schedule is repeated each round again and again.

11.1 Stable ABC Specification Configuration

We reuse the letter d to denote a stable ABC specification configuration. It consists of:

- a memory $d.m$ addressing K words. The first $ml/4$ words are used as a send buffer which we denote by $d.sb = d.m_{ml/4}(0)$. The next $ml/4$ words are used as a receive buffer which we denote by $d.rb = d.m_{ml/4}(ml/4)$. The remaining words are used for configuration parameters, e.g. the local schedule which we denote by $d.ft$.
- an interrupt-flag $d.int$ indicates if an interrupt is still pending, i.e. it was generated but has not been cleared by the drivers so far.

This stable ABC specification configuration behaves like an ordinary RAM and does neither take external input nor produce any external output (compare Section 6.1). Note that this specification is partial in the sense that it models the behavior of the ABC for a single slot only.

However, this configuration contains everything needed to argue about the operating system OLOS. A complete specification for several ABCs is given in Section 16.

12 OSEKtime Like Operating System (OLOS)

Based on the stable ABC specification configuration described in the previous section we specify the OSEKtime-like [OSE08b] operating system OLOS [Kna05, IdRK05, Kna08]. It features time-triggered scheduling, process separation, buffered communication, and drivers for the ABC.

OLOS itself is not preemptive. It operates in a time-triggered fashion like the ABC. In fact, the ABC signals the start of a new slot by generating an interrupt. OLOS rounds might be multiples of ABC rounds. In each slot one application is scheduled.

For this thesis one OLOS round corresponds to one ABC round; slots in OLOS directly correspond to ABC slots. This restriction preserves the idea of time-triggered systems and thus does not limit the applicability of the presented ideas to real systems.

Applications running under OLOS can communicate via system calls that update or read a message buffer, i.e. an application visible data structure of the operating system. Note that the message buffer is similar to the FTCom buffer [OSE08a] of the OSEKtime OS.

In addition, OLOS features a strict separation of user processes as well as drivers for the ABC.

12.1 Message Buffers, Applications and Schedules

The messages being broadcast are typed. As already mentioned the applications are executed according to a fixed schedule. All applications are known initially. Thus an upper bound for the different message types being broadcast in the system can be fixed. Let this bound be given by nmt. A message number mn is then given by:

$$mn \in Mn = [0 : nmt - 1]$$

OLOS maintains an array for storing messages that is called message buffer mb. For each message type this array keeps the latest message that has been broadcast:

$$mb : Mn \rightarrow Msg$$

The applications running under OLOS are C0 machines co as defined in Section 8.2. As already mentioned in Section 10.1 the number of processes running under the operating system is bounded by np thus a process number pn is given by:

$$pn \in Pn = [1 : np]$$

We combine the state of all applications into a process mapping pm. For all process numbers pn the process mapping $pm(pn)$ maps to the corresponding C0 machine configuration co.

In each slot in a round one application is scheduled according to a fixed mapping called the scheduling table st:

$$st : Sn \to Pn$$

In slot s the process $pm(st(s))$ is executed. The mapping is the same in each round. Thus we get a complete cyclic behavior.

In addition to the communication schedule the operating system needs to know which message type is being broadcast. This is specified by the slot content function sc:

$$sc : Sn \to Mn$$

It simply maps a slot number to the index in the message buffer of the message to be broadcast.

12.2 OLOS Configuration

An OLOS configuration $olos$ has the following components:

- the process mapping $olos.p.pm$ combines the state of all applications.
- the message buffer $olos.p.mb$ stores for each message type the latest message value that has been broadcast.
- the scheduling table $olos.p.st$ defines which application is being scheduled in a given slot.
- the scheduling table $olos.p.ft$ defines the sender in a given slot.
- the slot content $olos.p.sc$ defines the type of the message being broadcast in a given slot.
- a send flag $olos.p.send \in \mathbb{B}$ indicates if the OLOS instance is the sender in the next slot.
- an idle flag $olos.p.idle \in \mathbb{B}$ indicates if either the kernel ($olos.p.idle = 1$) or the scheduled application ($olos.p.idle = 0$) is executed.
- the current slot number $olos.p.csn$.
- an ABC specification configuration $olos.d$.

To specify the system calls offered by OLOS we introduce the concept of local configurations first.

12.3 Local Configurations (LC) and Transitions

A local configuration lc is a tuple with the following components:

- a C0 machine configuration $lc.co$, and
- a message buffer $lc.mb$.

OLOS offers three system calls. The predicate $is\_syscall(lc)$ indicates if the head of the program-rest in C0 is a system call. Thus it checks if the head of the program rest is either a $ttSend$ system call, i.e. $hd(lc.co) = (v = ttSend(mn, pmsg))$, or a $ttRecv$ system call, i.e. $hd(lc.co) = (v = ttRecv(mn, pmsg))$ or a $ttExFinished$ system call, i.e. $hd(lc.co) = (v = ttExFinished())$.

The transition function δ_{lc} of a local configuration splits cases depending on the head of the program rest in the C0 machine configuration $lc.co$. If the head of the program rest is a system call, i.e. $is\_syscall(lc)$ holds, the call is executed with the following semantics:

If a $ttSend$ system call is invoked, i.e. $hd(lc.co.pr) = ttSend(mn, pmsg)$, the message buffer is updated at position mn with the message given by the dereferenced message pointer $pmsg$. Furthermore, the call is deleted from the program rest:

$$\begin{aligned} lc'.mb(va(lc.co, mn)) &= va(lc.co, {}^*pmsg) \\ lc'.co.pr &= tl(lc.co.pr) \end{aligned}$$

If a $ttRecv$ system call is invoked, i.e. $hd(lc.co.pr) = ttRecv(mn, pmsg)$, the message given by $pmsg$ is updated with the value of the message buffer at position mn. The call is deleted from the program rest:

$$\begin{aligned} va(lc'.co, {}^*pmsg) &= lc.mb(va(lc.co, mn)) \\ lc'.co.pr &= tl(lc.co.pr) \end{aligned}$$

If a $ttExFinished$ system call is invoked, i.e. $hd(lc.co.pr) = ttExFinished()$, the call is simply deleted from the program rest. From the point of view of the application the $ttExFinished$ system call has no effect. It just indicates that the application has finished execution and wants to return the control back to the operating system:

$$lc'.co.pr = tl(lc.co.pr)$$

We will come back to this in Section 12.4.2.

If the head of the program rest is not a system call, i.e. $is\_syscall(lc)$ does not hold, the C0 small-step transition function δ_{co} is applied to the application. The message buffer is left unchanged:

$$lc'.co = \delta_{\mathrm{co}}(lc.co)$$

Next we detail the transition function of the whole system.

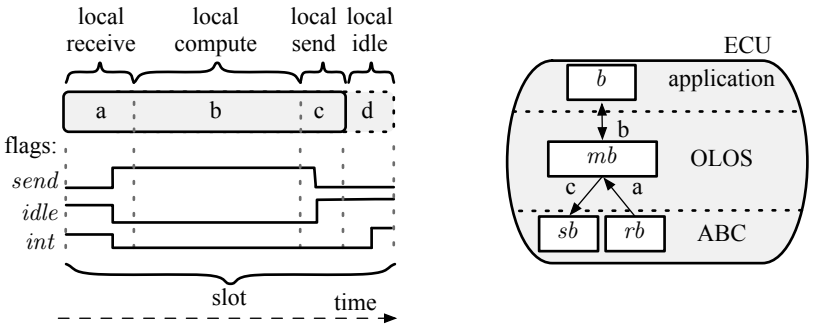

Figure 8: OLOS: Slot Partitioning and Data-Flow

12.4 OLOS Transition Function

The OLOS transition function δ_{olos} takes an OLOS configuration and returns an updated one, i.e. $olos' = \delta_{olos}(olos)$. We distinguish 4 phases:

(a) in the local receive phase the message buffer is updated with the message that has been broadcast in the previous slot and the interrupt flag of the ABC is cleared.

(b) in the local compute phase the currently scheduled application is executed.

(c) in the local send phase the current slot number is incremented and the sender in the next slot updates the send buffer of the ABC with the message to be broadcast.

(d) in the idle phase the next interrupt is awaited, the OLOS configuration is not changed at all.

Figure 8 illustrates the slot partitioning as well as the data-flow within the phases. Note that we use the indices (a), (b) and (c) throughout the rest of this thesis to refer to the corresponding phases.

To distinguish between the four phases we make use of the send-flag, the idle-flag and the ABC interrupt-register. The OLOS configuration $olos$ is within the local receive phase (a) if $olos.p.send = 0$ and $olos.p.idle = 1$ and $olos.d.int = 1$. The configuration is within the local compute phase (b) if $olos.p.idle = 0$. The configuration is within the local send phase (c) if $olos.p.send = 1$ and $olos.p.idle = 1$. The configuration is within the idle phase (d) if $olos.p.send = 0$ and $olos.p.idle = 1$ and $olos.d.int = 0$.

Next we will will case split according to the four phases. If none of the above cases applies, the configuration is not altered, i.e. $olos = \delta_{olos}(olos)$.

12.4.1 Local OLOS Receive Phase (a)

We denote the message number of the message that has been broadcast in the previous slot by $mn = olos.p.sc(olos.p.csn - 1)$.

If the $olos$ configuration is in the local receive phase the message buffer mb is updated with the message that has been broadcast in the previous slot. Furthermore, the interrupt is cleared, the idle-flag is set to 0 and the send-flag is set to 1. Then:

$$\begin{aligned} olos'.p.mb(mn) &= olos.d.rb \\ olos'.p.idle &= 0 \\ olos'.p.send &= 1 \\ olos'.d.int &= 0 \end{aligned}$$

Note that clearing the idle-flag leads to the local compute phase.

12.4.2 Local OLOS Compute Phase (b)

Let the process number of the application that is currently being scheduled be denoted by $pn = olos.p.st(olos.p.csn)$.

If the configuration $olos$ is in the local compute phase the LC transition function δ_{lc} is applied to the scheduled application; the ABC is left unchanged. In addition, if $ttExFinished$ is invoked by the application, the idle-flag is set:

$$(olos.p.pm(pn), olos.p.mb) = \delta_{lc}(olos.p.pm(pn), olos.p.mb)$$

$$hd(olos.p.pm(pn).pr) = (v = ttExFinished()) \Rightarrow olos'.p.idle = 1$$

Note that this definition enforces the scheduled application to be executed step by step until $ttExFinished$ is invoked which then will lead to the local send phase.

12.4.3 Local OLOS Send Phase (c)

Let the incremented slot number be denoted by $icsn = olos.p.csn + 1$ and let the type of the message to be broadcast in the next slot be denoted by $mn = olos.p.sc(icsn)$.

If the configuration $olos$ is in the local send phase the send-flag is cleared and the current slot number is incremented. Furthermore, if the configuration $olos$ is the sender in the next slot, i.e. if $olos.p.ft(icsn)$, the send buffer is updated with the message to be broadcast next:

$$\begin{aligned} olos'.p.send &= 0 \\ olos'.p.csn &= icsn \\ olos.p.ft(icsn) \Rightarrow olos'.d.sb &= olos.p.mb(mn) \end{aligned}$$

Note that clearing the send-flag will start the idle phase

12.4.4 Local OLOS Idle Phase (d)

During the the local idle phase the configuration simply stalls and is not altered at all:

$$olos = \delta_{olos}(olos)$$

In the remainder of this section we argue about a single slot only.

Formalization. Note that we have formalized the local OLOS model and the stable ABC model in the theorem prover Isabelle/HOL resulting in about 1500 lines of code.

12.5 OLOS Implementation Data Structures

OLOS is implemented by specializing the abstract CVM kernel. The C0 applications running under OLOS are compiled in order to obtain the CVM user processes.

Among others the kernel has to maintain the following variables and data structures:

- C0 implementations of the functions st, ft and sc denoted ST, FT and SC,
- an integer variable CSN keeping track of the current slot,
- an array $MB[0 : nmt - 1]$ capable of storing nmt messages,
- two boolean variables $Send$ and $Idle$ coding the corresponding flags.

Note that we have implemented an initial version of OLOS in C0. This implementation evolved over the years, e.g. due to changes in some CVM primitive signatures. In Section 12.7 we will describe the up-to-date OLOS implementation (maintained by Mareike Schmidt [Sch08]) along the lines of the OLOS correctness theorem.

12.6 OLOS Simulation Relation

Next we can define a simulation relation $olossim(aba(pn))(cvm, olos)$ between some instantiated CVM and some OLOS configuration. It is parameterized by a sequence aba of allocation functions $aba(pn)$, one for each process number pn:

1. the content of the OLOS message buffer is stored in the corresponding variables of the abstract kernel:

$$va(cvm.p.co, MB) = olos.p.mb$$

2. the OLOS flags correspond to the ones in the implementation:

$$\begin{aligned} va(cvm.p.co, Send) &= olos.p.send \\ va(cvm.p.co, Idle) &= olos.p.idle \end{aligned}$$

3. the interrupt flag in the device is the same:

$$cvm.d.int = olos.d.int$$

4. the user processes of CVM encode the applications running under OLOS:

$$cosim(aba(pn))(cvm.p.up(pn), olos.p.pm(pn))$$

5. the send buffer in the device is the same:

$$cvm.d.sb = olos.d.sb$$

6. the kernel keeps track of the current slot number:

$$va(cvm.p.co, CSN) = olos.p.csn$$

7. the receive buffer in the device is the same:

$$cvm.d.rb = olos.d.rb$$

8. the OLOS scheduling functions correspond to the one in the implementation:

$$\begin{array}{rcl} va(cvm.p.co, ST) & = & olos.p.st \\ va(cvm.p.co, FT) & = & olos.p.ft \\ va(cvm.p.co, SC) & = & olos.p.sc \end{array}$$

12.7 OLOS Correctness Theorem for One Slot

Assume the OLOS configuration is in phase (a) and all invariants hold. We construct the OLOS kernel such that the invariants are maintained during a whole slot. Note that in the two phases (a) and (c) the kernel is running; in phase (b) a user process is scheduled that might invoke system calls.

In phase (a) the ABC driver copies the content of the local receive buffer into the variable $MB[SC(CSN-1)]$ using the CVM $inputdevice$ primitive. Hence Part 1 of $olossim$ continues to hold after the local receive phase.

Furthermore, the send-flag is set and the idle-flag as well as the interrupt are cleared. Thus Part 2 and Part 3 hold after the local receive phase, too.

The next process to be scheduled is computed by $CUP = ST(CSN)$. An invocation of the CVM primitive $start(CUP)$ starts the compute phase (b).

During the compute phase (b) we have to worry about the scheduled process only. The handlers for the system calls $ttSend$ and $ttRecv$ are implemented with the help of the CVM $output$ and $input$ primitive such that parts 1 and 4 of $olossim$ continue to hold (details regarding the implementation of system calls were given in Section 10.3). Phase (b) ends with the system call $ttExFinished$ returning control to the kernel again. The kernel sets the idle-flag, i.e. $Idle = 1$, hence Part 2 holds after the local compute phase. This starts the send phase.

In phase (c) the kernel checks if it is the sender in the next slot, i.e. if $FT(CSN+1)$ equals 1. If this is the case it copies the content of variable $MB[SC(CSN+1)]$ into the local send buffer of the ABC (using the CVM $outputdevice$ primitive). This implies Part 5 of $olossim$. Furthermore, the send-flag is cleared and the kernel increments the current slot number CSN. Hence Part 2 and Part 6 of $olossim$ hold.

In phase (d) the kernel executes the $wait$ primitive. Thus the CVM configuration idles waiting until the interrupt flag in the ABC is raised which starts a new slot (see Section 10.5.5). So does the OLOS configuration.

Although we quantify in the following theorem over all all steps i of the OLOS computation, we will apply the theorem only for a bounded interval (see Section 20).

Theorem 6 (OLOS Correctness for One Slot) *Let the simulation relation between the CVM and OLOS hold initially, i.e.* $olossim(aba^0)(cvm^0, olos^0)$.

For every OLOS computation there is a CVM computation, step numbers $s(i)$*, and a sequence of allocation functions* aba^i *such that for all steps* i*:*

$$olossim(aba^i)(cvm^{s(i)}, olos^i)$$

This theorem is proven by induction on the steps i of the OLOS computation using the compiler correctness (see Theorem 4). A formal version of this theorem will be reported in [Sch08].

In the remainder of this thesis we will use the shorthand $olossim(cvm, olos)$ to refer to the OLOS simulation relation.

Part II
Distributed ECUs

So far we have considered a single ECU consisting of a processor and a device without external I/O only. In this part of the thesis we connect several instances of these ECUs and argue about the complete distributed system.

The verification of asynchronous communication systems must, at some point, deal with the low-level bit transmission between two automotive bus controllers (ABC) that are connected to the bus. The core idea is to ensure that the value broadcast on the bus is stable long enough so that it can be sampled correctly by the receiver. To stay within such a so-called sampling interval, the local clocks on the ABCs should not drift apart more than a few clock ticks and therefore need to be synchronized regularly. This is achieved by a message encoding that enforces the broadcast of special bit sequences used for synchronization.

The correctness of this low-level transmission mechanism cannot be carried out in a conventional, digital, synchronous model. It involves asynchronous and real-time-triggered register models taking setup- and hold-times as well as metastability into account. This part of the verification has been reported in [BBG$^+$05, KP07b, ABK08b].

Ensuring correct message transmission between two ABCs is only one part of the communication correctness. Let us consider a set of interconnected ABCs. To avoid bus contention only one ABC is allowed to broadcast at a time, all others should listen only. For that, time is divided into rounds, which are further subdivided into slots. A fixed schedule assigns a unique sender to a given slot number. The schedule is the same for each round. The gate-level implementation of the scheduler has to ensure that all ABCs have roughly the same notion of the slot start and end times, i.e. they must agree on the current sender and the transmission interval.

Due to drifting clocks a synchronization algorithm becomes necessary. We use a simple approach: a cycle offset is added at the beginning and end of each slot. This offset is chosen large enough to compensate the maximal clock drift that can occur during a full round. The local timers are synchronized only once, at the beginning of each round. This is done by choosing a distinguished master ABC which is the ABC sending in the first slot of a round.

The combination of the results into a lock-step and synchronous view of the system is now simple. The scheduler correctness ensures that at all time only one ABC sends and all other ABCs listen. Then we can conclude from the first part that the broadcast data is correctly received by all ABCs.

In contrast, others like [Pik07] have used purely digital models to argue about the scheduling correctness. However, abstract hardware models were considered only. The justification of the used abstractions for a concrete gate-level hardware implementation have not been reported, yet.

Note that we have defined the hardware in Isabelle/HOL [NPW02] being inspired by [Pau05]. Note that this implementation is based on boolean gates and sums up to 2800 lines of code. The resulting hardware definitions can be translated to Verilog and run on a FPGA. More details regarding the translation algorithm are reported in [AT08].

Overview of Part II

In the second part of this thesis we extend the results from [KP07a, Kna08].

In Section 13, we detail our ABC implementation that was inspired by [Pau05]. Then we show how to verify this implementation based on [BBG+05, Sch06, KP07b, ABK08b]. The lemmata regarding message transmission and clock synchronization are sketched in Sections 14 and 15. In particular, we review the clock synchronization mechanism from [KP07b, ABK08b].

In Section 16, we specify the communication behavior. To do so, we define a general distributed framework (DISTR) that models the interaction between the ECUs using the stable ABC specification from Section 11. In this framework the inter-ECU communication is clearly separated from the local computations done at each ECU during a slot. Note that we first reported on DISTR in [Kna08].

By instantiating DISTR in Section 17 with the processor configuration from Section 5.1, we obtain a gate-level specification of the communication mechanism. We prove the correctness of our implementation from Section 13 with respect to this instantiated framework. Note that in both cases, the processor part is the gate-level processor implementation from Section 5.1. However, the ABC implementation is abstracted to a stable ABC specification from Section 11.

Once the generic framework is justified at the gate level it is instantiated with the other local models introduced in Part I. We deal with the distributed ISA (DISA) model in Section 18, the distributed CVM (DCVM) model in Section 19, and finally with the distributed OLOS (DOLOS) model in Section 20. Using the corresponding local simulation theorems we prove the correctness of the distributed system on each layer at the end of the corresponding sections.

Finally, in Section 21, we define an automata-theoretic model, called communicating OLOS automata (COA), see [Kna08]. This model completely abstracts from the devices; communication is done directly via the local message buffers in OLOS. We prove the correctness of DOLOS with respect to the COA model.

Thus, by the end of Part II, we have established a pervasive correctness proof for the complete distributed system.

13 ABC Implementation

We consider a time-triggered scenario. As already mentioned in Section 11, time is divided into so-called rounds each consisting of ns slots. We uniquely identify slots by a tuple consisting of a round number $r \in \mathbb{N}$ and a slot number $s \in [0 : ns - 1]$. Predecessors $(r, s) - 1$ and successors $(r, s) + 1$ are computed modulo ns, see also Section 11:

$$(r, s) + 1 = \begin{cases} (r, s + 1) & s < ns - 1 \\ (r + 1, 0) & \text{otherwise} \end{cases}$$

The predecessor is defined analogously.

The ABC implementation is split in four main parts, as depicted in Figure 9:

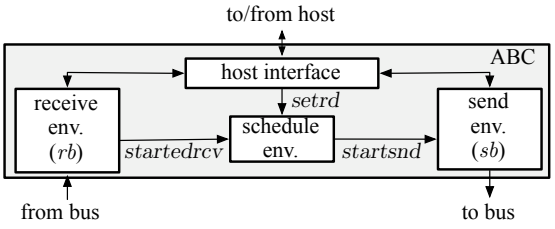

Figure 9: ABC Schematics

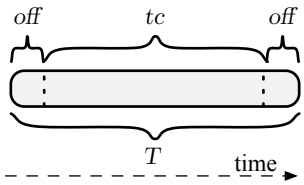

Figure 10: ABC Slots

- the host interface provides the connection to the host and contains configuration registers.
- the send environment performs the actual message broadcast and contains a send buffer.
- the receive environment takes care of the message reception and contains a receive buffer.
- the schedule environment is responsible for the clock-sychronization and the conformance to the schedule.

Note that we have implemented the ABC inspired by [Pau05]. Furthermore, we have extended the design in particular by a more sophisticated schedule environment.

Next we detail each part of the implementation starting with the host interface.

13.1 Host Interface

In this section we recall and extend the set of configuration parameters already mentioned in Section 11.

Unless synchronization is performed, slots are locally T hardware cycles long. A slot can be further subdivided into three parts (see Figure 10): an initial as well as a final offset (each *off* hardware cycles) and a transmission interval (tc hardware cycles). The length of the transmission interval is implicitly given by the slot-length and the offset.

Within each slot one fixed-length message msg is broadcast which consists of ml bytes, i.e. $msg \in \mathbb{B}^{8 \cdot ml}$.

A local schedule ft, implemented as a bit-vector, indicates if the ABC is the sender in a given slot. Intuitively, if the ABC is in slot s and $ft(s) = 1$ then the ABC broadcasts the message stored in the send buffer (see Section 13.2). Note that the ABC implementation is not aware of the round number. It simply operates according to the slot-based fixed schedule, that is repeated again and again.

The special parameter $iwait$ indicates the number of hardware cycles to be awaited before the ABC starts executing the schedule (see Section 13.4).

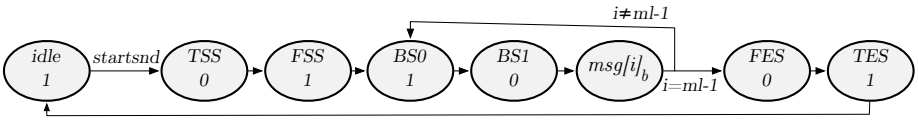

Figure 11: Automaton for the Message Encoding

All configuration parameters introduced so far need to be set by the host during an initialization phase using simple store word operations to device addresses (see Section 11). The host indicates that it has finished the initialization by invoking a $setrd$ command. It does so by writing the word $0^{31}1$ to the command port of the ABC.

13.2 Send Environment

The send environment starts broadcasting the message contained in the send buffer sb if the schedule environment raises the $startsnd$ signal (see Figure 9).

To define the message encoding we need to introduce some more notation first (see also Section 3). For natural numbers n and a bit y we denote by y^n the string in which y is replicated n times, e.g. $0^4 = 0000$. For strings $x[0 : k-1]$ consisting of k bits $x[i]$ we denote by $8 \cdot x$ the string obtained by repeating each bit eight times, i. e. $8 \cdot x = x[0]^8 \dots x[k-1]^8$.

The following protocol is used for transmission (see Figure 11). Given a message msg a frame $f(msg)$ is created by inserting falling edges between the message bytes and adding some bits at the start and at the end:

$$f(msg) = 0110 msg[0]_b 10 msg[1]_b 10 \dots 10 msg[ml-1]_b 01$$

Note that we refer to a complete message byte with number i by $msg[i]_b$.

In a frame $f(msg)$ we call the first 'zero' the transmission start sequence (TSS), the first 'one' the frame start sequence (FSS), the last 'zero' the frame end sequence (FES) and the last 'one' the transmission end sequence (TES). The two bits producing a falling edge before each byte are called the byte start sequence ($BS0, BS1$).

The sending ECU broadcasts $8 \cdot f(sb)$ over the bus. The send environment basically implements the automaton from Figure 11. The additional hardware to insert the protocol bits is implemented straight forward using signals from the send-automaton. The duplication of the single bits is achieved by clocking the send-automaton every 8 cycles only.

13.3 Receive Environment

The receive environment permanently listens on the bus. At an incoming message, indicated by a falling edge (the bus is high-active), it signals the start of a frame reception to the schedule environment via the $startedrcv$ signal. In addition it decodes the broadcast frame and writes the message into the receive buffer rb. Therefore it also implements an automaton similar to Figure 11 to keep track of the frame bits.

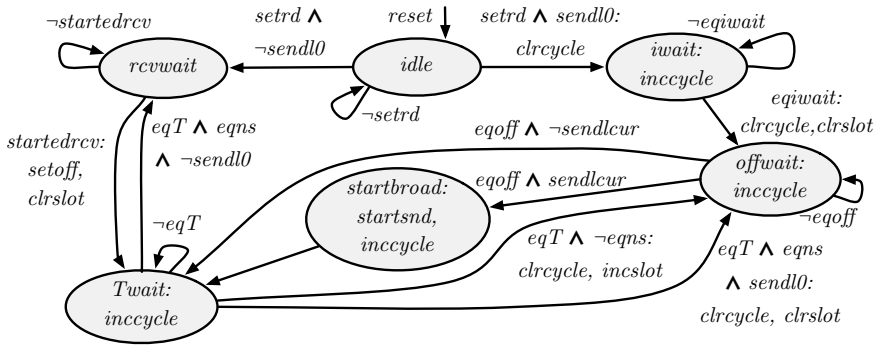

Figure 12: Schedule Automaton

The receive environment tries to sample the sequence of the 8 identical frame bits roughly in the middle. To do so, it uses a modulo-8 counter cnt that is synchronized regularly to stay roughly in the middle of the 8-bit interval. Further details are given in Section 14.4.

13.4 Schedule Environment

The correctness of the scheduling mechanism, which argues in particular about the implementation of our schedule environment, has been subject to formal verification in the master thesis of Peter Böhm [Böh07]. We supervised his work resulting in a joint paper [ABK08b]. Here we only sketch the main ideas behind the schedule environment. The interested reader may consult [Böh07] for further details regarding the latter.

The schedule environment maintains two counters: The cycle counter cy and the current slot counter csn. Both are periodically synchronized at the beginning of each round. All ECUs except the one broadcasting in slot 0 (we call the former *slaves* and the latter *master*) synchronize their counters according to the incoming transmission in slot 0. Hence, the $startedrcv$ signal from the receive environment is used to provide a synchronized time base (see below). Furthermore, the schedule environment initiates the message broadcast by raising the $startsnd$ signal.

The schedule environment implements the automaton from Figure 12. The automaton takes the following inputs (see Figure 9): The signal $setrd$ denotes the end of the configuration phase. The start of a message reception is indicated by the signal $startedrcv$. The signal $sendl0 = ft(0)$ indicates if the ECU is the sender in the first slot and thus the master. Three signals are used to categorize the cycle counter; $eqiwait$, $eqoff$ and eqT indicate if the initial $iwait$, off or T cycles have been reached. The signal $eqns$ indicates that the end of a round has been reached, i.e. that the slot counter equals $ns - 1$. Finally, $sendlcur$ indicates if the ABC is the sender in the current slot, i.e. $sendlcur = ft(csn)$.

The automaton has six states and is clocked each hardware cycle. Its functionality can be summarized as follows: If the $reset$ signal is raised (which is assumed to happen

only at power-up) the automaton is forced into the $idle$-state. If the host is done with the initialization and thus invoked $setrd$ we split cases on the $sendl0$ signal. If the ABC is the master, i.e. if $sendl0$ holds, the ABC waits first $iwait$ hardware cycles (in the $iwait$-state), then an additional off cycles (in the $offwait$-state) before it starts broadcasting the message (in the $startbroad$-state) and proceeds to the $Twait$-state.

If the ABC is a slave ($\neg sendl0$), it waits in the $rcvwait$-state for an incoming transmission (that is indicated by $startedrcv$). Then it clears its slot-counter ($clrslot$), sets its cycle counter to off cycles ($setoff$), and proceeds to the $Twait$-state. In the $Twait$-state all ABCs await the end of a slot indicated by eqT. At the end of a slot we split cases if the round is finished or not. If the round is not finished yet (indicated by $\neg eqns$), all ABCs proceed to the $offwait$-state. Then, after off many cycles, the sender in the current slot (indicated by $sendlcur$) proceeds to the $startbroad$-state, initiates the message broadcast, and proceeds to the $Twait$-state; all other ABCs skip the $startbroad$-state and proceed directly to the $Twait$-state. At the end of a round, the master ABC simply repeats the 'normal' sender cycle (from the $Twait$-state to the $offwait$-state and finally to the $Twait$-state again). All other ABCs proceed to the $rcvwait$-state to await an incoming transmission.

Once initialized, the master ABC follows the schedule without any synchronization. At the beginning of a new round it waits off many cycles and initiates the broadcast.

The clock synchronization on the slave ABCs is done in the $rcvwait$-state. In this state the cycle counter is not altered but simply stalls in its last value. At an incoming transmission (from the master) the slaves clear their slot-counter and set their cycle counter to off, i.e. the number of hardware cycles at which the master initiated the broadcast. After this all ABCs are (relatively) synchronized to the masters clock.

13.5 Hardware Construction

The number of ECUs that are connected to the bus is denoted by ne. Thus an ECU number is given by:

$$u \in En = [0 : ne - 1]$$

We use subscript ECU numbers to refer to single ECUs.

In this section we argue only about the ABC implementation. Later on, we will integrate the ABC implementation into the processor.

We denote the ABC configuration of ECU_u by abc_u. If the index u of the ECU does not matter, we drop it. The essential components of the ABC configuration are:

- two single bit-registers, one for sending and one for receiving. Both are directly connected to the bus. We denote them $abc.S$ and $abc.R$.
- a second receiver register, denoted by $abc.\hat{R}$, to deal with metastability (see Section 14.2).
- double buffers $abc.sb(par)$ and $abc.rb(par)$, where $par \in \{0, 1\}$, implement the send and receive buffers. Each pair of the double buffer is capable of storing one complete message.

- the registers of a non-trivial timer implementing a current slot counter $abc.csn$ and a cycle counter $abc.cy$.
- a 3-bit counter $abc.cnt$ that is used as a modulo-8 counter to perform a low level bit synchronization (see Section 14.4).
- all automata are implemented straight forward as a transition system on a unary coded bit-vector (see [MP00]). In the following we will argue only about the schedule automaton in detail. To do so, we use $abc.state$ to code the current state of this automaton (see Figure 12).
- configuration registers as detailed in Section 13.1.

The cycle counter $abc.cy$ counts the cycles of a slot. Unless the timer is synchronized, slots have locally T cycles. The slot counter $abc.csn$ counts the slot index s of the current slot (r, s).

The lowest bit of the slot counter keeps track of the parity of the current slot and is called the hardware parity signal, i.e. $par(abc) = abc.csn \bmod 2$.

The bus side of the interface sees the buffer $abc.sb(par(abc))$ as well as the buffer $abc.rb(par(abc))$. We will use the two shorthands $abc.sbd = abc.sb(par(abc))$ and $abc.rbd = abc.rb(par(abc))$ where 'd' refers to 'device'. Messages are always transmitted between these two copies of the buffers.

The host, on the other hand, writes to buffer $abc.sb(\neg par(abc))$ and reads from buffer $abc.rb(\neg par(abc))$. We use the shorthands $abc.sbp = abc.sb(\neg par(abc))$ and $abc.rbp = abc.rb(\neg par(abc))$ where 'p' refers to 'processor'. This does not work at boundaries of rounds unless the number of slots ns is even.

The configuration registers are written immediately after reset or power-up. They contain in particular the locally relevant portions of the scheduling function.

To simplify arguments regarding the schedule, we define a global scheduling function $send$. Given a slot number s it returns the number of the ECU sending in this slot. Let ft_u denote the local schedule of ECU_u, then $send(s) = u \Leftrightarrow ft_u(s) = 1$. Note that this definition implicitly requires the existence of a unique sender for each slot. Otherwise correct message broadcast becomes impossible due to bus contention.

Thus if ECU_u is (locally) in a slot with slot index s and $send(s) = u$ then ECU_u will transmit the content of the send buffer $abc.sbd$ via the bus during some transmission interval. A serial interface that is not actively transmitting during slot (r, s) puts by construction the idle value (the bit 1) on the bus.

14 ABC Verification

To argue about asynchronous distributed communication systems we have to formalize the behavior of the digital circuits connected to the analog bus, as reported in [BBG+05, KP07b]. Using the formalization of digital clocks from Section 14.1 we introduce a hardware model for continuous time in Section 14.2. In Section 14.3, we detail two important Lemmata regarding asynchronous communication, i.e. a bounded clock drift

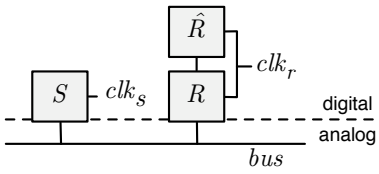

Figure 13: Serial Interface

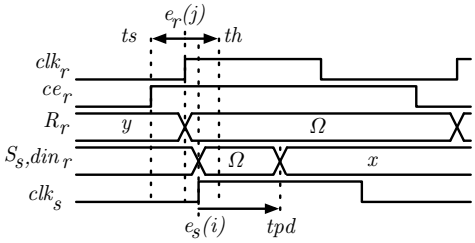

Figure 14: Clock Edges

and a correct sampling interval. In the remainder of this section we sketch the message transmission correctness, detail the scheduling correctness [ABK08b] and combine both into one correctness statement.

If we can guarantee that during the transmission interval *all* ECUs are locally in slot (r, s), then transmission is successful according to the message transmission correctness. The clock synchronization algorithm together with an appropriate choice of the transmission interval will ensure exactly that.

14.1 Clocks

The hardware of each ECU is clocked by an oscillator having a nominal clock period of τ_{ref}. The individual clock period τ_u of an ECU_u is allowed to deviate by at most $\delta = 0.15\%$ from τ_{ref}, i.e. $\forall u. \mid \tau_u - \tau_{ref} \mid \leq \tau_{ref} \cdot \delta$. Note that this bound can be easily achieved by current technology. Thus the relative deviation of two individual clock periods compared to a third clock period is bounded by $\mid \tau_u - \tau_v \mid \leq \tau_w \cdot \Delta$ where $\Delta = 2\delta/(1-\delta)$.

Given some clock-start offset $c_u < \tau_u$ the date of the clock edge $e_u(i)$ that starts cycle i on ECU_u is defined by:

$$e_u(i) = c_u + i \cdot \tau_u$$

Thus cycle i of ECU_u denotes the time interval $(e_u(i) : e_u(i+1)]$. Note that each τ_u is assumed to be constant. We do not deal with jitter here.

In our scenario all ECUs are connected to a bus. The sending ECUs broadcasts data which is sampled by all other ECUs. Due to clock drift it is not guaranteed that the timing parameter, i.e. set-up time and hold time, of the sampling registers are obeyed. This problem is tackled by a non-trivial message protocol as defined in Section 14.4. To argue formally we first introduce a continuous time model for bits being broadcast.

14.2 Hardware Model with Continuous Time

The problems solved by serial interfaces can by their very nature not be treated in a standard digital hardware model with a single digital clock clk. Nevertheless, we can describe the ABC of each ECU_u in such a model.

In order to argue about the sender register $abc.S$ of a sending ECU that is transmitting data via the bus to a receiver register $abc.R$ of a receiving ECU, as depicted in Figure 13, we have to extend the digital model.

For the registers connected to the bus –and only for those– we extend the hardware model such that we can deal with the concepts of propagation delay (tpd), set-up time (ts), hold time (th), and metastability of registers from hardware data sheets. In the extended model used near the bus we therefore consider time to be a real valued variable t.

Next we define in the continuous time model the output of the sender register $abc_u.S$ during cycle i of ECU_u. The content of the sender register $abc_u.S$ at time t is denoted by $S_u(t)$.

In the digital hardware model we denote the value of some register r during cycle i by $abc_u^i.r$ which equals the value at the clock edge $e_u(i+1)$.

If in cycle $i-1$ the digital clock enable $Sce(abc_u^{i-1})$ signal was off, we see during the whole cycle the old digital value $abc_u^{i-1}.S$ of the register. If the register was clocked ($Sce(abc_u^{i-1}) = 1$) and the propagation delay tpd has passed, we see the new digital value of the register, which is equal to the digital input $Sdin(abc_u^{i-1})$ during the previous cycle. Otherwise we cannot predict what we see, which we denote by Ω:

$$S_u(t) = \begin{cases} abc_u^{i-1}.S & : Sce(abc_u^{i-1}) = 0 \wedge t \in (e_u(i) : e_u(i+1)] \\ Sdin(abc_u^{i-1}) & : Sce(abc_u^{i-1}) = 1 \wedge t \in [e_u(i) + tpd : e_u(i+1)] \\ \Omega & : \text{otherwise} \end{cases}$$

The bus is an open collector bus modeled for all t by:

$$bus(t) = \bigwedge_u S_u(t)$$

Now consider a receiver register $abc_u.R$ on ECU_u whose clock enable is continuously turned on; thus the register always samples from the bus. In order to define the new digital value $abc_u^j.R$ of register R during cycle j on ECU_u we have to consider the value of the bus in the time interval $(e_u(j) - ts, e_u(j) + th)$, i.e. from the clock edge minus the set-up time until the clock edge plus the hold time. Only if the bus has a constant digital value $x \in \{0,1\}$ during that time interval, the register can sample this value, i.e. $\exists x \in \{0,1\}.\, \forall t \in (e_u(j) - ts, e_u(j) + th).\, bus(t) = x \Rightarrow abc_u^j.R = x$. Otherwise we do not know what is sampled and define $abc_u^j.R = \Omega$. Thus we have to argue how to deal with unknown values Ω as input to digital hardware.

The output of register $abc_u.R$ is used only as input to a second register $abc_u.\hat{R}$ whose clock enable is always turned on, too. If Ω is clocked into $abc_u.\hat{R}$ we assume that $abc_u.\hat{R}$ has an unknown but digital value, i.e. $abc_u^j.R = \Omega \Rightarrow abc_u^{j+1}.\hat{R} \in \{0,1\}$.

In real systems the counterpart of register $\hat{R}$ exists. The probability that R becomes metastable for an entire cycle *and* that this causes $\hat{R}$ to become metastable too is for practical purposes zero.

14.3 Continuous Time Lemmata for the Bus

Consider a pair of ECUs, where ECU_s is the sender and ECU_r is a receiver in a given slot. Let i be a sender *cycle* such that $Sce(abc_s^{i-1}) = 1$, i.e. the output of S is not

guaranteed to stay constant at time $e_s(i)$. This change can only affect the value of register R of ECU_r in cycle j if it occurs before the sampling edge $e_r(j)$ plus the hold time th, i.e. $e_s(i) < e_r(j) + th$. The first cycle that is possibly being affected is denoted by:

$$cy_{r,s}(i) = \min\{j \mid e_s(i) < e_r(j) + th\}$$

The receiver puts, by construction, the value 1 on the bus and keeps its Sce signal off (hence $bus(t) = S_s(t)$ for all t under consideration). Therefore we drop the indices r and s and simply write $cy(i)$ instead of $cy_{r,s}(i)$.

We will show that despite clock drift, the receiver misses at most one of the 8 broadcast copies of a frame bit. Figure 14 depicts a situation in which one bit is missed because the sender signal is not propagated, yet. If the two intervals $[e_s(i), e_s(i)+tpd]$ and $[e_r(j)-ts, e_r(j)+th]$ intersect, then the receiver samples some undefined signal Ω.

The potential miss in cycle i is modeled by the sampling drift $\sigma_{r,s}(i)$ which is defined by:

$$\sigma_{r,s}(i) = \begin{cases} 0 & : e_r(cy(i)) \geq e_s(i) + tpd + ts \\ 1 & : \text{otherwise} \end{cases}$$

Analogous to $cy(i)$ we write $\sigma(i)$ if sender and receiver are obvious from the context.

There are two essential lemmata whose proofs base on the continuous time model. The first lemma considers a situation, where the clock enable Sce of the sender ECU is activated in cycle $i-1$ but not in the following seven cycles. In the digital model we then have $abc_s^i.S = \ldots = abc_s^{i+7}.S$ and in the continuous time model we observe $x = bus(t) = S_s(t) = abc_s^i.S$ for all $t \in [e_s(i) + tpd : e_s(i+8)]$. Then x is correctly sampled in seven consecutive cycles.

Lemma 4 (Correct Sampling Interval) *Let the clock enable signal of the sender register S be turned on in cycle $i-1$, i.e. $Sce(abc_s^{i-1}) = 1$. Furthermore, let the same signal be turned off in the next seven cycles, i.e. $Sce(abc_s^j) = 0$ for $j \in \{i, \ldots, i+6\}$, then:*

$$abc_r^{cy(i)+\sigma(i)+k}.R = abc_s^i.S \quad \text{for } k \in \{0, \ldots, 6\}$$

The second lemma simply bounds the clock drift. It essentially states that within 300 cycles clocks cannot drift by more than one cycle; this is shown using $\delta \leq 0.15\%$.

Lemma 5 (Bounded Clock Drift) *Given a cycle $m \in \{1, \ldots, 300\}$ it holds that:*

$$cy(i) + m - 1 \leq cy(i+m) \leq cy(i) + m + 1$$

Detailed proofs of similar lemmata can be found in [BBG+05, KP07b], formal proofs are reported in [Sch06] as well as in the master thesis of Peter Böhm [Böh07] which we have supervised.

14.4 Message Broadcast Correctness

According to Lemma 4, for each bit of the frame that is replicated 8 times, a sequence of seven bits is sampled correctly. If the receiver succeeds to sample that sequence

roughly in the middle, he wins. For this purpose the receiver uses the modulo-8 counter $abc.cnt$ that is trying to keep track of which of the eight identical copies of a frame bit is currently being transmitted. When the counter value equals four a $strobe$ signal is raised. For frame decoding the bus is sampled with the $strobe$ signal. The automaton trying to keep track of the protocol is also clocked with this $strobe$ signal.

Clocks are drifting, hence the hardware has to perform a low level synchronization. The counter $abc.cnt$ is reset by a $sync$ signal in two situations: At the beginning of a transmission or at an expected falling edge during the byte start sequence. Let v denote the bit sampled from the bus. Abbreviating signals $s(abc_r^i)$ with s^i we write:

$$sync^i = (idle^i \vee BS0^i) \wedge (\neg v^i \wedge v^{i-1})$$

We do not go into further details here. The interested reader may consult [KP07b].

The crucial part of the message transmission correctness is a lemma which argues about three statements using induction on the receiver cycles:

Lemma 6 (Single Message Reception)

1. *the state of the automaton keeps track of the transmitted frame bit.*
2. *the $sync$ signal is activated at the corresponding falling edges of the sampled bit between $BS0$ and $BS1$.*
3. *sequences of identical bits are sampled roughly in the middle.*

We sketch the proof: Statement 1 is clearly true in the $idle$ state. From statement 1 follows that the automaton expects the falling edges of the sampled signal exactly when the sender generates them. Thus the counter is well synchronized after these falling edges. This shows statement 2. Immediately after synchronization the receiver samples roughly in the middle. There is a synchronization about every 80 sender cycles due to the message encoding (at the edge between $BS0$ and $BS1$). By Lemma 5 and because $80 < 300$, the sampling point can wander by at most two bits between activations of the $sync$ signal. This is good enough to stay within the correctly sampled seven copies. This shows statement 3. If transmitted frame bits are correctly sampled, then the automaton keeps track of them. This shows statement 1.

Note that the 8-bit replication for each encoded message bit is prescribed by the FlexRay standard. We do not consider any transmission errors therefore less replication would actually be needed in our case. Due to drifting clocks and metastability the reception could be off by 2 bits. Thus a 5 bit replication, i.e. 2 bits in each direction, would suffice to prove the theorem.

Next we state the correctness of the broadcasting mechanism of a single message.

Theorem 7 (Message Broadcast Correctness) *Let the broadcast message be started in the sender-cycle i. The value of the device visible send buffer of the current sender is copied to all device visible receive buffers within the transmission interval of tc sender cycles, where $tc = 41 + 80 \cdot ml$:*

$$abc_u^{cy(i+tc)}.rbd = abc_{send(s)}^i.sbd$$

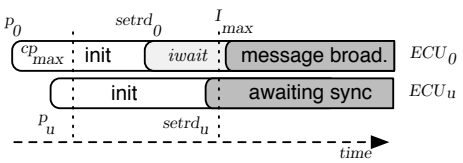

Figure 15: Startup

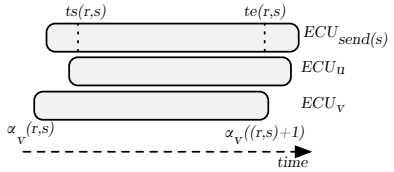

Figure 16: Schedule Correctness

The proof of this theorem is based on Lemma 6. We obtain the number of transmission cycles as follows: the replicated frame length ml is given by $32 + 80 \cdot ml$ cycles. Note that protocol bits are inserted and that we are using a eight-bit replication (as described in Section 13.2). The additional 9 hardware cycles result from local sender and receiver delays.

14.5 Startup

We assume w.l.o.g. that the ECU with number 0 is the master, i.e. $send(0) = 0$. Let p_u be the point in time when ECU_u is switched on (see Figure 15). We assume that at most cp_{max} hardware cycles have passed on the master ECU, since it was switched on, until all other ECUS are also switched on, i.e. $\forall u. \mid p_u - p_0 \mid \leq cp_{max} \cdot \tau_0$.

After initialization, all hosts invoke a $setrd$ command. Recall that this initialization phase is needed to set the configuration registers of the ABC. Once the $setrd$ command was invoked, the master ECU waits $iwait$ hardware cycles before it starts executing the schedule.

We assume that there exists a point in time denoted by I_{max} at which all slaves have invoked the $setrd$ command and await the first incoming message, however the master has not started the transmission, yet. This assumption can be easily discharged by deriving an upper bound for the duration of the initialization phase, say i_{max} hardware cycles in terms of the master ECU, and choosing $iwait$ to be $cp_{max} + i_{max}$. The upper bound can be obtained by industrial worst case execution time (WCET) analyzers [FMWA99] for the concrete processor and software.

14.6 Scheduling Correctness

We introduce some notation first. The start times of slot (r, s) on an ECU_u is denoted by $\alpha_u(r, s)$. Initially, for all u we define $\alpha_u(0, 0) = I_{max}$. To define the slot start times greater than slot $(0, 0)$ we need a predicate $schedexec$ that indicates if the schedule automaton is in one of three *executing* states, i.e. $schedexec(abc_u^i) = abc_u^i.state \in \{offwait, Twait, startbroad\}$. Let i be the smallest cycle such that $e_u(i)$ is greater than $\alpha_u((r, s) - 1)$ and the schedule is executed, the cycle counter has the value 0 and the slot counter the value s:

$$schedexec(abc_u^i) \wedge abc_u^i.cy = 0 \wedge abc_u^i.csn = s$$

Moreover let j be the smallest cycle such that $e_u(j)$ is greater than $\alpha_u((r,s)-1)$ and $abc_u^j.state = rcvwait$:

$$\alpha_u(r,s) = \begin{cases} e_u(i) & u = 0 \vee s > 0 \\ e_u(j) & otherwise \end{cases}$$

Similarly to the slot start times we define the slot end-times $\omega_u(r,s)$ of the slot (r,s) to be $\omega_u(r,s) = \alpha_u((r,s)+1) - \tau_u$.

Using the definition of a clock edge we obtain the hardware cycle corresponding to $\alpha_u(r,s)$. It is denoted by $\alpha t_u(r,s)$. The hardware-cycle corresponding to $\omega_u(r,s)$ is denoted by $\omega t_u(r,s) = \alpha t_u((r,s)+1) - 1$.

The local timers are synchronized each round. This point in time when the synchronization is performed in round r is called the synchronization end time of r. On ECU_u it is denoted by $\beta_u(r)$. It is the smallest time point $e_u(i)$ such that the schedule is executed, the cycle counter has the value *off* and the slot counter the value 0, i.e. $schedexec_u^i \wedge cycle_u^i = \mathit{off} \wedge slot_u^i = 0$.

A gate-level timing analysis of the synchronization process in the complete hardware design shows that:

Lemma 7 (Synchronization Times Relation) *For all u and y the synchronization of ECU_u to the master is completed within the adjustment time of $ad = 10$ hardware cycles:*

$$\begin{array}{rcl} \beta_0(r) & = & \alpha_0(r,0) + \mathit{off} \cdot \tau_0 \\ \beta_u(r) & < & \beta_0(r) + ad \cdot \tau_y \end{array}$$

The proof of this lemma is split in two parts. First we need to calculate the time it takes after the $startsnd$ signal was raised until the sender puts the first bit of a transmission on the bus. In the implementation this takes 2 cycles.

Second we need to bound the delay on the receiver side until the $startedrcv$ signal is raised after an incoming transmission plus an additional cycle to update the counters and the schedule control automaton. This bound is 7 cycles in the implementation. All in all this sums up to 9 cycles which is strictly smaller than 10.

Next we relate the start times of slots on the same ECU.

Lemma 8 (Slot Start Times Relation) *The start of slot (r,s) on the master ECU depends only on the progress of the local counter, i.e. $\alpha_0(r,s) = \alpha_0((r,s)-1) + T \cdot \tau_0$. The start of slot (r,s) on all other ECUs is given by:*

$$\alpha_u(r,s) = \begin{cases} \beta_u(r) + (T - \mathit{off}) \cdot \tau_u & s = 1 \\ \alpha_u((r,s)-1) + T \cdot \tau_u & s \neq 1 \end{cases}$$

Proof by induction on r and s. For $s > 1$ no synchronization takes place and the start of new slots is only determined by the progress of the local timer.

Lemma 9 (Difference between Slot Starts) *Let $\mathit{off} = (ad + \lceil ns \cdot T \cdot \Delta \rceil)$. The difference between the slot starts of the same slot on different ECUs is bounded by off:*

$$\alpha_u(r,s) - \alpha_v(r,s) < \tau_w \cdot \mathit{off}$$

We have $\mathit{off} = ad + \mathit{drift}$ with an adjustment time of $ad = 10$ and a maximal round drift of $\mathit{drift} = \lceil ns \cdot T \cdot \Delta \rceil$ cycles:

$$\begin{array}{rcll} \alpha_u(r,s) - \alpha_v(r,s) & < & ad \cdot \tau_w + (s \cdot T) \cdot (\tau_u - \tau_v) & \text{(Lemma 7 + Lemma 8)} \\ & \leq & ad \cdot \tau_w + \lceil ns \cdot T \cdot \Delta \rceil \cdot \tau_w) & \text{(round drift + Def. } \Delta\text{)} \\ & = & \tau_w \cdot (ad + \lceil ns \cdot T \cdot \Delta \rceil) & \\ & = & \tau_w \cdot \mathit{off} & \end{array}$$

The transmission is started in slot (r,s) by $ECU_{send(s)}$ when the local cycle count equals off. Thus the transmission start time is given by:

$$ts(r,s) = \alpha_{send(s)}(r,s) + \mathit{off} \cdot \tau_{send(s)}$$

According to Theorem 7 the transmission is finished within tc hardware cycles of the sending ECU:

$$te(r,s) = ts(r,s) + tc \cdot \tau_{send(s)} = \alpha_{send(s)}(r,s) + (\mathit{off} + tc) \cdot \tau_{send(s)}$$

The schedule is correct if the transmission interval $[ts(r,s), te(r,s)]$ is contained in the time interval, when all ECUs are in slot (r,s), as depicted in Figure 16 on Page 78.

Theorem 8 (Schedule Correctness) *All ECUs must be in slot (r,s) before the transmission starts. Furthermore, the transmission must be finished before any ECU thinks it is in the next slot:*

$$\begin{array}{rcl} \alpha_u(r,s) & < & ts(r,s) \\ te(r,s) & < & \alpha_u((r,s)+1) \end{array}$$

We prove the first statement:

$$\begin{array}{rcll} \alpha_u(r,s) & < & \alpha_x(r,s) + \tau_x \cdot \mathit{off} & \text{(Lemma 9 for some } x\text{)} \\ & = & ts(r,s) & \text{(for } x = send(s)\text{)} \end{array}$$

To prove the second statement we case split on the current slot. For $s > 0$:

$$\begin{array}{rcll} te(r,s) & = & \alpha_{send(s)}(r,s) + (T - \mathit{off}) \cdot \tau_{send(s)} & (T = 2 \cdot \mathit{off} + tc) \\ & = & \alpha_{send(s)}((r,s)+1) - \mathit{off} \cdot \tau_{send(s)} & \text{(Lemma 8 for } s > 0\text{)} \\ & < & \alpha_u((r,s)+1) & \text{(Lemma 9)} \end{array}$$

For $s = 0$ (the master is sending):

$$\begin{array}{rcll} te(r,0) & = & \alpha_{send(0)}(r,0) + (T - \mathit{off}) \cdot \tau_{send(0)} & (T = 2 \cdot \mathit{off} + tc) \\ & = & \alpha_{send(0)}(r,1) - \mathit{off} \cdot \tau_{send(0)} & \text{(Lemma 8 for the master)} \\ & < & \alpha_u(r,1) & \text{(Lemma 9)} \end{array}$$

Now the transmission correctness can be stated in the purely digital hardware model:

Theorem 9 (Transmission Correctness) *Consider slot (r,s). The value of the device visible send buffer of the sending ECU at the start of slot (r,s) is copied to all device visible receive buffers of all ECU_u by the end of the slot:*

$$abc_u^{\omega t_u(r,s)}.rbd = abc_{send(s)}^{\alpha t_{send(s)}(r,s)}.sbd$$

To prove this theorem we have to combined Theorem 7 and Theorem 8. According to Theorem 7 the actual broadcast is correctly performed if the transmission interval $[ts(r, s), te(r, s)]$ is big enough. The latter is proven by Theorem 8.

A formalization of the low-level bit-transmission correctness is described in [Sch06] based on [Pau05]. The formalized theorem does not argue about the broadcast of a message from a send buffer to a receive buffer but only about some intermediate registers involved in the transmission.

Formal proofs regarding the scheduling correctness have been reported in [Böh07, ABK08b]. In the formal Isabelle/HOL proofs an inductive invariant has been shown for all slots s. The arguments from [KP07b] have been extended to fit the new schedule environment implementation described in Section 13.4.

The startup mechanism described in Section 14.5 is novel. It has not been integrated into any formal Isabelle/HOL proofs, yet.

14.7 Extensions Towards Fault-Tolerance

In [ABK08a] we have shown how to extend the ABC implementation to become fault-tolerant. This requires an enhancement of the scheduling automaton, the message encoding, and the start-up as well as the normal scheduling algorithm. Furthermore, we have added a possibility for reintegration of faulty ABCs into the regular schedule.

Instead of relying on a distinguished master doing the synchronization, at the beginning of each round, every ABC listens on the bus for a specific time. If no transmission is detected by that time, the ABC declares itself as the master and starts broadcasting. The time span must be long enough to ensure that all previous ABCs (according to the schedule) had the possibility to start a transmission themselves. After synchronization and until the end of the current round the schedule of all ABCs is insensitive to bus activity and therefore also to failing ABCs.

For details regard fault-tolerance the interested reader may consult [ABK08a]. This paper is meant to be a proof of concept rather than a sequel in our pervasive verification approach. In the remainder of this thesis we will stick to the non-fault-tolerant case.

15 Distributed Implementation (DIMPL)

In this section we argue about the implementation configuration of the complete distributed system which we denote by $dimpl$. It consists of a mapping $dimpl.lm$ from ECU numbers to local implementation configurations. For a given ECU number en the mapping returns the corresponding local implementation configuration $impl$ that has been defined in Section 6.4:

$$dimpl.lm(en) = (impl.p, impl.d)$$

Note that each local device configuration $impl.d$ is an ABC implementation configuration abc as defined in the previous sections.

We denote the *first* implementation configuration in slot (r, s) by $dimpl(r, s)$. To define the latter we need to know the number of hardware cycles in slot (r, s) which

we denote by $nt_u(r,s)$. This number is:

$$nt_u(r,s) = \alpha t_u((r,s)+1) - \alpha t_u(r,s)$$

Given some initial configuration $dimpl^0$, the first configuration of the next slot $dimpl((r,s)+1)$ is defined by applying the implementation transition function δ_{impl} from Section 6.4 to each local implementation configuration as often as there are hardware cycles in slot (r,s), i.e. $nt_u(r,s)$ many times.

Note that the local transition function δ_{impl} takes external input. As the ABCs are connected by a bus, this external input is given by the value of the bus at the respective hardware cycle (see Section 14.2).

15.1 DIMPL Properties

Next, we state some properties of the combined processor and ABC implementation:

Lemma 10 (Current Slot Number Incrementation) *In the first configuration of each slot (r,s) the current slot number off each ECU_u is incremented modulo the number of slots in one round (compare Section 11 and 13):*

$$\begin{aligned} dimpl(0,0).lm(u).d.csn &= 0 \\ dimpl((r,s)+1).lm(u).d.csn &= s+1 \end{aligned}$$

Lemma 11 (Interrupt Generation) *In the first configuration of each slot (r,s) an interrupt is generated on all ECU_u:*

$$dimpl(r,s).lm(u).d.int = 1$$

Both lemmata are proven by an analysis of the scheduling automaton from Section 13.4 using the slot start times relation (Lemma 8). The interrupt is cleared by software via a write to the command port of the device as sketched in Section 6.4.

Lemma 12 (Message Broadcast) *The content of the send buffer of the current sender is correctly broadcast to the receive buffer of all ECUs and becomes visible to the processor at the start of the next slot:*

$$dimpl((r,s)+1).lm(u).d.rbp = dimpl(r,s).lm(send(s)).d.sbd$$

Note that the parity of the ABC configuration does not change *during* the slot. Using Theorem 9 we conclude that the message is correctly broadcast until the end of the slot. At the slot boundaries the current slot number is incremented resulting in a flip of the parity (as defined in Section 13.5). Thus the broadcast message becomes visible to the processors.

During a slot our ABC implementation is stable according to the definition in Section 6.4:

Lemma 13 (Stable ABC Implementation) *The ABC implementation is stable during a slot but not at the slot boundaries:*

$$\forall r,s.\ \forall t \in [\alpha t(r,s) : \alpha t((r,s)+1) - 1].\ stable(impl^t)$$

This lemma is proven by induction over all slots (r, s) and all hardware cycles t using the fact that the ABC implementation updates the processor visible buffers based on the double buffering mechanism as described in Section 13.5.

16 Distributed Framework (DISTR)

In this section we develop a distributed framework (DISTR) which defines the communication behavior between the ECUs. Thus it represents a specification of the communication mechanism provided by the ABCs.

Note that the DISTR framework is parameterized by a local model such that it can be used throughout an existing model stack for a single ECU to obtain the corresponding distributed models. The use of DISTR will greatly simplify the argumentation regarding the distributed models as huge parts of the framework are literally the same for each instantiation.

In the remainder of this thesis the DISTR framework is instantiated with each layer of the (local) model stack that has been developed in Part I.

16.1 DISTR Configuration

To define the DISTR configuration we first introduce a parameterized local state ls. Such a local state ls is a tuple consisting of the parameterized model itself and a stable ABC specification configuration from Section 11. We refer to the parameterized model with $ls.p$ for 'processor' and to the ABC specification configuration with $ls.d$ for 'device'.

A DISTR configuration $distr$ is parameterized by a local model. It consists of:

- a mapping $distr.lm$ from ECU numbers to local states. For a given ECU number en the mapping returns the corresponding local state ls:

$$distr.lm(en) = (ls.p, ls.d)$$

- the message that has been broadcast in the previous slot, denoted by $distr.oldm$.
- the message to be broadcast in the current slot, denoted by $distr.newm$.
- the current slot number, denoted by $distr.csn$.
- the current round number, denoted by $distr.crn$.

16.2 DISTR Transition Function

The DISTR transition function δ_{distr} specifies the execution of all ECUs for a complete slot. It is split in three phases:

(1) in the global receive phase δ_{dr} the receive buffers of all ECUs are updated with the message that has been broadcast in the previous slot,

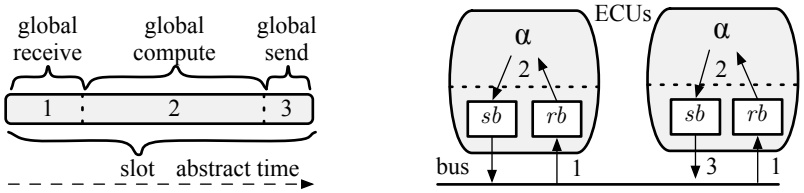

Figure 17: DISTR: Slot Partitioning and Data-Flow

(2) in the global compute phase δ_{dc} all ECUs do a computation for one complete slot,

(3) in the global send phase δ_{ds} the current sender broadcasts a message, the current slot numbers are incremented and interrupts are generated.

Figure 17 illustrates the slot partitioning as well as the data-flow within the phases. Note that we use the indices (1), (2) and (3) throughout the rest of this thesis to refer to the corresponding global DISTR phases.

The transition function of the distributed framework δ_{distr} applies first δ_{dr}, then δ_{dc} and finally δ_{ds} on a given configuration:

$$\delta_{distr}(distr) = \delta_{ds}(\delta_{dc}(\delta_{dr}(distr)))$$

Next we define the transition function of each phase. For a better reading ECU numbers like en are not quantified explicitly.

16.2.1 Global DISTR Receive Phase (1)

The transition function δ_{dr} of the global receive phase updates the receive buffer of all ECUs with the message that has been broadcast in the previous slot:

$$distr'.lm(en).d.rb = distr.oldm$$

16.2.2 Global DISTR Compute Phase (2)

In the global compute phase the ECUs are executed for the whole slot. The semantics of this so-called slot-step must be specified for each instantiation of the framework specifically. Thus for each instantiation, we have to define a function δ_{lcomp} doing a slot-step, i.e. $ls' = \delta_{lcomp}(ls)$. The transition function δ_{dc} of the global compute phase simply applies δ_{lcomp} to all ECUs:

$$distr'.lm(en) = \delta_{lcomp}(distr.lm(en))$$

Note that δ_{lcomp} requires stable devices as introduced in Part I.

16.2.3 Global DISTR Send Phase (3)

We require the local bus schedules be valid, i.e. for each slot number s there exists a unique sender:

$$\exists! en.\ distr.lm(en).d.ft(s) = 1$$

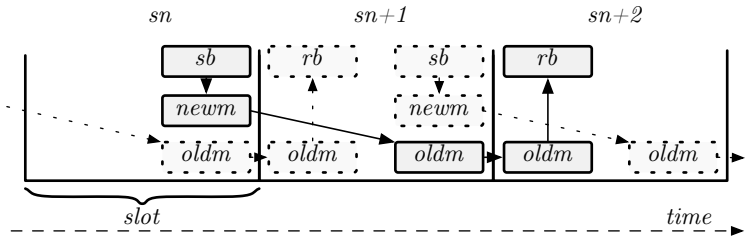

Figure 18: Communication Delay in DISTR

The ECU number of the sender in slot s is denoted by $send(s)$ as introduced in Section 13.5.

The transition function δ_{ds} of the global send phase increments the current slot number and the current round number. Furthermore, an interrupt is raised, the sender in the next slot broadcasts the message stored in its send buffer, and the broadcast of the message in the current slot is modeled by updating the old-message component:

$$\begin{aligned}(distr'.crn, distr'.csn) &= (distr.crn, distr.csn) + 1\\ distr'.lm(en).d.int &= 1\\ distr'.newm &= distr.lm(send(distr.csn + 1)).d.sb\\ distr'.oldm &= distr.newm\end{aligned}$$

Note that the interrupt flag of the ABC is cleared by software during the global compute phase. This flag is needed in the upper layers to specify the start of local computations.

Furthermore, note that the inter-ECU communication is delayed by one slot as done (see Figure 18). At the end of the global send phase in slot sn the message is taken from the send-buffer of the sender, it is broadcast in slot $sn + 1$ and can be accessed after the global receive phase in slot $sn + 2$ by all other ECUs. The actual broadcast is modeled by the update of the old-message component.

16.2.4 DISTR Notation

The first DISTR configuration in slot (r, s) is denoted by $distr(r, s)$. Given some initial DISTR configuration $distr^0$ it is defined by:

$$\begin{aligned}distr(0, 0) &= distr^0\\ distr((r, s) + 1) &= \delta_{distr}(distr(r, s))\end{aligned}$$

Formalization. Note that we have formalized DISTR in the theorem prover Isabelle/-HOL resulting in about 1000 lines of code.

17 Distributed Hardware (DH)

To simplify the argumentation, we separate the correctness of the ABC implementation from the correctness of the processor implementation.

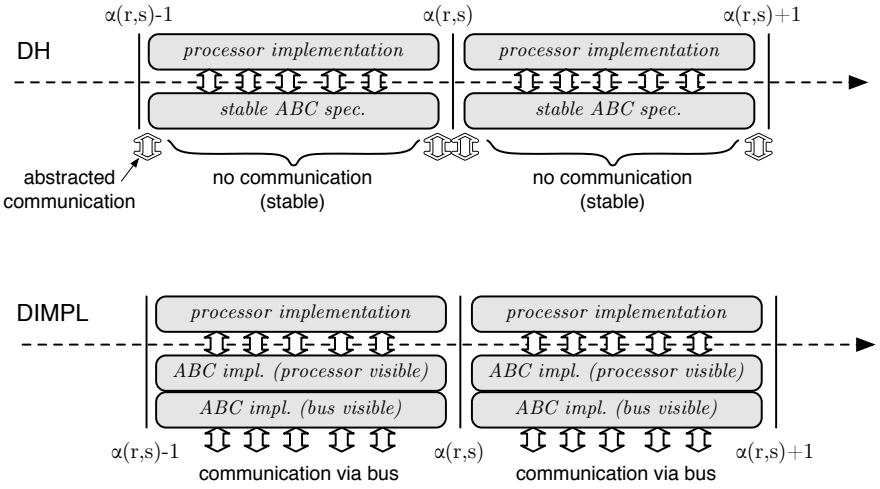

Figure 19: Abstraction of ABC from DIMPL to DH

In this section, we introduce a distributed hardware (DH) model. We argue about the correctness of the ABC implementation with respect to that model. The correctness of the processor implementation is dealt with later on in Section 18.

As the main result of this section we will establish a simulation theorem between DIMPL and DH.

In DIMPL the communication and the computation are performed in parallel (compare Figure 19). The processor only sees a subset of the ABC implementation through the I/O ports. This subset is changed from time to time due to the double buffering mechanism in the ABC implementation. Thus, in general the I/O ports in the implementation are not stable.

The DH model represents the processors point of view on the system. In DH the ABC is abstracted. We do only argue about the processor visible subset of the ABC. Communication and computation are serialized and clearly separated into three phases. The inter-ECU communication is performed at the start and at the end of the slot abstracting from the behavior of the implementation. During the slot, the processor can access a stable ABC specification configuration as defined in Section 11. Note that the stability is explicitly coded in the transition function.

This is possible due to the fact that the processor visible state of the ABC is altered at slot boundaries only. During the slot the processor visible state of the ABC is only changed by the processor itself.

Once this simulation theorem is established we can use the stable local models introduced in Part I to argue about the local computation on each ECU.

17.1 DH Configuration

We obtain a DH configuration dh by instantiating the DISTR configuration with a hardware configuration h from Section 5.1. Thus for each ECU number en the local mapping $dh.lm$ returns the corresponding hardware configuration hd with an integrated stable ABC specification:

$$dh.lm(en) = hd$$

17.2 DH Transition Function

The transition function executing the distributed hardware for a complete slot is denoted by δ_{dh}. It is obtained using the definition of δ_{distr}. The only function that is left undefined is the local slot-step function δ_{lcomp}, compare Section 16. To define the latter, we use the hardware transition function δ_{hd} from Section 6.5. Note that this function defines a transition using the stable device semantics.

Recall that the number of hardware cycles in slot (r, s) is given by $nt(r, s)$. The transition function δ_{hd} is applied $nt(r, s)$ many times on each configuration hd:

$$\delta_{lcomp}(hd) = \delta_{hd}^{nt(r,s)}(hd)$$

Given some initial DH configuration dh^0 we denote the first DH configuration in slot (r, s) by $dh(r, s)$. It is defined by:

$$\begin{aligned} dh(0,0) &= dh^0 \\ dh((r,s)+1) &= \delta_{dh}(dh(r,s)) \end{aligned}$$

Note that δ_{dh} first applies δ_{dr}, then δ_{dc} and finally δ_{ds} where δ_{dc} is defined in a model specific way (using δ_{lcomp} as defined above).

17.3 DIMPL Correctness

Next we prove the correctness of the implementation with respect to the DH model. The simulation relation $dhsim(s, dimpl, dh)$ between some slot number s, some DIMPL configuration $dimpl$, and some DH configuration dh states that:

1. the processor configurations are the same:

$$dimpl.lm(u).p = dh.lm(u).p$$

2. the new-bus component contains the message to be broadcast, i.e. the message stored in the device visible send buffer of the sender implementation:

$$dimpl.lm(send(s)).d.sbd = dh.newm$$

3. the old bus component contains the message that has been broadcast in the previous slot and that is being stored in the processor visible receive buffer of *all* implementations:

$$dimpl.lm(u).d.rbp = dh.oldm$$

4. the current slot numbers of *all* implementations are equal to the current slot number in DH which itself is equal to s:

$$dimpl.lm(u).d.csn = dh.csn = s$$

5. the interrupt flags are equal:

$$dimpl.lm(u).d.int = dh.lm(u).d.int$$

6. the local scheduling functions are equal:

$$dimpl.lm(u).d.ft = dh.lm(u).d.ft$$

We require that the processor accesses the send buffer only by store word operations and the receive buffer only by load word operations. In addition, we require, that the sender always updates the complete send buffer. Thus the message to be broadcast is well defined.

Furthermore, we require that the device is not accessed in the last cycle of each slot. Note that this assumption cannot be justified at this level since we do not deal with any concrete code here. Instead we will propagate the requirements up in the model stack resulting in concrete requirements for the code to be run in the system.

Theorem 10 (DIMPL Correctness) *Let the simulation relation between DIMPL and DH hold initially, i.e.* $dhsim(0, dimpl(0,0), dh(0,0))$. *Furthermore, let an interrupt be generated on all ECUs in both models initially. Then, the simulation relation holds for all slots* (r, s)*:*

$$dhsim(s, dimpl(r,s), dh(r,s))$$

The theorem is proven by induction over all slots (r, s). We argue along the global DH phases.

In the global receive phase (1) only the content of the receive buffer is changed. It is updated with the message that has been broadcast in the previous slot:

$$\begin{array}{lcll} dimpl(r,s).lm(u).d.rbp & = & dh(r,s).oldm & \text{(Ind. Hyp.)} \\ & = & (\delta_{dr}(dh(r,s))).lm(u).d.rb & \text{(Def. } \delta_{dr}\text{)} \end{array}$$

Note that the processor configuration is not altered in the global receive phase.

In the global compute phase (2) the DH configuration is executed for $nt(r,s)$ hardware cycles, so is the DIMPL configuration. Note that during this computation all ABC implementations $dimpl(r,s).lm(u).d$ are stable according to Lemma 13. Thus, according to Lemma 1, we can use the semantics for stable devices to argue about both DH and DIMPL for all the $nt(r,s)$ cycles.

The following argumentation holds inductively for all $nt(r,s)$ cycles: The processor configurations in both models are still equal after the execution of read operations since only the receive buffers are accessed (according to the assumptions for read operations) which are equal as shown above. The processor configurations remain equal

after the execution of a write operation. As the processor configurations are equal the processor sending in the next slot updates its send buffer in the same way.

We summarize this fact as follows: The ABC implementation and the ABC specification get the same inputs and thus produces the same outputs. We conclude:

$$dimpl((r,s)+1).lm(u).p = (\delta_{dc}(\delta_{dr}(dh(r,s)))).lm(u).p$$

At slot boundaries the current slot number is incremented and thus the parity flips. According to the assumptions, the device is not being accessed in this cycle, thus we can conclude that the send buffers to be broadcast in the next slot are the same after phase (2):

$$dimpl((r,s)+1).lm(send(s+1)).d.sbd = (\delta_{dc}(\delta_{dr}(dh(r,s)))).lm(send(s+1)).d.sb$$

Hence statement 1 of the simulation relation holds in slot $(r,s)+1$ as the processor part is not altered in the global send-phase:

$$dimpl((r,s)+1).lm(u).p = dh((r,s)+1).lm(u).p \quad (\text{Def. } \delta_{ds})$$

Furthermore, after the execution of phase (3) statement 2 holds in slot $(r,s)+1$:

$$dimpl((r,s)+1).lm(send(s+1)).d.sbd = dh((r,s)+1).newm \quad (\text{Def. } \delta_{ds})$$

According to Lemma 12 the send buffer is correctly broadcast in the implementation. Thus, the message stored in the receive buffers equals the message stored in the send buffer of the sender which, according to statement 2, equals the content of the new-bus component in the DH configuration. Hence, after the execution of the global send-phase (3) the old-bus component contains the message that has been broadcast in the current slot, i.e. the new-bus component:

$$\begin{array}{lcll} dimpl((r,s)+1).lm(u).d.rbp & = & dimpl(r,s).lm(send(s)).d.sbd & (\text{Lemma 12}) \\ & = & dh(r,s).newm & (\text{Ind. Hyp. }) \\ & = & dh((r,s)+1).oldm & (\text{Def. } \delta_{ds}) \end{array}$$

Thus statement 3 holds in slot $(r,s)+1$.

The current slot number is incremented in the implementation configuration according to Lemma 10 and in the DH model according to the definition of δ_{ds}:

$$dimpl((r,s)+1).lm(u).d.csn = dh((r,s)+1).csn \quad (\text{Lemma 10 and Def. } \delta_{ds})$$

Thus statement 4 holds in slot $(r,s)+1$.

The interrupt flag is set in the implementation according to Lemma 11 and in the DH model according to the definition of δ_{ds}:

$$dimpl((r,s)+1).lm(u).d.int = dh((r,s)+1).lm(u).d.int \quad (\text{Lemma 11 and Def. } \delta_{ds})$$

Thus statement 5 holds in slot $(r,s)+1$.

Statement 6 is invariant after initialization, as the configuration registers are not allowed to be written at all.

Formalization. Note that we have worked on an initial version of the simulation relation $dhsim$ in the theorem prover Isabelle/HOL resulting in about 1500 lines of code.

18 Distributed ISA (DISA)

In the DH model each ECU is executed from one slot boundary to the next. At these boundaries the ABCs generate interrupts that indicate the start of a slot to the processor.

Note that the DH model defines the behavior of the ABCs strictly from the point of view of the devices themselves, i.e. the start of the slot directly corresponds to the point in time when the interrupt is generated by a device. However, we aim at proving the correctness of the software running on the processors where the interrupt sampling is possibly delayed. To bridge this gap we introduce the distributed ISA (DISA) model.

To argue about the DISA model we need to go into details regarding the interrupt sampling as well as the worst case execution time (WCET) of programs to be run on the processor.

18.1 DISA Configuration

The DISA configuration $disa$ is obtained by instantiating the distributed framework from Section 16.1 with an ISA configuration from Section 4. Thus for each ECU number en the local mapping $disa.lm$ returns the corresponding ISA configuration $isad$ with an integrated ABC:

$$disa.lm(en) = isad$$

18.2 DISA Transition Function

We consider only programs of the form:

$$\{0 :\ \mathrm{P}\ ;\ a : jump\ a;\ a + 4 : noop\}$$

The program does the useful work in portion P and then waits in an idle loop for a timer interrupt. The idle loop starts at address a. Note that we have a byte addressable memory and that in an ISA with delayed branch the idle loop has two instructions.

P initially has to clear and then to unmask the timer interrupt, which is masked when P is started (see Section 4.5).

The transition function executing a DISA configuration for a complete slot is denoted by δ_{disa}. It is obtained using the definition of δ_{distr}. The only function left undefined is δ_{lcomp}.

Using the δ_{isad} semantics we execute the local ISA configurations until an instruction is fetched from address a, i.e. until the idle loop is entered. The runtime of such a computation is defined by:

$$T_{isa}(isad) = \min\{t \mid \delta^t_{isad}(isad).p.dpc = a\}$$

The result of this computation is:

$$res_{isa}(isad) = \delta_{isad}^{T_{isa}(isad)+1}(isad)$$

Thus δ_{lcomp} is set to the result of the computation:

$$\delta_{lcomp}(isad) = res_{isa}(isad)$$

Given some initial DISA configuration $disa^0$ we denote the first DISA configuration in slot (r, s) by $disa(r, s)$. It is defined by:

$$\begin{aligned} disa(0,0) &= disa^0 \\ disa((r,s)+1) &= \delta_{disa}(disa(r,s)) \end{aligned}$$

18.3 DISA Correctness

The simulation relation $disasim(dh, disa)$ between some DH and some DISA configuration states that:

1. the local simulation relation $isadsim$ from Section 6.7 holds for all local models:

$$isadsim(dh.lm(u), disa.lm(u))$$

2. the old-bus component, the new-bus component, and the current slot- and round number are equal:

$$\begin{aligned} dh.oldm &= disa.oldm \\ dh.newm &= disa.newm \\ dh.csn &= disa.csn \\ dh.crn &= disa.crn \end{aligned}$$

Furthermore, we obtain a weakened (local) simulation relation $isadsim'(hd, isad)$ based on the simulation relation $isadsim(hd, isad)$ by dropping the requirement that the PC and the delayed PC match. We denote the weakened global simulation relation that is defined using $isadsim'(hd, isad)$ by $disasim'(dh, disa)$.

We require that the processor accesses the send buffer only by store word operations and the receive buffer only by load word operations. In addition, the sender always updates the complete send buffer.

Furthermore, we require that the ISA computation terminates, i.e. that the bound for the ISA runtime $T_{isa}(isad)$ exists.

Theorem 11 (Distributed Processor Correctness) *Let the simulation relation between DH and DISA hold initially, i.e.* $disasim(dh(0,0), disa(0,0))$. *Furthermore, let the idle loop be executed and let an interrupt be generated on all ECUs in* $dh(0,0)$ *and* $disa(0,0)$ *respectively. Then the weakened simulation relation holds for all slots* (r, s)*:*

$$disasim'(dh(r,s), disa(r,s))$$

The proof is done by induction on the slot (r, s). It is split along the global phases.

The argumentation in the global receive and in the global send phase is straight forward due to the definition of the DISTR framework. However, to argue about the global compute phase we need to go into details.

In the global receive phase we apply the same function to both configurations. Only the device configuration is changed, thus the simulation relation still holds:

$$disasim'(\delta_{dr}(dh(r,s)), \delta_{dr}(disa(r,s)))$$

The argumentation in global compute phase is purely local. We abbreviate the local configurations at the start of the global compute phase by:

$$\begin{aligned} hd_u(r,s) &= \delta_{dr}(dh(r,s)).lm(u) \\ isad_u(r,s) &= \delta_{dr}(disa(r,s)).lm(u) \end{aligned}$$

Furthermore, we abbreviate the local configurations at the end of the global compute phase by:

$$\begin{aligned} hd'_u(r,s) &= \delta_{dc}(\delta_{dr}(dh(r,s))).lm(u) \\ isad'_u(r,s) &= \delta_{dc}(\delta_{dr}(disa(r,s))).lm(u) \end{aligned}$$

Since the arguments are the same for each ECU we drop the index and simply write $hd(r,s)$, $hd'(r,s)$, $isad(r,s)$, and $isad'(r,s)$.

Before going into further details we first sketch the main idea behind the rest of the proof. We will argue that the following invariant holds: In both, the hardware and the ISA configuration, the idle loop is executed and the interrupt flag is raised in the first configuration of each slot. As the processor configuration and the interrupt flag are not altered during the global receive phase, the invariant holds for the start of the global compute phase, too.

Due to the fact, that the idle loop is being executed (only the PC and the delayed PC are altered), the weakened simulation relation holds until the interrupt is being sampled. Then, the pipeline is drained and the PC and the delayed PC are forced to 0 and 4 in both configurations. Thus the stronger simulation relation holds and we can instantiate the processor correctness (Theorem 2) for each step until the idle loop is entered again. We require that the code being executed does indeed enter the idle loop before the start of the next slot. Once being in idle loop again, only the PC and the delayed PC are altered and the weakened simulation relation holds until the end of the slot.

18.3.1 DISA Interrupt Sampling

Slot boundaries are indicated to the processor by means of interrupts. As we have already argued in Section 5.5, interrupts are sampled in the write-back stage of the processor implementation in case of non-memory operations. Furthermore, interrupts are not sampled each hardware cycle but only if the write-back state is $full$.

Therefore, the point in time when the device generates an interrupt might differ from the point in time, when the processor actually samples the interrupt. The time that elapses in between depends on the content of the processor pipeline.

To argue about this time period we have to assume some kind of liveness for the processor. For a given hardware configuration hd there exists the smallest number of hardware cycles o, called liveness offset, such that after o hardware cycles the next instruction is written back:

$$o(hd) = \min\{t \mid WB(\delta^t_{hd}(hd).p)\}$$

We assume that the liveness offset o is bounded by some constant C:

$$o(hd) < C$$

The slot start from the point of view of the processor is denoted by $\alpha p(r, s)$. Using the liveness offset it is defined by:

$$\alpha p(r, s) = \alpha t(r, s) + o(hd(r, s))$$

We consider the computation starting at the configuration $hd(r, s)$ and ending at configuration $hd'(r, s)$, i.e. we argue about the cycles t where:

$$\alpha t(r, s) \leq t < \alpha t((r, s) + 1)$$

We already know that there are at least $T - \mathit{off}$ hardware cycles between two consecutive slot starts from the point of view of the device (see Section 14.6):

$$\alpha t((r, s) + 1) - \alpha t(r, s) > T - \mathit{off}$$

Thus we conclude that there exist at least $T - \mathit{off} - C$ hardware cycles between the point in time when the interrupt is sampled by the processor and the point in time the next interrupt is generated by the device:

$$\alpha t((r, s) + 1) - \alpha p(r, s) > T - \mathit{off} - C$$

Note that the liveness offset must be much smaller than $T - \mathit{off}$ hardware cycles. Otherwise there is no time to do any useful computation in a slot.

We denote the processor configuration in which the interrupt is sampled by $hd^*(r, s)$:

$$hd^*(r, s) = \delta^{o(hd(r,s))}_{hd}(hd(r, s))$$

The following arguments are the same for each slot hence we will drop the slot index and simply write hd, hd^*, hd', $isad$, and $isad'$.

Let us first consider the computation starting at configuration hd until configuration hd^*. Due to the definition of the liveness offset, no instruction is written back in between. Thus the simulation relation $isadsim'$ holds until configuration hd^*:

$$isadsim'(hd^*, isad)$$

We use the following abbreviations $hd^+ = \delta_{hd}(hd^*)$ and $isad^+ = \delta_{isad}(isad)$. As the processor samples the interrupt in configuration hd^*, the pipeline is drained in the next cycle, i.e. $drained(hd^+)$. The $JISR$ forces the PC and the delayed PC to 4 and 0, see Section 4.6. Hence the stronger simulation relation $isadsim$ holds for hd^+ and $isad^+$:

$$isadsim(hd^+, isad^+)$$

Now we can instantiate the processor correctness (Theorem 2). However, to argue about the computation until the end of the slot, we first need to introduce some more notation.

18.3.2 Worst Case Execution Time

We consider some hardware configuration hd and some ISA configuration $isad$. Let $isadsim(hd, isad)$ hold. Then the ISA configuration $isad$ can be decoded from the hardware configuration hd using a function dec_{isa}:

$$isad = dec_{isa}(hd)$$

We formulate conditions using predicates on configurations. The configurations obeying the condition are denoted using sets. The set $H_{\mathrm{h}}(E)$ of all hardware configurations hd encoding an ISA configuration $isad \in E$ is defined by:

$$H_{\mathrm{h}}(E) = \{hd \mid dec_{isa}(hd) \in E\}$$

While the decoding is unique, the encoding is not. E.g. portions of the ISA memory can be kept in the caches in various ways.

Given some hardware configuration hd the hardware run time $T_{\mathrm{h}}(hd, a)$ until a fetch from address a is the smallest number of cycles t' such that an instruction, which has been fetched in an earlier cycle $t < t'$ from address a, is in the write back stage WB. Using the scheduling functions from Section 5.3 we define:

$$T_{\mathrm{h}}(hd, a) = \min\{t' \mid \exists t < t'.\ s(WB, t') = s(IF, t) \wedge hd^t.p.dpc = a\}$$

We consider a set of ISA configurations E obeying some given condition. The worst case execution time $WCET(E, a)$ until a fetch from address a is the largest hardware runtime $T_{\mathrm{h}}(hd, a)$ of a hardware configuration encoding a configuration in E:

$$WCET(E, a) = \max\{T_{\mathrm{h}}(hd, a) \mid hd \in H_{\mathrm{h}}(E)\}$$

In analogy we define the best case execution time $BCET(E, a)$ until a fetch from address a as the smallest runtime $F_{\mathrm{h}}(hd, a)$ of a hardware configuration encoding a configuration in E:

$$F_{\mathrm{h}}(hd, a) = \min\{t \mid s(IF, t) \wedge hd^t.p.dpc = a\}$$

$$BCET(E, a) = \min\{F_{\mathrm{h}}(hd, a) \mid hd \in H_{\mathrm{h}}(E)\}$$

Using the above definitions we define the worst case execution time $WCET(E, a, b)$ of a computation starting with the fetch from address a and ending with the write-back of the instruction stored at address b by:

$$WCET(E, a, b) = WCET(E, b) - BCET(E, a)$$

Such estimates can be obtained from industrial tools [Abs07] based on the concept of abstract interpretation [FMWA99]. AbsInts WCET analyzer does not calculate the "real" worst-case execution time $WCET(E, a)$ but an upper bound $WCET'(E, a) \geq WCET(E, a)$. Furthermore, it does not calculate the "real" best-case execution time $BCET(E, a)$, but an lower bound $BCET'(E, a) \leq BCET(E, a)$.

In the remainder of this thesis we require that the idle-loop is reached within the boundaries of one slot:

$$WCET(E, 0, a) \leq T - \mathit{off} - C$$

We look again at the ISA configuration $isad^+$ and the hardware configuration hd^+. Considering the computation for hardware run time many cycles, where $T_{\mathrm{h}}(hd^+, a) < T - \mathit{off} - C$, we conclude that the instruction in the write back stage (at the end of the computation) is the first instruction being fetched from a. The same holds for the ISA computation.

As argued above the processor pipeline is drained, i.e. $drained(hd^+)$. Furthermore, the simulation relation $isadsim$ holds:

$$isadsim(hd^+, isad^+)$$

Let $res_{\mathrm{h}}(hd^+) = \delta_{hd}^{T_{\mathrm{h}}(hd^+,a)}(hd^+)$ be the hardware configuration after T_{h} cycles and let $isad' = res_{isa}(isad^+) = \delta_{isad}^{T_{isa}(isad^+,a)}(isad^+)$ be the corresponding ISA configuration.

We apply the processor correctness (Theorem 2) for all $T_{\mathrm{h}}(hd^+, a)$ cycles using hd^+ and $isad^+$ as initial configurations and obtain:

$$isadsim(res_{\mathrm{h}}(hd^+), isad')$$

In this situation the pipeline is almost drained. It contains nothing but instructions from the idle loop. In particular, no memory operation is executed until the end of the slot.

Note that the weakened simulation relation $isadsim'$ does not argue about the PC or the delayed PC:

$$isadsim(res_{\mathrm{h}}(hd^+), isad') \Rightarrow isadsim'(res_{\mathrm{h}}(hd^+), isad')$$

Until the end of the global compute phase only instructions from the idle loop are executed which do not affect the $isadsim'$ relation, hence:

$$isadsim'(hd', isad')$$

Thus the simulation relation holds after the global compute phase:

$$disasim'(\delta_{dc}(\delta_{dr}(dh(r, s))), \delta_{dc}(\delta_{dr}(disa(r, s))))$$

In the global send phase the same function is applied to both configurations. Only the bus components, the configuration of the device, the slot number and the round number are changed. The simulation relation still holds after the global send phase:

$$disasim'(dh((r, s) + 1), disa((r, s) + 1))$$

Note that after the global compute phase and thus after the global send phase the idle loop is being executed on all ECUs. Furthermore, interrupts are generated on all ECUs during the global send phase. Thus, the initial conditions of Theorem 11 are satisfied for the slot $(r, s) + 1$ and are therefore invariant.

18.4 Pervasive Program and Processor Correctness

Using classical program correctness proofs at the ISA level we can show that: Given the precondition E holds for an ISA configuration, then, after the execution of the program P, the postcondition Q is satisfied. Written in a Hoare-triple style this reads $\{E\}P\{Q\}$. We require that the definition of E and Q does neither involve the PC nor the delayed PC, nor the configuration of the device.

We say that all ISA configurations satisfying the precondition E terminate within one slot if their WCET is smaller than $T - \mathit{off} - C$ cycles:

$$term_{isa}(E, a) = (WCET(E, a) < T - \mathit{off} - C)$$

For any slot (r, s) let $hd_u(r, s)$ denote the hardware configuration of the ECU with number u, i.e. $hd_u(r, s) = dh(r, s).lm(u)$. Furthermore, let $isad_u(r, s)$ denote the ISA configuration of the ECU with number u, i.e. $isad_u(r, s) = disa(r, s).lm(u)$.

We consider some DIMPL configuration $dimpl$ and some DH configuration dh. Let $dhsim(s, dimpl, dh)$ hold. Then a local HD configuration hd can be decoded from the local implementation $impl$ using a function dec_{h}:

$$hd = dec_{\mathrm{h}}(impl)$$

Note that dec_{h} is defined through the simulation relation $dhsim$.

Now we can state the pervasive program and processor correctness:

Theorem 12 (Pervasive Program and Processor Correctness) *Let all the simulation relations down to the implementation configuration hold at the start of slot (r, s), i.e. $dhsim(s, dimpl(r, s), dh(r, s))$ and $disasim(dh(r, s), disa(r, s))$. Let the pre condition and the post condition hold for an ISA configuration, i.e. $isad_u(r, s) \in E$ and $isad_u((r, s) + 1) \in Q$ and let the ISA configuration terminate within one slot, i.e. $term_{isa}(E, a) = 1$. Then, at the end of the slot, the decoded implementation obeys the postcondition Q:*

$$dec_{isa}(dec_{\mathrm{h}}(dimpl((r, s) + 1).lm(u))) \in Q$$

The theorem is proven by induction over all slots (r, s). Using the processor correctness (Theorem 11) we conclude that the simulation relation $disasim'$ holds at the start of slot $(r, s) + 1$:

$$disasim'(dh((r, s) + 1), disa((r, s) + 1))$$

Note that the idle loop only changes the PC and the delayed PC of the computation. As E and Q do not argue about the PC and the delayed PC we derive that the property Q holds during the execution of the idle loop:

$$dh_u((r, s) + 1) \in H_{\mathrm{h}}(Q)$$

Using the implementation correctness (Theorem 10) we derive:

$$dhsim(s + 1, dimpl((r, s) + 1), dh((r, s) + 1))$$

The simulation relation $dhsim$ requires that the processor component is equal in the DIMPL and the DH model. Thus the we get the desired theorem (see also [KP07a]).

19 Distributed CVM (DCVM)

Next we introduce the distributed CVM (DCVM) model based on the local CVM model from Section 10.

19.1 DCVM Configuration

A DCVM configuration $dcvm$ is given by instantiating the distributed framework with the processor part of a CVM configuration. Thus for each ECU number en the local mapping $dcvm.lm$ returns the corresponding CVM configuration cvm with an integrated ABC:

$$dcvm.lm(en) = cvm$$

19.2 DCVM Transition Function

We only consider programs of the form $P2; wait()$. Thus we will argue about the $wait$ primitive instead of the idle loop. Note that this structure fits the structure introduced in Section 18.2 as the $wait$ primitive is implemented using an idle-loop.

The transition function executing a DCVM configuration for a complete slot is denoted by δ_{dcvm}. It is obtained using the definition of δ_{distr} based on the following definition of δ_{lcomp}.

The local CVMs are executed using the δ_{cvm} semantics until the invocation of the wait primitive. The runtime T_{cvm} of such a computation is given by:

$$T_{cvm}(cvm) = \min\{t \mid (hd(\delta^t_{cvm}(cvm).p.co.pr) = (v = wait()))\}$$

We require that the CVM computation terminates, i.e. that the bound for the CVM runtime $T_{cvm}(cvm)$ exists. The result of this computation is then given by:

$$res_{cvm}(cvm) = \delta^{T_{cvm}(cvm)+1}_{cvm}(cvm)$$

The function δ_{lcomp} is set to the result of the CVM computation:

$$\delta_{lcomp}(cvm) = res_{cvm}(cvm)$$

Given some initial DCVM configuration $dcvm^0$ we denote the DCVM configuration at the start of slot (r, s) by $dcvm(r, s)$. It is defined by:

$$\begin{aligned} dcvm(0,0) &= dcvm^0 \\ dcvm((r,s)+1) &= \delta_{dcvm}(dcvm(r,s)) \end{aligned}$$

19.3 DCVM Correctness

We use the shorthand $cvmsim$ to denote the local CVM simulation relation as introduced in Section 10.6. The simulation relation between some DISA configuration $disa$ and some DCVM configuration $dcvm$ is denoted by $dcvmsim(disa, dcvm)$. It requires:

1. the local CVM simulation relation holds for all ECUs:

$$cvmsim(disa.lm(en), dcvm.lm(en))$$

2. the old-bus component, the new-bus component, and the current slot and round number are the same:

$$\begin{aligned} disa.oldm &= dcvm.oldm \\ disa.newm &= dcvm.newm \\ disa.csn &= dcvm.csn \\ disa.crn &= dcvm.crn \end{aligned}$$

Now we can state the DCVM correctness theorem:

Theorem 13 (DCVM Correctness) *Let the simulation relation between the DISA and the DCVM model hold initially, i.e.* $dcvmsim(disa(0,0), dcvm(0,0))$. *Let the idle loop be executed on all ECUs in* $disa(0,0)$ *and let the wait primitive be executed on all ECUs in* $dcvm(0,0)$. *Furthermore, let an interrupt be generated on all ECUs in both models respectively. Then the simulation relation holds for all slots* (r,s)*:*

$$dcvmsim(disa(r,s), dcvm(r,s))$$

The theorem is proven by induction over all slots (r,s).

In the global receive phase we apply the same function to both configurations. The simulation relation is not affected:

$$dcvmsim(\delta_{dr}(disa(r,s)), \delta_{dr}(dcvm(r,s)))$$

In the global send phase we also apply the same function to both configurations. If the simulation relation holds after the global compute phase the simulation relation still holds after the global send phase and thus at the start of the next slot:

$$\begin{gathered} dcvmsim(\delta_{dc}(\delta_{dr}(disa(r,s))), \delta_{dc}(\delta_{dr}(dcvm(r,s)))) \Rightarrow \\ dcvmsim(disa((r,s)+1), dcvm((r,s)+1)) \end{gathered}$$

Thus we have to show that the simulation relation indeed holds after the global compute phase.

In the global compute phase the argumentation is purely local. We abbreviate the local configurations at the start and at the end of the global compute phase by:

$$\begin{aligned} isad_u &= \delta_{dr}(disa(r,s))).lm(u) \\ cvm_u &= \delta_{dr}(dcvm(r,s))).lm(u) \\ isad'_u &= \delta_{dc}(\delta_{dr}(disa(r,s)))).lm(u) \\ cvm'_u &= \delta_{dc}(\delta_{dr}(dcvm(r,s)))).lm(u) \end{aligned}$$

The arguments are the same for each ECU, hence we drop the index and simply write $isad$, cvm, $isad'$, and cvm'. Note that according to the definition of $dcvmsim$ the local simulation relation $cvmsim(isad, cvm)$ holds after the global receive phase.

We apply the local CVM correctness (Theorem 5) for all $T_{cvm}(cvm) + 1$ steps of the CVM during the global compute phase and derive:

$$cvmsim(isad', cvm')$$

Details regarding this local simulation proof that go beyond the sketch from Section 10.6 will be given in [Tsy08, IdR08].

The global simulation relation $dcvmsim$ holds after the global compute phase as neither the bus components, nor the slot counter, nor the round counter are altered:

$$dcvmsim(\delta_{dc}(\delta_{dr}(disa(r,s))), \delta_{dc}(\delta_{dr}(dcvm(r,s))))$$

Note that after the global compute phase and thus after the global send phase the idle loop is being executed in all ECUs of the DISA model due to the definition of the result of an ISA computation from Section 18.2. Similarly, the $wait$ primitive is executed in all ECUs of the DCVM model according to the definition of the result of a CVM computation from Section 19.2. Furthermore, in the global send phase an interrupt is generated. Thus the initial conditions of the above theorem also hold for slot $(r,s)+1$ and are therefore invariant.

19.4 Pervasive DCVM Correctness

We consider some ISA configuration $isad$ and some CVM configuration cvm. Let the local simulation relation $cvmsim(isad, cvm)$ hold Then the CVM configuration cvm can be decoded from the ISA configuration $isad$ using a function dec_{cvm}:

$$cvm = dec_{cvm}(isad)$$

Note that this decoding function is based on the allocation function of the compiler (see Section 8).

Given some precondition E' holds for the CVM configuration, we assume that the post condition Q' is satisfied after the execution of P2 (see Section 19.2). The corresponding pre and post conditions for the ISA level are given by:

$$\begin{array}{rcl} H_{isa}(E') & = & \{isa \mid dec_{cvm}(isa) \in E'\} \\ H_{isa}(Q') & = & \{isa \mid dec_{cvm}(isa) \in Q'\} \end{array}$$

We require that in the compiled CVM code the idle-loop being part of the $wait$ primitive implementation starts at address a. A CVM configuration satisfying the precondition E' terminates within one slot if the WCET until the execution of the idle loop is smaller than $T - \mathit{off} - C$ cycles:

$$\begin{array}{rcl} term_{cvm}(E', a) & = & (WCET(H_{isa}(E'), 0, a) < T - \mathit{off} - C) \\ & = & term_{isa}(H_{isa}(E'), a) \end{array}$$

For any slot (r,s) let $cvm_u(r,s)$ and $isad_u(r,s)$ denote the CVM and the ISA configuration of the ECU with number u, i.e. $cvm_u(r,s) = disa(r,s).lm(u)$ and $isad_u(r,s) = disa(r,s).lm(u)$.

Theorem 14 (Pervasive DCVM Correctness) *Let all simulation relations down to the implementation hold at the start of slot (r, s), i.e. $dhsim(s, dimpl(r, s), dh(r, s))$ and $disasim(dh(r, s), disa(r, s))$ and $dcvmsim(disa(r, s), dcvm(r, s))$. Furthermore, let a pre and post condition hold for the CVM configuration, i.e. $cvm_u(r, s) \in E'$ and $cvm_u((r, s) + 1) \in Q'$ and let the CVM configuration terminate within one slot, i.e. $term_{cvm}(E, a) = 1$. Then, at the end of the slot, the decoded implementation obeys the postcondition Q':*

$$dec_{cvm}(dec_{isa}(dec_{\mathrm{h}}(dimpl((r, s) + 1).lm(u)))) \in Q'$$

The theorem is proven by induction over all slots (r, s). Using the CVM correctness (Theorem 13) we obtain that the simulation relation holds at the start of slot $(r, s) + 1$:

$$dcvmsim(disa((r, s) + 1), dcvm((r, s) + 1))$$

We conclude that the ISA configuration at the start of slot $(r, s) + 1$ obeys the post-condition, i.e. $isad_u((r, s) + 1) \in H_{isa}(Q')$.

The theorem is proven by applying the pervasive program and processor correctness (Theorem 12) using $H_{isa}(E')$ and $H_{isa}(Q')$ as pre and post-conditions.

20 Distributed OLOS (DOLOS)

In this section we introduce the distributed OLOS (DOLOS) model based the local OLOS model from Section 12.

20.1 DOLOS Configuration

The configuration of the distributed OLOS (DOLOS) model is obtained by instantiating the distributed framework with the processor part of an OLOS configuration. Note that for each ECU number en the local mapping $dolos.lm$ returns the corresponding OLOS configuration $olos$ with an integrated ABC:

$$dolos.lm(en) = olos$$

20.2 DOLOS Transition Function

The definition of the DOLOS local transition function for one slot δ_{dolos} is based on the function δ_{distr} which requires a definition of δ_{lcomp}.

Let pn be the application that is scheduled in the current slot on a given OLOS configuration $olos$:

$$pn = olos.p.st(olos.p.csn)$$

The OLOS runtime is the smallest number of steps until the application with number pn executes the $ttExFinished$ system call according to the δ_{olos} semantics (see Section 12.4):

$$T_{olos}(olos) = \min\{t \mid (hd(\delta^t_{olos}(olos).p.pm(pn).pr) = (v = ttExFinished()))\}$$

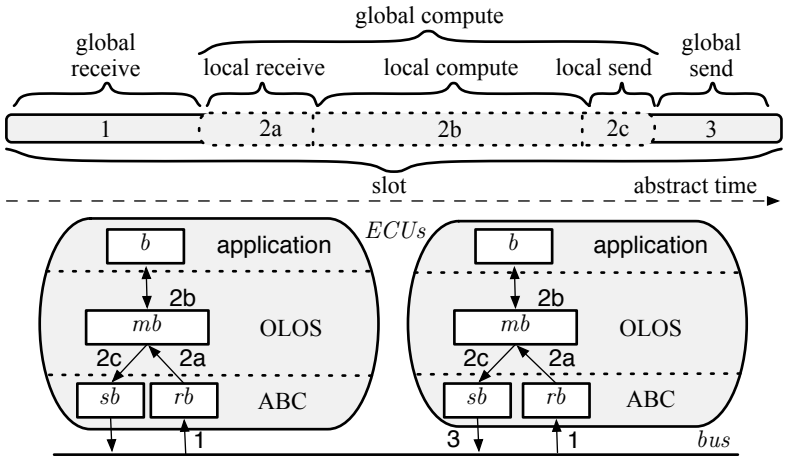

Figure 20: DOLOS: Slot Partitioning and Data-Flow

We require that the OLOS computation terminates, i.e. that the bound for the OLOS runtime $T_{olos}(olos)$ exists. The result of the computation is then given by:

$$res_{olos}(olos) = \delta_{olos}^{T_{olos}(olos)+1}(olos)$$

As mentioned in Section 12.4, the OLOS transition function is split in four phases. First, the local receive phase (a) is executed. Second, during the local compute phase (b) the message-buffer and the scheduled application is updated with the result of the local computation. Third, the local send phase (c) is executed. And finally, in the local idle phase (d) the configuration simply stalls.

In the following definitions we abstract from the local idle phase (d). Note that the local receive phase and the local send phase are executed in one step of the OLOS model while the number of steps during the local compute phase is process specific as defined above:

$$\delta_{lcomp}(olos) = \delta_{olos}(res_{olos}(\delta_{olos}(olos)))$$

The definitions of δ_{dolos} and δ_{lcomp} are illustrated in Figure 20.

Note that the interrupt, that is generated during the global send phase (3), as defined in Section 16.2.3, directly leads to the start of the local receive phase (compare Section 12).

Given some initial DOLOS configuration $dolos^0$ we denote the DOLOS configuration at the start of slot (r, s) by $dolos(r, s)$. It is defined by:

$$\begin{aligned} dolos(0,0) &= dolos^0 \\ dolos((r,s)+1) &= \delta_{dolos}(dolos(r,s)) \end{aligned}$$

Formalization. Note that we have formalized DOLOS in the theorem prover Isabelle/HOL using our DISTR formalization as well as our OLOS formalization. All in all the formalization of DOLOS sums up to about 3000 lines of code.

20.3 DOLOS Correctness

The simulation relation between some DCVM configuration and some DOLOS configuration is denoted by $dolossim(dcvm, dolos)$. In its definition we use the shorthand $olossim$ for the local OLOS simulation relation as introduced in Section 12.6. Thus $dolossim$ states that:

1. the local OLOS simulation relation from Section 12.6 holds for all ECUs:

$$olossim(dcvm.lm(en), dolos.lm(en))$$

2. the old-bus component, the new-bus component, and the current slot and round number are the same:

$$\begin{aligned} dcvm.oldm &= dolos.oldm \\ dcvm.newm &= dolos.newm \\ dcvm.csn &= dolos.csn \\ dcvm.crn &= dolos.crn \end{aligned}$$

Now we can state the DOLOS correctness theorem:

Theorem 15 (DOLOS Correctness) *Let the simulation relation between DCVM and DOLOS hold initially, i.e.* $dolossim(dcvm(0,0), dolos(0,0))$. *Let let all ECUs in* $dcvm(0,0)$ *execute the* $wait$ *primitive and let all ECUs in* $dolos(0,0)$ *be in the local receive phase. In addition, let an interrupt be generated on all ECUs in both configurations respectively. Then the simulation relation holds for all slots* (r, s)*:*

$$dolossim(dcvm(r,s), dolos(r,s))$$

The theorem is proven by induction over all slots (r, s).

In the global receive phase we apply the same function to both configurations. Only the device state is changed, thus the simulation relation still holds:

$$dolossim(\delta_{dr}(dcvm(r,s)), \delta_{dr}(dolos(r,s)))$$

As no communication between the ECUs is done during the global compute phase, the argumentation is purely local. We abbreviate the local configurations before and after the local compute phase by:

$$\begin{aligned} cvm_u &= \delta_{dr}(dcvm(r,s))).lm(u) \\ olos_u &= \delta_{dr}(dolos(r,s))).lm(u) \\ cvm'_u &= \delta_{dc}(\delta_{dr}(dcvm(r,s)))).lm(u) \\ olos'_u &= \delta_{dc}(\delta_{dr}(dolos(r,s)))).lm(u) \end{aligned}$$

The argumentation is the same for each ECU thus we drop the indices and simply write cvm, $olos$, cvm', and $olos'$. After the global receive phase the local simulation relation holds:

$$olossim(cvm, olos)$$

We can apply the OLOS correctness (Theorem 6) for all OLOS steps during the local compute phase and obtain:

$$olossim(cvm', olos')$$

Details regarding this local simulation proof that go beyond the sketch from Section 12.7 will be given in [Sch08].

The simulation relation $dolossim$ holds after the global compute phase:

$$dolossim(\delta_{dc}(\delta_{dr}(dcvm(r,s))), \delta_{dc}(\delta_{dr}(dolos(r,s))))$$

In the global send phase we again apply the same function to both configurations. The simulation relation still holds after the global send phase and thus at the start of the next slot:

$$dolossim(dcvm((r,s)+1), dolos((r,s)+1))$$

Note that during the global send phase an interrupt is generated on all ECUs of both models. After the global compute phase and thus after the global send phase all ECUs in the DOLOS model are again in the local receive phase due to the definition of the result of an OLOS computation from Section 20.2. Furthermore, the wait primitive is being executed on all ECUs of the DCVM model according to the definition of the result of a CVM computation from Section 19.2. Hence, the initial conditions of the above theorem also hold for slot $(r,s)+1$ and are therefore invariant.

20.4 Pervasive DOLOS Correctness

We consider only programs of the form R; A; S where the letter R denotes the code executed in the local receive phase, A the code executed by an application during the local compute phase, and S the code executed in the local send phase. Note that this structure fits the requirements from Section 19 as the local send-phase is implemented using the CVM $wait$ primitive which itself is implemented using an idle loop.

We consider some CVM configuration cvm and some OLOS configuration $olos$. Let $olossim(cvm, olos)$ hold. Then the OLOS configuration $olos$ can be decoded from the CVM configuration cvm using a function dec_{olos}:

$$olos = dec_{olos}(cvm)$$

Given some precondition E'' holds for the OLOS configuration, we assume that some post condition Q'' is satisfied after the execution of R, A, and S. The corresponding pre and post conditions for the CVM level are given by:

$$\begin{array}{lcl} H_{cvm}(E'') & = & \{cvm \mid dec_{olos}(cvm) \in E''\} \\ H_{cvm}(Q'') & = & \{cvm \mid dec_{olos}(cvm) \in Q''\} \end{array}$$

We require that the code of the receive phase starts at address 0, the code of the application at address b, the code of the send phase at address c and the idle loop at address a.

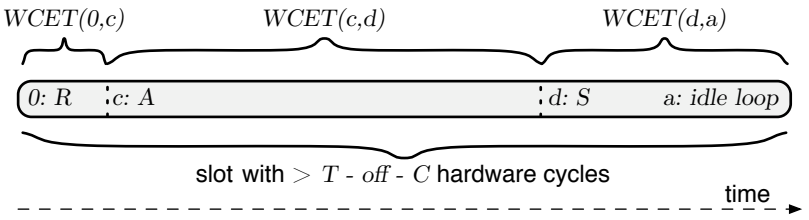

Figure 21: Worst Case Execution Time (WCET) Constraint

A OLOS configurations satisfying the precondition E'' terminates within one slot if the WCET until the idle loop is smaller than $T - \mathit{off} - C$ cycles:

$$WCET(H_{isa}(H_{cvm}(E'')), 0, a) < T - \mathit{off} - C$$

Note that at this abstraction level we know the concrete OLOS code. Thus we can reformulate the requirements for the WCET resulting in a constraint for the WCET of the applications. We require that the WCET of the application fits in one slot together with the WCET of R and S as illustrated in Figure 21:

$$term_{olos}(E'', c, d, a) = (WCET(H_{isa}(H_{cvm}(E'')), c, d) < (T - \mathit{off} - C - WCET(H_{isa}(H_{cvm}(E'')), 0, c) - WCET(H_{isa}(H_{cvm}(E'')), d, a)$$

The system calls can be analyzed independently from the applications. By annotating the function calls with the corresponding WCETs the results can be integrated into the analysis of the application code A.

Note that the above requirement implies the requirements for DCVM from Section 19.4:

$$term_{olos}(E'', c, d, a) \Rightarrow term_{cvm}(H_{cvm}(E''), a)$$

For any slot (r, s) let $olos_u(r, s)$ and $cvm_u(r, s)$ denote the OLOS and the CVM configuration of the ECU with number u, i.e. $olos_u(r, s) = dolos(r, s).lm(u)$ and $cvm_u(r, s) = disa(r, s).lm(u)$.

Theorem 16 (Pervasive DOLOS Correctness) *Let all simulation relations down to the implementation hold at the start of slot (r, s), i.e. $dhsim(s, dimpl(r, s), dh(r, s))$ and $disasim(dh(r, s), disa(r, s))$ and $dcvmsim(disa(r, s), dcvm(r, s))$ and last but not least $dolossim(dcvm(r, s), dolos(r, s))$. Furthermore, let pre and post condition hold for the OLOS configuration, i.e. $olos_u(r, s) \in E''$ and $olos_u((r, s) + 1) \in Q''$ and let the OLOS configuration terminate in one slot, i.e. $term_{olos}(E'', c, d, a) = 1$. Then, at the end of the slot, the decoded implementation obeys the postcondition Q'':*

$$dec_{olos}(dec_{cvm}(dec_{isa}(dec_{\mathrm{h}}(dimpl((r, s) + 1).lm(u))))) \in Q''$$

The theorem is proven by induction over all slots (r, s). Using the DOLOS correctness (Theorem 13) we obtain that:

$$dolossim(dcvm((r, s) + 1), dolos((r, s) + 1))$$

Thus the CVM configuration at the start of slot $(r, s) + 1$ obeys the post-condition, i.e. $cvm_u((r, s) + 1) \in H_{cvm}(Q'')$.

To prove the theorem we apply the CVM pervasive correctness (Theorem 14) using $H_{cvm}(E'')$ and $H_{cvm}(Q'')$ as the pre and post-condition.

21 Communicating OLOS Automata (COA)

The communicating OLOS automata (COA) model defines the behavior of the distributed system from the point of view of the applications themselves, as reported in [Kna08]. It completely abstracts from the devices; communication is done directly via the message buffers of the operating systems. Furthermore, the global knowledge like the slot-content function or the current slot number is made explicit. In the COA model we do not have to deal with those local details any more. Thus the argumentation, for instance about application correctness, is eased a lot.

Note that, it is possible to move from the COA model to an even more abstract automata theoretic model called AFTM. The latter specifies the semantics of the CASE tool AutoFocus [Aut08] that is being used in the automotive industry to specify applications to be run in the distributed system. We do not go into details here, a simulation proof between a very initial version of COA, and AFTM is reported in [BBG$^+$08].

21.1 COA Configuration

The local knowledge of an ECU is combined into a configuration denoted by ecu. This configuration contains:

- a process mapping $ecu.pm$,
- a message buffer $ecu.mb$,
- a bus schedule $ecu.ft$, and
- a scheduling table $ecu.st$.

All these components have been defined in Section 12 already.

A COA configuration coa consists of:

- a mapping from an ECU number en to the corresponding ECU configuration, i.e. $coa.em(en) = ecu$,
- a slot-content function $coa.sc$ as defined in Section 12,
- the current slot number $coa.csn$,
- the current round number $coa.crn$,
- the message that has been broadcast in the previous slot $coa.oldm$, and
- the message to be broadcast $coa.newm$.

Note that the two bus components $coa.oldm$ and $coa.newm$ have the same semantics as the corresponding components in the DISTR model.

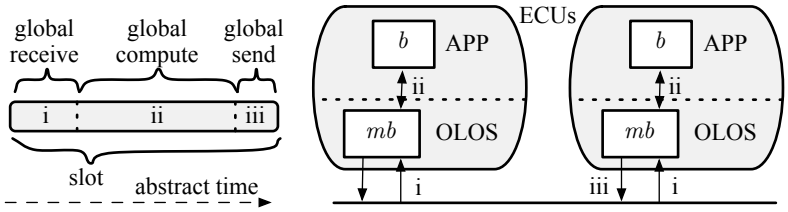

Figure 22: COA: Slot Partitioning and Data-Flow

21.2 COA Transition Function

The COA transition function δ_{coa} is split in 3 phases:

(i) in the receive phase δ_{cr} the message buffers of all ECUs are updated with the message that has been broadcast in the previous slot,

(ii) in the compute phase δ_{cc} all ECUs are executed locally for one slot,

(iii) in the send phase δ_{cs} the sender in the next slot updates the new bus component with the message to be broadcast. Furthermore, the current slot number is incremented.

Figure 22 illustrates the slot partitioning as well as the data-flow within the phases. Note that we use the indices (i), (ii) and (iii) throughout the rest of this thesis to refer to the corresponding COA phases.

The COA transition function δ_{coa} first applies δ_{cr} then δ_{cc} and finally δ_{cs} on a given configuration, i.e. $\delta_{coa}(coa) = \delta_{\text{cs}}(\delta_{\text{cc}}(\delta_{\text{cr}}(coa)))$.

21.2.1 Global COA Receive Phase (i)

Let the number of the message that has been broadcast in the previous slot be $mn = coa.sc(coa.csn - 1)$. The transition function δ_{cr} of the global receive phase updates all message buffers at position mn with the message that has been broadcast in the previous slot:

$$coa'.em(en).mb(mn) = coa.em(en).oldm$$

21.2.2 Global COA Compute Phase (ii)

In the definition of the global compute phase we reuse the concept of a local configuration from Section 12.3. The runtime $T_{lc}(lc)$ of a local configuration lc is the smallest number of steps until $ttExFinished$ is executed using the δ_{lc} semantics:

$$T_{lc}(lc) = \min\{t \mid hd(\delta^t_{lc}(lc).co.pr) = (v = ttExFinished())\}$$

We require that the computation terminates, i.e. that the bound for the runtime $T_{lc}(lc)$ exists. The result $res_{lc}(lc)$ of a local configuration lc is then obtained by applying the

function δ_{lc} runtime plus one many times (note that the $ttExFinished$ system call is executed, too):

$$res_{lc}(lc) = \delta_{lc}^{T_{lc}(lc)+1}(lc)$$

Let the process number of the application that is scheduled in the current slot on the ECU with number en be denoted by $pn(en) = coa.em(en).st(coa.csn)$. Furthermore, let co' and mb' denote the result of the local message buffer and the currently scheduled application:

$$(co'(en), mb'(en)) = (res_{lc}(coa.em(en).pm(pn(en))), coa.em(en).mb)$$

The transition function δ_{cc} of the global compute phase updates the message buffer and the scheduled application with the result of their computation:

$$\begin{aligned} coa'.em(en).pm(pn(en)) &= co'(en) \\ coa'.em(en).mb &= mb'(en) \end{aligned}$$

21.2.3 Global COA Send Phase (iii)

Like in DISTR we require the bus schedule in COA be valid, i.e in each slot s there exists a unique sender (see Section 16):

$$\exists! en(s).\ coa.em(en).ft(s) = 1$$

We denote the ECU number of the sender in slot s by $send(s)$.

Let the message number of the message to be broadcast in the next slot be given by $mn = coa.sc(coa.csn + 1)$. The transition function δ_{cs} of the global send phase increments the current slot number. Furthermore, the actual message broadcast is modeled by updating the old-bus component with the value of the new-bus component. The new-bus component itself is updated with the content of the send buffer of the ECU sending in the next slot:

$$\begin{aligned} (coa'.crn, coa'.csn) &= (coa.crn, coa.csn) + 1 \\ coa'.newm &= coa.em(send(coa.csn + 1)).mb(mn) \\ coa'.oldm &= coa.newm \end{aligned}$$

21.2.4 COA Notation

Given some initial COA configuration coa^0 we denote the COA configuration at the start of slot (r, s) by $coa(r, s)$. It is defined by:

$$\begin{aligned} coa(0, 0) &= coa^0 \\ coa((r, s) + 1) &= \delta_{coa}(coa(r, s)) \end{aligned}$$

21.3 COA Correctness

First we define a local simulation relation $ecusim(olos, ecu)$ between some OLOS configuration $olos$ and some ECU configuration ecu. It requires that the configuration

of the applications, the message buffers, the scheduling tables, and the bus schedules of both configurations are the same. Note that in $olos$ the bus schedule is stored in two places, in the operating system and in the ABC:

$$\begin{aligned} olos.p.pm &= ecu.pm \\ olos.p.mb &= ecu.mb \\ olos.p.st &= ecu.st \\ olos.p.ft &= ecu.ft \\ olos.d.ft &= ecu.ft \end{aligned}$$

The global simulation relation between some DOLOS configuration $dolos$ and some COA configuration coa is denoted by $coasim(dolos, coa)$. It requires that the local simulation relation holds for all ECUs. Furthermore, the two bus components, the slot-content function, and the current slot numbers must be equal. Note that in $dolos$ the current slot number is counted in two parts, in the operating system and in the ABC:

$$\begin{aligned} & ecusim(dolos.lm(en), coa.em(en)) \\ dolos.oldm &= coa.oldm \\ dolos.newm &= coa.newm \\ dolos.lm(en).p.sc &= coa.sc \\ dolos.csn &= coa.csn \\ dolos.crn &= coa.crn \\ dolos.lm(en).p.csn &= coa.csn \end{aligned}$$

Next we state the COA correctness theorem:

Theorem 17 (COA Correctness) *Let the simulation relation between DOLOS and COA hold initially, i.e.* $coasim(dolos^0, coa^0)$. *Let an interrupt be generated on all ECUs in* $dolos^0$ *and let all ECUs start in the local receive phase. Then the simulation relation holds for all slots* (r, s)*:*

$$coasim(dolos(r, s), coa(r, s))$$

The theorem is proven by induction over all slots (r, s). The proof is split along the global and local phases (compare Figure 20 and 22).

In the COA receive phase (i) all message buffers are updated with the message that has been broadcast in the previous slot. In the DOLOS model this message is first written in the receive buffers of the ABCs during the global receive phase (1). In the local receive phase (2a) the ABC driver updates the message buffer with the content of the receive buffer. Thus the simulation relation holds at this point in time.

During the global compute phase (ii) in COA as well as during the local compute phase (2b) in DOLOS the scheduled application is executed. Note that two different transition functions are being used in the definition of the result, i.e. δ_{olos} in DOLOS while in COA the function δ_{lc} is applied directly. The simulation relation holds after this computation, too.

In the local send phase (2c), the sender in the next slot updates the send buffer of the ABC with the message to be broadcast in the next slot. In the global send phase (3) this send buffer is written to the new-bus component. Similar, in the COA global send

phase (iii) the message to be broadcast in the next slot is taken from the message-buffer of the sender and is written in the new-bus component directly. Thus the simulation relation holds at the start of the next slot:

$$coasim(dolos((r,s)+1), coa((r,s)+1))$$

Note that during the global send phase an interrupt is generated on all ECUs in the DOLOS model. Thus after the global send phase, all ECUs in the DOLOS model start in the local receive phase again, due to the definition of the result of an OLOS computation from Sections 20.2. Hence, the initial conditions of the above theorem also hold for slot $(r,s)+1$ and are therefore invariant.

Formalization. Note that we have formalized the COA model in the theorem prover Isabelle/HOL resulting in about 1000 lines of code. In addition we have proven the above COA correctness theorem in Isabelle/HOL, as reported in [Kna08]. The proof is about 3500 lines of code long.

21.4 Pervasive COA Correctness

Note that in the step from DOLOS to COA some communication related details are hidden. However, the configurations of the applications themselves are not altered.

We consider some OLOS configuration $olos$ and some ECU configuration ecu. Let $ecusim(olos, ecu)$ hold. Then the ECU configuration ecu can be decoded from the OLOS configuration $olos$ using a function dec_{ecu}:

$$ecu = dec_{ecu}(olos)$$

Given some precondition E''' holds for the ECU configuration, we assume that the post condition Q''' is satisfied after the execution of δ_{coa}. The corresponding pre and post conditions for the OLOS level are given by:

$$\begin{array}{rcl} H_{olos}(E''') & = & \{olos \mid dec_{ecu}(olos) \in E'''\} \\ H_{olos}(Q''') & = & \{olos \mid dec_{ecu}(olos) \in Q'''\} \end{array}$$

We consider applications having the following structure: The application code starts at address c and ends with an invocation of the $ttExFinished$ system call at address d. The system call returns the control back to the operating system.

We require that the WCET of the application fits in one slot together with the WCET of the OLOS code:

$$\begin{array}{c} term_{ecu}(E'', c, d, a) = (WCET(H_{\mathrm{h}}(H_{isa}(H_{olos}(E''))), c, d) < T - off - C - \\ (WCET(H_{\mathrm{h}}(H_{isa}(H_{olos}(E''))), 0, c) + WCET(H_{\mathrm{h}}(H_{isa}(H_{olos}(E''))), d, a)) \end{array}$$

Note that the above requirement implies the termination requirement for DOLOS from Section 20.4:

$$term_{ecu}(E''', c, d, a) \Rightarrow term_{olos}(H_{olos}(E'''), c, d, a)$$

For any slot (r,s) let $ecu_u(r,s)$ and $olos_u(r,s)$ denote the ECU and the OLOS configuration with number u, i.e. $ecu_u(r,s) = coa(r,s).em(u)$ and $olos_u(r,s) = dolos(r,s).lm(u)$.

Theorem 18 (Pervasive COA Correctness) *Let all simulation relations down the implementation hold at the start of the slot* (r, s) *i.e.* $dhsim(s, dimpl(r, s), dh(r, s))$ *and* $disasim(dh(r, s), disa(r, s))$ *and* $dcvmsim(disa(r, s), dcvm(r, s))$ *and furthermore* $dolossim(dcvm(r, s), dolos(r, s))$ *as well as* $coasim(dolos(r, s), coa(r, s))$*. Let a pre and post condition hold for an ECU, i.e.* $ecu_u(r, s) \in E'''$ *and* $ecu_u((r, s) + 1) \in Q'''$ *and let the ECU terminate within one slot, i.e.* $term_{ecu}(E''', c, d, a) = 1$*. Then, at the end of the slot, the decoded implementation obeys the postcondition* Q'''*:*

$$dec_{ecu}(dec_{olos}(dec_{cvm}(dec_{isa}(dec_{\mathrm{h}}(dimpl((r, s) + 1).lm(u)))))) \in Q'''$$

The theorem is proven by induction over all slots (r, s). Using the COA correctness (Theorem 17) we obtain that:

$$coasim(dolos((r, s) + 1), coa((r, s) + 1))$$

Thus we derive that the OLOS configuration at the start of slot $(r, s) + 1$ obeys the post-condition, i.e. $olos_u((r, s) + 1) \in H_{olos}(Q''')$.

We apply the pervasive DOLOS correctness (Theorem 16) using the pre and post-conditions $H_{olos}(E''')$ and $H_{olos}(Q''')$ to prove the theorem.

22 Conclusion

We have presented a complete model stack for a distributed system starting at the asynchronous gate-level implementation and going up to the synchronous simple-to-use COA model. Using the methodology described in this thesis the pervasive verification of industrial-sized real-time systems becomes feasible.

Note that this model stack results from the joint work of many people. In particular, the model stack for a single ECU introduced in Part I has been subject to research for several years within the Verisoft Project [Ver08b]. With the number of people that are working on parts of the model stack increasing, the integration of the achieved results into a single consistent theory becomes more and more difficult.

The contributions of the author are the following: based on our early approaches in the formalization of the OSEKtime and the FlexRay standard [IdRK05, BBG+05] we have elaborated on the ABC correctness [KP07b] and, in particular, on the scheduler correctness [ABK08b]. Based on [KP07a] we have integrated already established theorems in the analysis of the model stack. Besides programming an initial OLOS implementation in C0, we have defined the ABC in Isabelle/HOL based on boolean gates (inspired by [Pau05]), and we have detailed the integration of devices throughout the stack, see also [AHK+07]. We have formalized OLOS, the ABC, the distributed framework (DISTR), and COA in Isabelle/HOL, too. In particular, our new DISTR model is vital for the extension of the model stack for single ECUs to a model stack for the complete distributed system. We have shown how to instantiate DISTR with all local models to obtain the corresponding distributed models, i.e. DH, DISA, DCVM and DOLOS.

All presented models are linked by simulation theorems. We have shown how to propagate the WCET of applications and systems software throughout the model stack resulting in a pervasive correctness proof for the complete distributed system.

To illustrate the applicability of DISTR we have formally proven the COA correctness (Theorem 17) in Isabelle/HOL, as reported in [Kna08]. All in all we have contributed approximately 9.000 lines of Isabelle/HOL code to the formalization of the model stack (as detailed in the corresponding sections) and an additional 2.800 lines in form of the ABC implementation.

Besides, we have supervised the master thesis of Peter Böhm [Böh07] which reports on the formal ABC scheduler correctness proof and we have coordinated the work of Mareike Schmidt on the formal OLOS correctness (see Section 12).

Last but not least, we have summarized and highlighted the essential parts of the whole theory on a few pages (based on [KP07a]), which can be taught in computer science lectures.

23 Future Work

The formalization of the local models and the local correctness theorems presented in Part I has either been completed already [LP08] or will hopefully be completed soon [Tve08, IdRT08]. However, the initialization (power-up) of the system has not been addressed in detail, yet.

The formal argumentation regarding the ABC implementation has been approached by Schmaltz and Böhm [Sch06, Böh07]. However, the results still need to be integrated following the arguments from Section 17.

One of the challenging proofs that is still left to be formalized in the theorem prover is the distributed processor correctness. Arguments regarding the liveness offset and the WCET have to be formalized and combined to obtain the DISA model.

To prove the distributed correctness theorems for the upper layers of the model stack the corresponding local correctness theorems need to be applied. The argumentation has been simplified a lot due to the DISTR framework. Note that the global send and receive phases are literally the same for each instantiation. In addition, the argumentation in the global compute phase is purely local allowing a straightforward application of the local correctness theorems.

The pervasive correctness theorems pose another challenge. So far, the transfer of properties between several layers of abstraction has not been argued about in the theorem prover.

Hereby we encourage the community to have a closer look at the theories and to extend and prove the remaining theorems along the lines of Part II.

24 Abbreviations

Single electronic control unit (ECU):

Abbreviation	Meaning
isa	instruction set architecture (ISA) configuration (see Section 4)
h	hardware (H) configuration (see Section 5)
d	device (D) configuration (see Section 6)
$impl$	implementation (IMPL) configuration with a device (see Section 6.4)
hd	H configuration with a device (see Section 6.5)
$isad$	ISA configuration with a device (see Section 6.6)
vm	virtual machine configuration (see Section 7)
co	C0 machine configuration (see Section 8)
cvm	communicating virtual machine (CVM) configuration (see Section 10)
$olos$	OSEKtime-like operating system (OLOS) configuration (see Section 12)
lc	local configuration (LC) (see Section 12.3)
δ_{isa}	ISA transition function (see Section 4)
δ_{h}	hardware transition function (see Section 5)
δ_{isad}	ISA transition function with devices (see Section 6.6)
δ_{hd}	hardware transition function with devices (see Section 6.5)
δ_{co}	C0 machine transition function (see Section 8)
$\delta_{co_{\mathrm{A}}}$	$\mathrm{C0_A}$ machine transition function (see Section 9)
δ_{cvm}	CVM transition function (see Section 10)
δ_{olos}	OLOS transition function (see Section 12)
δ_{lc}	LC transition function (see Section 12.3)

Distributed System (Several ECUs):

Abbreviation	Meaning
$dimpl$	distributed implementation (DIMPL) configuration (see Section 15)
$distr$	distributed framework (DISTR) configuration (see Section 16)
dh	distributed hardware (DH) configuration (see Section 17)
$disa$	distributed ISA (DISA) configuration (see Section 18)
$dcvm$	distributed CVM (DCVM) configuration (see Section 19)
$dolos$	distributed OLOS (DOLOS) configuration (see Section 20)
coa	communicating OLOS automata (COA) configuration (see Section 21)
δ_{distr}	DISTR transition function (see Section 16)
δ_{dr}	DISTR global receive phase (1) transition function (see Section 16)
δ_{dc}	DISTR global compute phase (2) transition function (see Section 16)
δ_{ds}	DISTR global send phase (3) transition function (see Section 16)
δ_{coa}	COA transition function (see Section 21)
δ_{cr}	COA global receive phase (i) transition function (see Section 21)
δ_{cc}	COA global compute phase (ii) transition function (see Section 21)
δ_{cs}	COA global send phase (iii) transition function (see Section 21)

References

[ABK08a] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Correctness of a Fault-Tolerant Real-Time Scheduler Algorithm and its Hardware Implementation. In *Formal Methods and Models for Codesign (Memocode'08)*, pages 175–186. IEEE Computer Society Press, 2008.

[ABK08b] Eyad Alkassar, Peter Böhm, and Steffen Knapp. Formal Correctness of a Gate-Level Automotive Bus Controller Implementation. In *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'08)*. Springer Science and Business Media, 2008. To appear.

[Abs07] AbsInt Angewandte Informatik GmbH, 2007. `www.absint.com`.

[AHK+07] Eyad Alkassar, Mark Hillebrand, Steffen Knapp, Rustilav Rusev, and Sergey Tverdyshev. Formal Device and Programming Model for a Serial Interface. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY'07), Bremen, Germany*, pages 4–20. CEUR-WS Workshop Proceedings, 2007.

[AT08] Eyad Alkassar and Sergey Tverdyshev. Efficient bit-level model reductions for automated hardware verification. In *15th International Symposium on Temporal Representation and Reasoning (TIME'08)*. IEEE Computer Society Press, 2008. To appear.

[Aut08] AutoFocus Project, 2008. `http://autofocus.in.tum.de`.

[BBG+05] Sven Beyer, Peter Böhm, Michael Gerke, Mark Hillebrand, Thomas In der Rieden, Steffen Knapp, Dirk Leinenbach, and Wolfgang J. Paul. Towards the Formal Verification of Lower System Layers in Automotive Systems. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'05), San Jose, CA, USA, Proceedings*, pages 317–324. IEEE, 2005.

[BBG+08] Jewgenij Botaschanjan, Manfred Broy, Alexander Gruler, Alexander Harhurin, Steffen Knapp, Leonid Kof, Wolfgang J. Paul, and Maria Spichkova. On the Correctness of Upper Layers of Automotive Systems. In *Formal Aspects of Computing*, 2008. To appear.

[Bey05] Sven Beyer. *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An Approach To Systems Verification. *Journal of Automated Reasoning (JAR)*, 5(4):411–428, December 1989.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In *Proc. of the 12th Advanced*

Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03), LNCS, pages 51–65. Springer, 2003.

[BJK+05] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together - Formal Verification of the VAMP. *STTT Journal*, 2005.

[BKS03] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System Level Design and Verification Using a Synchronous Language. In *ICCAD'03*, pages 433–440, 2003.

[BP06] Geoffrey M. Brown and Lee Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *LNCS*, pages 58–72. Springer, 2006.

[BY91] William R. Bevier and William D. Young. The proof of correctness of a fault-tolerant circuit design. In *Second IFIP Conference on Dependable Computing For Critical Applications (DCCA'91), Tucson, Arizona*, pages 107–114, 1991.

[Böh07] Peter Böhm. Formal Verification of a Clock Synchronization Method in a Distributed Automotive System. Master's thesis, Saarland University, Saarbrücken, Germany, 2007.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang J. Paul. On the Verification of Memory Management Mechanisms. In *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a Practical, Verified Kernel. In *11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA, 2007.

[Fle06] FlexRay Consortium. FlexRay Communications System Specifications Version 2.1, 2006.

[Fle08] FlexRay Consortium, 2008. `http://www.flexray.com`.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the Correctness of Operating System Kernels. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[HEK+07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. *ACMSIGOPS*, 41(3):9, July 2007.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *ICCD'05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, second edition, 1996.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Programm Languages and Operating Systems*, July 2005.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 165–169. Association for Computing Machinery (ACM), 2002.

[IdR08] Thomas In der Rieden. *CVM – A formally verified framework for microkernel programmers*. PhD thesis, Saarland University, Saarbrücken, Germany, 2008. To appear.

[IdRK05] Thomas In der Rieden and Steffen Knapp. An Approach to the Pervasive Formal Specification and Verification of an Automotive System (Status Report). In *Tenth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'05)*, 2005.

[IdRT08] Thomas In der Rieden and Alexandra Tsyban. CVM - A Verified Framework for Microkernel Programmers. In *3rd intl Workshop on Systems Software Verification (SSV'08)*. Elsevier Science B.V., 2008.

[KMP00] Daniel Kroening, Silvia M. Mueller, and Wolfgang J. Paul. Proving the Correctness of Processors with Delayed Branch using Delayed PCs. In *Proceedings of the Symposium on Numbers, Information and Complexity, Bielefeld*, pages 579–588. Kluwer Academic Publishers, 2000.

[Kna05] Steffen Knapp. Towards the Verification of Functional and Timely Behavior of an eCall Implementation. Master's thesis, Saarland University, Saarbrücken, Germany, 2005.

[Kna08] Steffen Knapp. Pervasive Layered Verification of a Distributed Real-Time System. In *Third International Conference on Systems (ICONS'08)*, pages 323–328. IEEE Computer Society Press, 2008.

[KP07a] Steffen Knapp and Wolfgang J. Paul. Pervasive Verification of Distributed Realtime Systems. In T. Hoare M. Broy, J. Grünbauer, editor, *Software System Reliability and Security*, volume 9 of *NATO Security Through Science Series. Sub-Series: Information and Communication*. IOS, 2007.

[KP07b] Steffen Knapp and Wolfgang J. Paul. Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, volume 4444 of *LNCS*, pages 53–81. Springer, 2007.

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Saarbrücken, Germany, 2001.

[Lei08] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, Germany, 2008.

[LMS85] Leslie Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, 1985.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive Compiler Verification – From Verified Programs to Verified Systems. In *3rd intl Workshop on Systems Software Verification (SSV'08)*. Elsevier Science B.V., 2008.

[MJ96] Paul S. Miner and Steve D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. In *Designing Correct Circuits*. Springer, 1996.

[Moo03] J S. Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2003.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version 1999.

[Nor98] Michael Norrish. C Formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs'07, Kaiserslautern, Germany*, pages 189–206. LNCS, 2007.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE'92)*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[OSE08a] OSEK/VDX Group. *Fault-Tolerant Communication (FTCom) Layer*, 2008. `http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf`.

[OSE08b] OSEK/VDX Group. *Time Triggered Operating System Specification*, 2008. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.

[Pau05] Wolfgang J. Paul. Lecture Notes: Computer Architecture II (Automotive Systems) WS0506, 2005. `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur2/ws0506/temp/060302_CA2_AUTO.pdf`.

[Pau08] Wolfgang Paul. Lecture Notes: Computer Architecture I WS0708, 2008.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Saarbrücken, Germany, 2007.

[Pik07] Lee Pike. Modeling Time-Triggered Protocols and Verifying Their Real-Time Schedules. In *FMCAD'07*, pages 231–238, 2007.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In Bernhard Steffen, editor, *TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In *Dependable Computing for Critical Applications (DCCA'99)*, volume 12, pages 207–226, San Jose, CA, 1999. IEEE Computer Society.

[Rus99] John Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

[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 *LNCS*, pages 83–105, Oldenburg, Germany, September 2002. Springer.

[Sch06] Julien Schmaltz. A Formal Model of Lower System Layers. In *FMCAD'06*, pages 191–192, Los Alamitos, CA, USA, November 2006. IEEE Computer Society.

[Sch08] Mareike Schmidt. *The Formal Correctness of Real-Time Operating System*. PhD thesis, Saarland University, Saarbrücken, Germany, 2008. To appear.

[SH98] Jun Sawada and Warren A. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV'98*, pages 135–146. Springer, 1998.

[Sha92] Natarajan Shankar. Mechanical Verification of a Generalized Protocol for Byzantine Fault Tolerant Clock Synchronization. In *FTRTFT'92*, volume 571, pages 217–236, Netherlands, 1992. Springer.

[SHS+97] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture (TTA). In *Advances in Information Technologies: The Business Challenge*. IOS, 1997.

[ST08] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In *3rd intl Workshop on Systems Software Verification (SSV'08)*. Elsevier Science B.V., 2008.

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the EROS Confinement Mechanism. In *IEEE Symposium on Security and Privacy (SP'00)*, pages 166–176. IEEE, 2000.

[Tsy08] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken, Germany, 2008. To appear.

[Tve08] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Saarbrücken, Germany, 2008. To appear.

[Ver08a] Verifix. *Verifix Project Website*, 2008. `http://www.info.uni-karlsruhe.de/~verifix`.

[Ver08b] The Verisoft Project, 2008. `http://www.verisoft.de`.

[Win93] Glynn Winskel. *The formal semantics of programming languages: An introduction*. MIT Press, Cambridge, MA, USA, 1993.

[WL88] Jennifer Lundelius Welch and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Communication*, 77(1):1–36, April 1988.

[Zha06] Bo Zhang. On the Formal Verification of the FlexRay Communication Protocol. *Automatic Verification of Critical Systems (AVoCS'06)*, pages 184–189, 2006.