Formal Verification of Gate-Level Computer Systems

Dissertation

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

Sergey Tverdyshev

deru@wjpserver.cs.uni-sb.de

Saarbrücken, Mai 2009

Tag des Kolloquiums:	15. Mai 2009
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Bernd Finkbeiner
akademischer Mitarbeiter:	Dr. Mark A. 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, im Mai 2009

Acknowledgment

At this place I would like to thank all those who have contributed to the work described in this thesis.

First of all, I would like to thank my parents and my sister Sveta which have supported me during my whole education. They have managed that despite 10.000 kilometers which separate us.

I thank my girl-friend Lena for her infinite patience, encouragement, and being on my side despite 200 kilometers between us.

My special thanks go to Prof. Paul for giving me an interesting research topic and scientific guidance. I highly appreciate his involvement starting from my master study in Saarbrücken up to the point where I am now.

I am very thankful to Dirk Leinenbach and Mark Hillebrand for their priceless advices in technical writing.

Furthermore, I thank Eyad Alkassar for fruitful discussions and Irena Dotcheva for her careful proofreading.

Last but not least, I thank all the members of Prof. Paul's chair for a nice working and after-working atmosphere.

This thesis work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Abstract

This thesis presents the formal verification of a gate-level computer system. This computer system consists of a microprocessor called VAMP and a generic device model. The VAMP processor is a 32 bit RISC CPU featuring a DLX instruction set, out-of-order execution, precise interrupts, and address translation. The generic device model is a formal framework which can be instantiated with arbitrary devices. We verify the gate-level computer system against a model as seen by an assembly programmer.

As proof of concept, we instantiate the verified computer system with an automotive bus controller. Thus, we built and verified an electronic control unit for a distributed automotive system.

This work is a part of the Verisoft project which targets pervasive formal verification of an entire computer system stack. Our results provide the base for the Verisoft computer system stack and the justified abstraction of the hardware which is suitable for system programming.

We employ the interactive theorem prover Isabelle/HOL as the verification environment for hardware. To decrease the user's involvement in the verification process, we develop an environment for the hardware design and verification called IHaVeIt. We integrate it into Isabelle/HOL. We show that the usage of IHaVeIt can save up to 40% of the user work.

Kurzzusammenfassung

In dieser Arbeit beschreiben wir die formale Verifikation eines Computersystems. Das System ist definiert auf Gatterebene und besteht aus einem Prozessor (VAMP) und einem generischen Gerätemodell. Der VAMP ist ein 32-Bit RISC Prozessor mit DLX–Instruktionssatz, out-of-order Instruktionausführung, präzisen Interrupts und Adressübersetzung. Das generische Gerätemodell ist eine formale Umgebung, die mit beliebigen Geräte instanziiert werden kann. Wir beweisen einen Simulationssatz zwischen dem Computersystem auf Gatterebene und einem Modell aus Sicht eines Assembler-Programmierers.

Als Fallbeispiel instanziieren wir das verifizierte Computersystem mit einem Bus-Controller, der in einem verteilten Automotive-System eingesetzt wird.

Diese Arbeit ist Teil des Verisoft-Projektes, welches eine durchgängige formale Verifikation eines kompletten Computersystems, bestehend aus Hardware und Software (z.B. Betriebssystem oder E-Mail Client) anstrebt. Unsere Ergebnisse stellen die Hardware-Basis für dieses Computersystem bereit sowie eine Abstraktion dieser Hardware, die zur Systemprogrammierung geeignet ist.

Für die Verifikation der Hardware benutzen wir den interaktiven Theorembeweiser Isabelle/HOL. Wir erweitern Isabelle/HOL mit dem Tool IHaVeIT, welches als Umgebung zur Entwicklung und automatischen Verifikation von Hardware dient. Wir zeigen, dass die Anzahl von Beweisschritten mittels IHaVeIt um bis zu 40% reduziert

werden kann.

Extended Abstract

Modern computer systems are used in many safety-critical applications. In order to guarantee an error-free behavior of such a system one often employs formal methods, e.g. model checking, and theorem proving. However, usually formal methods are only applied to stand-alone components or to abstract models. Thus, pervasive correctness of computer systems, with all their details, can not be guaranteed.

The goal of the Verisoft project [The03] is to show that it is possible to build and to verify a computer system, which consists of a hardware platform with devices, a compiler for a C-like language, a micro-kernel, an operating system, and user applications. The goal is to verify these components and to guarantee formally that they can be combined into one computer system stack. In this paper, we present the verification of a complex *gate-level* computer system. This hardware is formally verified against an assembly-level model, i.e. a model as seen by an assembly programmer.

All models in the Verisoft project are defined/verified in Isabelle/HOL, which is an interactive theorem prover for higher-order logic. To reduce the user work, we developed an environment IHaVeIt [Tve05a, TA08]. It couples Isabelle/HOL with external tools, such as model checkers (NuMSV [CCG+02] and SMV [McM99]) and SAT solvers. It also provides several reduction and abstraction techniques which increase the application efficiency of the external tools. In this thesis we develop and verify the hardware in Isabelle/HOL and then automatically translate it into Verilog (via IHaVeIt) and run it on an FPGA.

The verified computer system consists of a VAMP processor [DHP05] and a generic device model for memory mapped devices.

The VAMP processor features the DLX instruction set, out-of-order execution, precise interrupts, delayed branch, and support for virtual memory. We verified the gate-level processor against a model as seen by an assembly programmer, i.e. a model which executes a complete instruction with every step.

The device model on the gate level is modelled as an I/O automaton. It can contain arbitrary devices which run in parallel and communicate with the external environment, e.g. with a network. This model is verified against an interleaved device model where the devices progress one after another.

The gate-level computer system is built by a parallel composition of the VAMP and the gate-level device model, which communicate via a bus with a common clock. This system is verified against an assembly-language model which executes with every transition either a device step or the processor step, with or without device access. The correctness criterion states that every run of the gate-level model can be simulated by a run of the assembly-level model. The criterion is proved via a non-trivial combination of the proofs for the processor and the generic device model. The proof is carried out interactively in Isabelle/HOL with the help of IHaVeIt.

Finally, we instantiated the verified computer system, on the gate and assembly levels, with an automotive bus controller. Thus, we built a verified electronic control unit for a distributed automotive system [Kna08]. This unit has been synthesised and run on a Xilink FPGA. The size of the unit is 5,180,002 gates.

For the first time, we report on the formal verification of a gate-level computer system, which consists of a non-trivial processor and devices. The closest related work is the famous CLI stack [BHMY89], where the verification of a small computer system stack is reported. Our work also closes the gap between verification of the devices [Coh00, Rus02, BKS03, ALD06, RPS01] and the processors [MS06, Vel05, ADJ04, SJ02, HGS03, BJK+05, DHP05] as stand-alone components.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays modern computer systems are extensively used in safety and security critical areas. These systems are taking control in cars, trains, airplanes, military weapon and defence systems, space shuttles and sputniks, and medical devices. An error in any of these systems can cause not only huge financial loss, but also loss of human lives. The developers and testers of these systems spend tremendous efforts trying to catch all errors in the final product. They use a number of methods such as testing, simulating, introducing redundancy in the final product, and applying formal methods. Unfortunately, it was not always possible to find and to exclude all errors. These are some results if you "google" *top hardware and software bugs*:

- Mariner I space probe [Neu89] – A bug in the software caused the rocket to divert its path and it was automatically destroyed. The reason was that a formula written on *paper* was not correctly transferred to the code.
- Therac-25 medical accelerator [LT93] – A radiation therapy device delivered lethal radiation dose. At least five patients died, others were seriously injured. The major reasons were reuse of the software for the previously developed hardware, race conditions in the software, and an overflow error which allows to skip a safety check.
- Pentium FDIV bug [Hal95] – A flaw in the FPU caused mistakes when dividing floating-point numbers which occur within a specific range. Intel started a big call-back campaign; this led to huge financial losses (ca. 500 million US dollars).
- Ariane 5 Flight 501 [Nus97] – The code for the Ariane 4 rocket was reused, but the new faster engines together with the different flight path triggered a bug in an arithmetic routine (data conversion from a 64-bit floating point to a 16-bit signed integer) inside the rocket's flight computer. This led to a cascade of problems, culminating in the explosion of the rocket.
- BMW 5 series [Lef04] – Exceeding pressure on the break pedal disabled all stability systems (ABS and DCS). These systems can only be turned on after

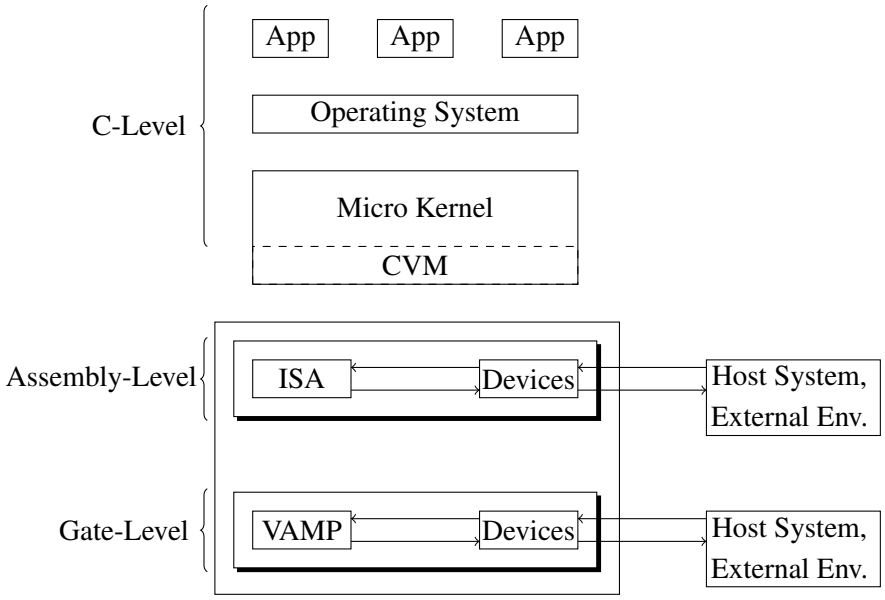


Figure 1.1: The Verisoft computer system stack.

the ignition is switched off for five seconds. The error was tracked back to a micro-controller.

These and others failures are often due to the incomprehensive testing of system components; also combining tested/verified components into one system can result in an unexpected behaviour of the combined system. Note that comprehensive testing of big systems can not be achieved in practice, because there are too many possible test cases. Nowadays, testing is extened with formal verification where the design is formally proved, via a mathematical proof, to comply with its specification, i.e. the design is 100% correct with respect to a given specification. There are industry reports stating that verification costs are less than those of testing [BBM+07]. On the other hand, considering a system constructed from verified components as a whole can help to exclude global malfunctions. For example, one considers a computer system as a whole, starting from the gate-level hardware up to the running software.

The Verisoft project [The03] worked out a methodology for developing and pervasively verifying an entire computer system. This computer system can be represented as a stack (Figure 1) which consists of a gate-level hardware, an assembly-level machine, a micro kernel, an operating system, and user applications. The hardware is developed and verified as part of this thesis and is based on the previous correctness proofs of the VAMP processor in PVS [BJK+05, DHP05]. A micro kernel [DDS08] provides a basic functionality for the low-level management of the system resources (e.g. memory allocation). The CVM [IdRT08] part of the micro kernel allows us to run many virtual machines on the same hardware, i.e. user

processes with their own virtual memory. The micro kernel comes with a scheduler which is proved to be fair [DDSW08]. The simple operating system (SOS) [Bog08] is a multi-user operating system. SOS also provides the user visible interfaces to the devices, e.g. a network socket, a file system. Finally, verified user applications, such as an e-mail client [BB05] and an SMTP server [LNRS07], are executed. The software part in this project is mainly implemented in a C-like language with portions of assembly code. Therefore, a compiler for the VAMP instruction set architecture was developed and verified [LPP05, LP08].

Most proofs in the Verisoft project are carried out in Isabelle/HOL, which is an interactive theorem prover. This allows the Verisoft team to guarantee the pervasiveness of the verification results, because all computational models are defined in one language. Thus, a theorem that is proved for one layer (or module) can be easily reused in the upper layers (or connected modules).

As part of this thesis, we developed and verified a hardware platform which consists of the VAMP processor [BJK+05, DHP05] and memory mapped devices. This hardware platform is the base for the Verisoft computer system.

Originally, the task was to take the VAMP processor developed in PVS, which is an interactive theorem prover, and to translate automatically the PVS constructions and proofs into the Isabelle/HOL language. Here, we have to split the task into two subgoals: (i) translation of the gate-level model of the VAMP and its specification, and (ii) translations of the PVS proofs into Isabelle/HOL proofs. The main problem of the first subgoal is that PVS natively supports subtypes (e.g. bit vectors of a given length) and Isabelle/HOL does not. The second subgoal is more challenging, and this task is hardly achievable, if feasible at all. The main reason is the different nature of the PVS and Isabelle proof engines. Obviously, it is impossible to syntactically match a PVS proof step to an Isabelle proof step. This syntactic match would only work out for the very basic steps, such as conjunction elimination, introduction rules, etc. However, the correctness proofs of the VAMP processor mostly rely on powerful PVS tactics, e.g. *grind* [SORSC01], and the team of the VAMP project definitely made the proofs as efficient as possible. Since the VAMP is verified in a closed-source system PVS[1], there is no way to understand how these tactics are working/implemented. Therefore, it is not clear that a sequence of the Isabelle proof steps can repeat a step of a PVS proof. For example, Skalberg et al. [OS06] implemented a tool for translating proofs and theories of the HOL theorem prover into Isabelle ones. The authors estimate their efforts to eight person months [TS07]. The implementation of this tool was only possible thanks to the similar proof engines, because Isabelle is a successor of the HOL theorem prover. Therefore, we decided to prove the correctness of the VAMP processor directly in Isabelle and not to translate the PVS proofs into Isabelle. This also allows us more flexible changing of the VAMP design.

To speed up the verification process in Isabelle, we decided to employ automatic proof techniques, such as model checking and SAT solving. Thus, in the scope of this thesis, we developed an environment for hardware design and verification in Is-

[1] Since December 2006 PVS is an open-source system.

abelle/HOL. We also verified the VAMP processor and decreased the user involvement into the verification process with respect to the previous VAMP project. Moreover, the usage of the VAMP processor in the computer system requires a complete hardware platform including external devices, e.g. hard drive disk and serial interface. We successfully extended the processor with a full support of the memory mapped devices and created a formally verified generic device theory. Thus, we built a hardware platform with external devices and verified it against a model with devices as seen by an assembly programmer. Finally, we instantiated the device theory with a concrete device.

1.1 Outline

In the rest of this chapter, we present the software tools which are used and are relevant to this thesis. In Section 1.3, we discuss related work. Finally, we present the notation which is used in this thesis.

In Chapter 2, we present IHaVeIt [TA08, Tve05a], the environment for hardware design and verification in Isabelle/HOL. It supports our interactive proofs of the developed hardware platform. We present the reduction and transformation algorithms behind IHaVeIt and give their correctness proofs. At the end of this chapter, we present several benchmarks and describe other applications of IHaVeIt.

In Chapter 3, we describe the VAMP processor [DHP05, BJK$^+$05] on the gate level and give its instruction set architecture (ISA). ISA is a specification model, which processes one instruction with every step. We present modifications which are needed in order to make the VAMP fit for the integration into a platform with external devices. We also prove the correctness statement of the VAMP hardware against ISA.

In Chapter 4, we present a generic device theory. This theory defines two generic models, one for the gate level and one for the assembly level. The gate-level model supports the parallel execution of many devices and the assembly-level model is a purely sequential one. We formulate a correctness statement and give the correctness proofs.

In Chapter 5, we build a hardware platform. On the gate level, we combine the VAMP with the generic device model. We verify this platform against a model where the processor and the devices are progressing one after another. In this chapter, we show how the VAMP proofs and the proofs of the device theory imply the overall system correctness.

In Chapter 6, we instantiate the device theory with a concrete device. Thus, we build a concrete hardware platform, which is formally verified.

Finally, in Chapter 7, we give a summary of this work and point out the direction of future work.

1.2 Tools

Modern formal verification is supported by mechanised systems. The landscape of these systems is very wide and can be split into three groups: interactive, automated, and hybrid systems. The interactive tools usually have a very expressive specification language, but require a substantial user involvement in the verification process. The automated systems come with a less expressive language (e.g. propositional or first order logic) and require the user only to push the button. The hybrid systems are various combinations of interactive and automatic tools. In the following subsections, we shortly present the systems which are used and are relevant to this thesis.

1.2.1 Interactive Tool

PVS

The *Prototype Verification Systems* (PVS) [OSR92] is an interactive theorem prover developed at SRI. The PVS specification language is typed higher oder logic (HOL). The PVS system is based on a set of predefined libraries, e.g. for bit vectors. The system supports the usage of subtypes (e.g. consider only bit vectors of a specific length) and can automatically resolve many subtype violations (e.g. when the function signature specifies the result as a bit vector of length n, but function definition computes something else). PVS features powerful decision procedures and provides the user with standard proof techniques, e.g. induction, case distinctions, skolemization, application of lemmata, and quantifier instantiation.

We shortly present the PVS system, since the VAMP processor, which we consider in Chapter 3, was originally developed and verified in this system.

Isabelle/HOL

Isabelle [Pau94] is a generic interactive theorem prover that is developed at the Technical University of Munich and Cambridge University. Isabelle supports several object logics and we use it with its instantiation of HOL, which is referred to as Isabelle/HOL. Isabelle provides the user with an interactive mode that allows all standard proof techniques. It also provides the user with several automatic decision procedures, e.g. a term rewriting engine (called a simplifier) and a tableaux prover (called a classical reasoner). Isabelle/HOL comes with a bunch of theories such as lists, naturals, and integers.

In the academic part of the project Verisoft [The03] Isabelle/HOL is used as a major tool for system design and verification. In this thesis we use Isabelle/HOL as a design and verification environment for hardware.

1.2.2 Automated Tools

SAT and SMT

SAT (SATisfiability) solvers check the satisfiability of a formula which is expressed in propositional logic. Modern SAT solvers (e.g. zChaff [TYRM04], MiniSAT [EMS07]) are highly effective for checking huge formulae. One of the drawbacks is that propositional logic is not expressive enough for real-life applications. For example, in software and hardware verification formulae are to be tested modulo some background theories, e.g. linear arithmetic, uninterpreted symbols, and bit vectors. Therefore, SAT solvers are often used as back ends for other tools, e.g. model checkers, theorem provers, and SMT solvers.

Satisfiability Modulo Theories (SMT) solvers have additionally axioms about the supported theory. They use these axioms and SAT solving technique to check given formulae. The first versions of SMT solvers supported only one special theory, e.g. uninterpreted functions. Modern SMT solvers (e.g. Z3 [dMB07], Yices [DdM06]) support several theories and can be very useful in the practice, e.g. software verification [Sch07]. However, an efficient combination of many theories is still a non-trivial task [Pfe07].

Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems. The technique has a number of advantages over traditional techniques, such as simulation, testing, and deductive reasoning. The main disadvantage is the infamous state-explosion problem which arises when a system with numerous components has to be modelled, e.g. modelling two registers 64-bit each will induce a state space of the size 2^{128}.

To apply model checking technique the user defines (or compiles) his/her design in the language accepted by a model checker. The design specification is usually formulated in *temporal logic* [CGP99], which can assert how the system progresses over time. In the best case application of the model checker is fully automatic, but in practice user interaction is required, e.g. for analysing the verification results. In the case the model checker reports that the design does not comply with the specification, it also generates an error trace. This trace can be used as a *counter-example* and can help the designer to find an error which can be in the design or in the specification. Due to the mentioned state explosion problem the model checker can fail to terminate, e.g. the model size is too big to fit into the computer memory or to be explored in the user life time. A possible solution to avoid this problem is to change the model, e.g. by applying additional manual or automatic abstraction.

The progress in SAT solving opened a new branch in model checking: *bounded model checking*. This approach requires an additional parameter which specifies the length (or the bound) of the model runs to be checked. Such a model checker unrolls the model for a given number of steps and then uses a SAT/SMT solver to check the unfolded formula. Thus, if a bounded model checker confirms the validity of a model,

the model is only valid for a given number of steps. Usually these model checkers are fast and are very good for debugging.

In this thesis we consider two state-of-the-art model checkers NuSMV [CCG+02] and Cadence SMV [McM99].

NuSMV

NuSMV [CCG+02] is a symbolic model checker, that is a re-implementation of CMU SMV [JED+94], for CTL, LTL, and past-LTL properties. It can perform bounded model checking using an external SAT solver. We use NuSMV to verify temporal properties defined over Kripke structures and as an external BDD decision procedure. The NuSMV input language only allows to define finite state machines. The only data types provided by the language are booleans, bounded subset of integers, symbolic enumerations, and bounded arrays of supported types. The description of a complex system can be split into modules. The modules can be composed either synchronously or asynchronously. NuSMV allows modelling deterministic and non-deterministic systems. Last but not least NuSMV is an open source project and hence soundness of the implemented algorithms can be checked by anyone.

Cadence SMV

Cadence SMV[McM99] is a symbolic model checker for LTL properties. It supports a specification language similar to the one of NuSMV. It also provides some advanced features, e.g. support for bit-vector arithmetic, and predicate abstraction. We use SMV mainly as an external decision procedure for bit-vector arithmetic.

1.2.3 Hybrid Tools

The hybrid tools vary in many aspects. There are integrations of basic interactive proof features in the automatic tools, e.g. in the KeY System [MLH07]. In this case the user applies some top-level interactive steps and the system works further in an automatic mode. There are many attempts to connect interactive theorem provers with automatic tools. Such a combination helps the user to prove boring lemmata by applying automatic tools. The proof is still interactive and the user has still to recognize when and which automatic tool can be applied. For example, Shankar [RSS95] combined PVS theorem prover with a model checker, and Paulson et al. [MQP06] combined Isabelle/HOL with different first oder theorem provers.

In this thesis we present a further combination of a theorem prover with several external tools. The combination targets at the first place verification of the temporal properties of hardware systems by applying externals model checkers. In order to overcome the state explosion problem in the model checkers, we proposed several transformation algorithms. These algorithms are applied before calling a model checker and they are:

- user guided functional abstraction (Section 2.4.2)

- automatic data abstraction, which is based on the data independence of the verified system and can reduce the domain of the system state (Section 2.2.2)
- a transformation technique which allows an efficient handling of uninterpreted functions and memories (Section 2.3)

Our approach is implemented as the IHaVeIt environment which we present in Section 2.4.

1.3 Related Work

The work in this thesis covers several topics.

- Automatic verification tools, their automatic applications, and their usage in the hybrid verification systems.
- Efficient verification of microprocessors with out-of-order execution.
- Verification of a real-life gate-level computer systems.

Therefore, we split this section according to these topics.

1.3.1 Verification Tools

In this thesis we developed the environment IHaVeIt [TA08, Tve05a] which is based on the theorem prover Isabelle/HOL. The main contribution of IHaVeIt is the introduction of a preprocessor for Kripke structures that includes an automatic data abstraction and an elimination of function applications.

Automatic Tools

The principle of our data abstraction algorithm is related to symmetry reduction [ID96], abstraction of data insensitive models [LNR05, PMV98], and syntactic program transformation [NK00]. In contrast to symmetry reduction our algorithm works entirely on the symbolic level rather than on the explicit state transition graph.

Lazik et al. [LNR05] presented a semantic definition and a rigorous analysis of domain independent systems. Their domain reduction algorithm targets safety properties which can be expressed as reachability properties of concurrent programs. In contrast, we can verify arbitrary temporal properties.

Paruthi et al. [PMV98] presented a domain reduction algorithm that distinguishes three kinds of variables: control, data, and mixed ones. While their reduction can only be applied to data variables, we extend it to arbitrary variables.

Manolios et al. [MSV06] proposed a memory abstraction algorithm that is implemented in a bounded model checker. The algorithm basically consists of two parts: (1) unrolling the given transition relation and the property for a given number of steps, i.e. transforming it to a SAT problem (2) computing the number of required memory

cells and reducing memory domains to preserve validity of the the unrolled formula. This approach works for *bounded* model checking of properties of the form AG or AF. Furthermore, it supports domain-specific operations on addresses. Compared to his work, our approach is not restricted to bounded model checking and can be applied on any CTL* properties.

Namjoshi [NK00] proposed a syntactic predicate abstraction to compute an abstract version of a given program. Later the author presented a prototype implementation AUTOABC [Rob02] which could only handle fairly small examples.

Our function elimination is related to the classical "freeing" technique and is widely used in hardware verification. For example, it is applied in [GGA05] where the authors construct a verification model by eliminating memory arrays and retain only the memory interface signals. Their approach is implemented in a bounded model checker and supports only memory reads and writes (no memory comparison). However, to our knowledge IHaVeIt is the first tool applying function elimination on transition systems and temporal properties rather than on combinational ones.

Hybrid Tools

The coupling of heterogeneous systems is by no means a novel idea.

Müller [Mül98] connected external tools with Isabelle through input-output automata. In his approach the user defines a model and manually specifies its abstraction. Then an LTL model checker is used to prove temporal properties of the abstracted model and a μ-calculus model checker is used to check forward simulations between these two models. A drawback is that defining a suitable abstraction can be a very hard task. In contrast, our approach does not have this disadvantage because an abstracted model is derived fully automatically.

A very interesting ongoing work of L. Paulson's group [MQP06] is the integration of automatic theorem provers (ATP) SPASS [WBH$^+$02] and Vampire [RV01] into Isabelle. A highlight of this approach is that the proof generated by an ATP can be converted into an Isabelle proof and then rechecked by Isabelle. Thus, the user does not have to trust an external tool, and soundness of the translation can be guaranteed.

The UCLID system [LSB02] is another interesting tool that can handle big problems with much automation. It also has a lot of built-in features, e.g. handling of uninterpreted functions, efficient algorithms for term reduction. The UCLID system can be used for verification of system invariants (AG safety properties) but not for the verification of liveness properties [LSB02]. Our approach allows us verification of both kinds of properties. Integration with Isabelle increases the domain of applications of our tool but we have to pay for that with the user's involvement in the proof process.

1.3.2 Processor Verification

There is much work concerning processor verification. This work can be split into two general groups: processors which were verified by automatic or interactive tools.

The work from the first group is characterised by the absence of user involvement in the proof process. The main advantage of this approach is that the user solely specifies a model, a correctness criterion, and an abstraction function and then pushes the button. The application of this approach produces one of the following results: the model is correct, the model is not correct and a counter-example might be generated, or the tool ran out of memory or of the user's patience. There are several drawbacks of this approach. First of all the defining of an abstraction function can be a very hard task. Another drawback is that the size and the complexity of the considered processors are very restricted. A typical benchmark from this group is a pipelined processor with up to 5 stages in the pipeline and in-order execution of fetched instructions [Vel05, ADJ04, MS06, ACHK04]. These papers use variations on the method of Burch and Dill [BD94], where an abstraction function is constructed by flushing the implementation, i.e. inserting null operations until all pending instructions are completed. Often, to overcome the state explosion in the verification tool, one specifies the processor not on the bit-level but on the term-level. For instance, term-level models of processors only implement a subset of the instruction set architecture (ISA) and data paths, memories, and processor elements, such as decoders and ALU's are left uninterpreted. Moreover, there is often no obvious connection between the implementations and their abstractions [LSB02, VB99].

The interactive proof tools allow verification of very big and detailed processors, but they require user guidance [BHK94, SJ02, HGS03, Krö01, BJK+05, Dal06, Jac02]. For instance, Sawada and Hunt [SJ02] verified an entire processor with out-of-order execution, precise interrupts, and a store buffer for the memory operations. They treated self-modifying code and used *sync* instructions to avoid read-after-write hazards. McMillan [JM01] showed the correctness of an abstracted term-level processor with a Tomasulo scheduler with a high degree of automation. Sometimes, the actual processor implementations are extended with auxiliary constructs. These additional constructs simplify the proofs but can not be implemented in the real hardware. For example, Hosabettu [HGS03] used tags from an infinite set to distinguish instructions in the processor.

To the best of our knowledge, the VAMP processor [Krö01, Jac02, BJK+03, DHP05, BJK+05, Dal06] is the most complex verified processor presented in the open literature. The VAMP processor was first verified in the interactive theorem prover PVS. The correctness proofs of the VAMP employ only a negligible support of the automatic tools. Jacobi [Jac02] used the PVS theorem prover, the PVS built-in model checker, and SMV to verify complex models, such as floating point units. However, a substantial human interaction is still required because the abstraction is built manually and the conversion between PVS and SMV is done manually.

Our version of the VAMP processor has to fit in a computer system stack [The03]. The rest of the stack is developed in Isabelle/HOL and to guarantee the pervasiveness of the verification efforts we developed and verified the VAMP processor in Isabelle/HOL as well. The original plan was to reuse the proof strategy which was developed in the previous work in PVS. However, it is not always possible to reuse the previous results due to the differences between Isabelle/HOL and PVS, for example:

- Absence of the subtype mechanism in Isabelle/HOL.
- Isabelle/HOL does not support bit vectors, which makes the verification of the hardware more difficult.
- Applications of the complex built-in decision procedures of Isabelle/HOL and PVS on equivalent proof goals produce very different proof subgoals. Hence, from such a point a PVS proof can not be reused any more.
- We experienced that the built-in decision procedures of Isabelle/HOL are less powerful than those of PVS.
- PVS provides a better way to manage the proofs and the theories than Isabelle, e.g. proof visualisation.

Additionally, to speed up the verification process in Isabelle/HOL, we increase the usage of the automatic tools, i.e. we use the developed environment IHaVeIt. We have to note that application of automatic tools is most efficient when models and properties are *tailored* for the tool. In our case, it was not always possible to fine tune the models, because the proof strategy heavily employs the higher oder logic, and the automatic tools support at most first order logic. Nevertheless, we could decrease the user interaction by 30% compared the pure interactive proofs in PVS.[2]

1.3.3 Computer System Verification

One of the major results of this thesis is a formally verified computer system, which consists of a processor and external devices. The system implementation is a gate-level model and the system specification is a model as seen by an assembly programmer. These models are used as a base for a complete computer system stack (see Figure 1).

The closest related work on this topic is the famous CLI stack [BHMY89]. This stack consists of a non-pipelined processor [BHK94], an assembler [Moo94], a compiler for a simple high-level language [You94], and an elementary operating system kernel [Bev94]. The external devices were not considered in this work. In 2002 J. S. Moore [Moo02], the principal researcher for CLI stack, declared the formal verification of a computer system as a grand challenge. To the best of our knowledge up to 2002 there were no other attempts to take up this challenge.

In a Verisoft subproject Hillebrand et al. [HIdRP05] present the *paper-and-pencil* formalisations of a system with a hard disk drive. They define the system on the gate level and on the assembly level. They also sketch the correctness arguments which justify these models.

Devices are often modelled and verified as stand alone systems [Coh00, BKS03, ALD06, RPS01]. For instance, one models a device at one level and checks some properties of this model. In oder to use the devices in a computer system stack

[2] We compared the number of proof steps of the VAMP proofs in PVS with the amount of steps we used in Isabelle/HOL. For more details, we refer the reader to Section 3.4.

they have to be modelled at different levels. For example, in the automotive subproject of the Verisoft project, one deals with the verification of a time triggered bus interface, which is used in the distributed automotive systems. The verification of such a system requires considering this interface at different levels, e.g. low-level for clock synchronisation in serial interface [Sch06a], gate-level implementation, and real-time properties of the software driver for this bus interface. These models and proofs are finally combined into one pervasive correctness proof. A *paper-and-pencil* style description for the latter work can be found in Knapp and Paul [KP07]. In this thesis we show how a verified electronic control unit (ECU) for the distributed system can be constructed. This ECU is an instantiation of the gate-level computer system which is developed and verified in this thesis.

Alkassar et al. [AHK+07] present an assembly level model for a processor with I/O memory mapped devices. They also present a simple software driver for an UART device (serial interface) and proved its correctness on paper. They used an assembly-level model, which is connected to the machine-code model developed in this thesis, as an abstraction of a gate-level model.

1.4 Notation

In this section, we introduce some basic shorthands and notations used in this thesis.

1.4.1 Basics

For the whole thesis, we use two kinds of implication signs. We employ $\longrightarrow$ to denote the ordinary logical implication. The sign $\Longrightarrow$ is used to split the premises and conclusions of lemmata and theorems.

Whenever we introduce a definition, we employ symbol $\triangleq$. To keep definitions short and readable, we can introduce aliases via symbol := .

We employ $\mathbb{B}$ to denote the set of boolean values: $\mathbb{B} \triangleq \{True, False\}$. $\mathbb{N}$ denotes the set of natural numbers *including* zero and $\mathbb{N}^- \triangleq \mathbb{N} \setminus \{0\}$. The set of integers $\{\ldots, -1, 0, 1, \ldots\}$, we denote by $\mathbb{Z}$. We often use *hardware cycles* as the time notion on the gate level. The domain of hardware cycles is denoted by $\mathbb{T}$, and it is a subset of natural numbers $\mathbb{T} \subset \mathbb{N}$.

We use the following notation to denote the subranges of integers.

- $[n : m] \triangleq \{x \in \mathbb{Z} : \ x \leq m \ \wedge n \leq x\}$
- $]n : m] \triangleq \{x \in \mathbb{Z} : \ x \leq m \ \wedge n < x\}$
- $[n : m[\triangleq \{x \in \mathbb{Z} : \ x < m \ \wedge n \leq x\}$
- $]n : m[\triangleq \{x \in \mathbb{Z} : \ x < m \wedge n < x\}$

We use the same notation to denote subranges of natural numbers because $\mathbb{N} \subset \mathbb{Z}$.

Bit-Vectors

We employ standard notation for bits $a, b \in \{0, 1\}$ and bit vectors $x, y \in \{0, 1\}^n$ of length n. For the sake of the simplicity, especially in the conditional expressions, we identify bit values and boolean values. Thus, $a = 0 \equiv a = \mathit{False}$ and $a = 1 \equiv a = \mathit{True}$.

We can construct a bit vector by enumeration of all its elements, e.g. $[x_n, \ldots, x_0]$. We denote by x_i the value of the i^{th} bit of the bit vector x. Note that i must be less than the length of x otherwise x_i specifies an undefined result. We use notation $x[a : b]$ to access a subrange of bits from a to b, e.g. $x[1 : 0]$ is a bit vector with two elements $[x_1, x_0]$.

We use the notation $\|_b$ to access byte (eight bits) with the index b of a bit vector.

$$\mid a \mid_b \triangleq a[8 * (b + 1) - 1 : 8 * b]$$

A bit vector can be interpreted as a natural number, where the value of a bit is treated as natural number 1 or 0.

$$\langle x \rangle \triangleq \sum_{i \in [0:n-1]} x_i * 2^i$$

We use the notation $a +_n b$ to denote the sum of two bit vectors modulo 2^n, i.e. $\langle a +_n b \rangle \triangleq \langle a \rangle + \langle b \rangle \; mod \; 2^n$.

In this thesis we employ the usual comparison operators, such as $<, \leq$, to compare the naturals encoded by bit vectors. For example:

$$x \leq y \triangleq \langle x \rangle \leq \langle y \rangle$$

For details on bit vector arithmetic see also [MP00].

1.4.2 λ – Calculus

We use λ–notation [Bar84] to specify a function without giving this function a name. For example, $\lambda x.\ x + 1$ defines a function which increments a given argument. Such a function is called a λ–term. An application of such a λ–term is defined in the usual way $(\lambda x.\ x + 1)(y)$. The β–reduction can be used to unfold an application of such a function application, e.g.

$$(\lambda x.\ x + 1)(y) \stackrel{\beta}{\equiv} y + 1$$

We can update any function or λ–term with the help of the λ–notation. We use the notation $f(y := z)$ to denote updating of the function term f with the value z on the position y. The update is defined as follows:

$$f(y := z) \triangleq \lambda x. \begin{cases} z & : \; x = y \\ f(x) & : \; else \end{cases}$$

1.4.3 Hilbert's Choice Operator

The Hilbert choice operator *Some* specifies *an* element which satisfies a given predicate.

Definition 1.1
Let z be specified by *Some* operator with a predicate P, i.e. $z = Some\ x \in X.\ P(x)$ If in the domain X there exists an element which satisfies P, the element z has the property P. If there is no such element in X, we can not state anything about z:

$$\exists y.\ P(y) \Longrightarrow P(z)$$

There is a stricter version of the Hilbert Choice operator *The* which specifies *the* element which satisfies a given predicate.

Definition 1.2
Let z be specified by *The* operator with a predicate P, i.e. $z = The\ x \in X.\ P(x)$. If there is a unique element in X which satisfies the predicate P, the element z has the property P. Otherwise we can not state anything about z:

$$\begin{array}{l}(\exists y.\ P(y))\ \wedge (\forall x\ y.\ P(x) \wedge P(y) \longrightarrow x = y)\\ \Longrightarrow\\ P(z)\end{array}$$

1.4.4 Sequences

A sequence is a pair $\sigma = (l, seq)$, where

- $l \in \mathbb{N}$ defines the length of the sequence.
- $seq : \mathbb{N} \to \mathbb{S}$ is a mapping from natural numbers (position in the sequence) $n \in \mathbb{N}$ to the corresponding element $el \in \mathbb{S}$.

We employ the following operators to work with sequences:

- [] – an empty sequence
- $[x]$ – a sequence with one element x
- $len(\sigma)$ – it returns the length of the sequence σ, i.e. $len(\sigma) \triangleq \sigma.l$
- $\sigma(i)$ – it returns an element on the position i, i.e. $\sigma(i) \triangleq \sigma.seq(i)$. Note that if $len(\sigma) \leq i$ then the result is undefined.
- $x \in \sigma$ – if it holds, x is a sequence element:

$$x \in \sigma \triangleq \exists i < len(\sigma).\ \sigma(i) = x$$

- $\sigma_1 \circ \sigma_2$ – concatenation of two sequences:

$$\begin{aligned}
&\sigma = \sigma_1 \circ \sigma_2 \\
&\Longleftrightarrow \\
&len(\sigma) = len(\sigma_1) + len(\sigma_2) \wedge \\
&\forall i < len(\sigma_1).\ \sigma_1(i) = \sigma(i) \wedge \\
&\forall i < len(\sigma_2).\ \sigma_2(i) = \sigma(i + len(\sigma_1))
\end{aligned}$$

- $next(s, \sigma, P)$ – it returns the index of the next element above s satisfying predicate P. If there is not any such elements satisfying P, $next(s, \sigma, P)$ is undefined.

$$\begin{aligned}
&i = next(s, \sigma, P)\ \wedge \exists j < len(\sigma).\ s < j\ \wedge P(\sigma(j)) \\
&\Longrightarrow \\
&i < len(\sigma)\ \wedge s < i\ \wedge\ P(\sigma(i)) \wedge \\
&\forall j < i.\ s < j \longrightarrow \neg P(\sigma(j))
\end{aligned}$$

  We employ the shorthand $first(\sigma, P)$ to denote $next(-1, \sigma, P)$.

- $last(s, \sigma, P)$ – it returns the index of the last element below s satisfying predicate P: If there is not any such elements satisfying P, $last(s, \sigma, P)$ is undefined.

$$\begin{aligned}
&i = last(s, \sigma, P) \wedge \exists j < len(\sigma).\ j < s\ \wedge P(\sigma(j)) \\
&\Longrightarrow \\
&i < len(\sigma)\ \wedge i < s\ \wedge\ P(\sigma(i)) \wedge \\
&\forall j < s.\ i < j \longrightarrow \neg P(\sigma(j))
\end{aligned}$$

  We employ the shorthand $last(\sigma, P)$ to denote $last(len(\sigma), \sigma, P)$.

- $take(n, \sigma)$ – it takes first n elements from the sequence σ:

$$\begin{aligned}
&\sigma_1 = take(\sigma, n) \Longrightarrow \\
&len(\sigma_1) = min(len(\sigma), n)\ \wedge\ \forall i < len(\sigma_1).\ \sigma(i) = \sigma_1(i)
\end{aligned}$$

  where $min(len(\sigma), n)$ returns the minimum between $len(\sigma)$ and n.

- $\sigma_1 \prec \sigma_2$ – if it holds, σ_1 is a prefix of σ_2

$$\begin{aligned}
&\sigma_1 \prec \sigma_2 \\
&\Longleftrightarrow \\
&\forall i < len(\sigma_1).\ \sigma_1(i) = \sigma_2(i)
\end{aligned}$$

An interesting observation is that hardware signals and register values can be treated as infinite sequences.

Definition 1.3
Let S be a hardware signal over domain D, i.e. $S : \mathbb{T} \to D$. Let P be a predicate on D and $t \in \mathbb{T}$. The signal S can be considered as a sequence of its values. Thus, the value of signal at cycle t, S^t, denotes an element from D at the "position" t. We introduce a shorthand notation P^t to denote $P(S^t)$. Therefore, we can treat P as a an infinite sequence with boolean domain.

We define operator $last_{hw}(t, P) \triangleq max\{t' < t \mid P^{t'}\}$ which specifies the last cycle prior t where P held. If there is no such a cycle the result of $last_{hw}$ is undefined. Similarly, we define operator $next_{hw}(t, P) \triangleq min\{t' > t \mid P^{t'}\}$ which specifies the next cycle after t where P held.

Records

In this thesis we use record notation. Let R be a record type and $r \in R$ be a record. Let r have two fields $x \in \mathbb{B}^n$ and $y \in \mathbb{B}^n$. We use the notation $r.x$ to access field x. Similarly we access other record fields. We employ operator $[\![]\!]$ to construct a record, we introduce this notation by the following example:

$$r = \left[\!\!\left[\begin{array}{l} x = 1^n \\ y = 0^n \end{array} \right]\!\!\right]$$

We often use records to describe the state of hardware. Such a hardware state depends on hardware cycles. We model this dependency between state of hardware and cycles via functions, e.g. $f : \mathbb{T} \to R$ where function f maps hardware cycles to records. For a given hardware cycle t, f^t represents the whole record at cycle t. We access field x at cycle t as $f^t.x$.

Sometimes we need values of a particular field for all hardware cycles. In this case we employ the notation $f.x : \mathbb{T} \to \{0, 1\}^n$, where $f.x$ maps hardware cycles to values of field x.

Chapter 2

IHaVeIt

Hardware design companies often work on models described at the Register Transfer Level (RTL). An ideal formal verification technique for hardware should target these models with a high-degree of automation. Due to the details and the complexity of these models, this ideal is not yet reality.

On the one side, algorithmic techniques, such as model checking, SAT or SMT solvers, verify combinational and temporal properties automatically but they suffer from the infamous state explosion problem. To overcome that problem, many automatic abstraction techniques have been developed [ID96, VB05]. Still, these techniques apply to term-level models which are already abstractions of actual implementations. For instance, term-level models of processors only implement a subset of the instruction set architecture (ISA) and data paths, memories, processor elements, such as decoders and ALU are left uninterpreted. Moreover, there is often no obvious connection between the implementations and their abstractions, e.g. [LSB02, VB99].

On the other side, deductive methods can handle very large and detailed designs but require significant human efforts from expert users. For instance, the ACL2 [KMM00] and PVS [OSR92] theorem provers have been successfully used to verify complex out-of-order pipelined machines described at low-levels of abstraction [SH98, BJK+05].

In this chapter we present a fully automatic technique to verify large systems, which are characterized by domain-independence. Domain-independence requires that the correctness of the system does not rely on domain-specific properties, e.g. ordering of bits in a bit vector. This often holds for correctness criteria of caches and datapaths of processors.

We also present an abstraction technique which allows reduction of the transition systems. It is based on the fact that very often the exact definition of a function is irrelevant for the property to be verified, and in this case we replace such a function by an uninterpreted one. This step is a classical trade-off. On the one hand we reduce the system description by dropping some definitions. On the other hand we increase the state space of the system since the uninterpreted functions are modelled as fixed tables (or memories) which maps input arguments to some result data. To overcome the drawbacks introduced by the last step, we present a transformation technique which

allows handling of the models containing uninterpreted functions efficiently.

At the end of this chapter we present the IHaVeIt environment [Tve05a, TA08] (Isabelle Hardware Verification Infrastructure), which is based on the Isabelle/HOL theorem prover [Pau94], the model checkers NuSMV [CCG+02] and SMV [McM99], as well as different SAT solvers. This environment implements the above outlined techniques. It can also translate Isabelle/HOL code into Verilog descriptions which are then synthesized on FPGA platforms.

2.1 Preliminaries

2.1.1 Logic of Equality with Function Symbols

Equality Logic (EL) is propositional logic, augmented with terms and equality over terms. For clarity, we consider variables ranging over integer domains. This assumption does not limit the expressiveness of the logic. A term t is an integer constant, a variable, or an *if-then-else* construct (ITE). The formulae are Boolean variables, negations of formulae, conjunctions of formulae, or equalities between two terms. Let *Var*, *BVar* denote the sets of variables and Boolean variables respectively. Formally, the set of valid terms and formulae is defined as follows:

$$t^{EL} ::= \quad c \in \mathbb{N} \mid v \in Var \mid ITE(\psi, t_1^{EL}, t_2^{EL})$$

$$\psi ::= \quad b \in BVar \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \; t_1^{EL} = t_2^{EL}$$

The introduced logic is very often extended by function symbols: Equality Logic with Function symbols (ELF). Additionally to variables, one introduces function symbols and memories. Note that the latter both have similar semantics: both are mappings, and function applications are equivalent to memory read operations. The only difference is that memories can be updated.

Formally, let *MVar*, and *Fun* be the sets of memory and function names respectively. Then a term in ELF is either an EL-term, a memory, a memory read or write access, or a function application:

$$\begin{aligned} t ::= \quad & t^{EL} \mid m \in MVar \mid ITE(\psi, t_1, t_2) \mid \\ & read(t_1, t_2) \mid write(t_1, t_2, t_3) \mid \\ & f(t_1, \ldots, t_n) \qquad \text{with } f \in Fun \mid \end{aligned}$$

$$\psi ::= \quad b \in BVar \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \; t_1 = t_2$$

where the term $read(t_1, t_2)$ denotes a read from the memory term t_1 at location t_2, and the term $write(t_1, t_2, t_3)$ represents update of the memory term t_1 at location t_2 with the data t_3.

In this chapter, we employ the notation $t_1 \sqsubseteq t_2$ to express that t_1 is a sub-term/a sub-formula of the term/the formula t_2.

$$
\begin{array}{lcl}
eval(I, c) & \triangleq & c \\
eval(I, x) & \triangleq & I_v(x) \\
eval(I, m) & \triangleq & I_v(m) \\
eval(I, ITE(\psi, t_1, t_2)) & \triangleq & \text{if } eval(I, \psi) \text{ then } eval(I, t_1) \text{ else } eval(I, t_2) \\
eval(I, read(t_1, t_2)) & \triangleq & (eval(I, t_1))(eval(I, t_2)) \\
eval(I, write(t_1, t_2, t_3)) & \triangleq & (eval(I, t_1))(eval(I, t_2) := eval(I, t_3)) \\
eval(I, f(t_1, \ldots, t_n)) & \triangleq & I_f(f)(eval(I, t_1), \ldots, eval(I, t_n)) \\
 & & \\
eval(I, b) & \triangleq & I_v(b) \\
eval(I, \psi_1 \wedge \psi_2) & \triangleq & eval(I, \psi_1) \wedge eval(I, \psi_2) \\
eval(I, \neg\psi) & \triangleq & \neg\, eval(I, \psi) \\
eval(I, t_1 = t_2) & \triangleq & eval(I, t_1) = eval(I, t_2)
\end{array}
$$

Figure 2.1: Evaluation of terms and formulae.

An interpretation I is a pair $I = (I_v, I_f)$. I_v is a mapping from variable or memory names to booleans, integer values, or memories. I_f is a mapping from function names to functions.

We define function *eval* which for a given interpretation evaluates a given formula/term. A recursive definition of *eval* is given in Figure 2.1.

An interpretation I *satisfies* a formula ψ, written $I \models \psi$, if ψ evaluates to true under I, i.e. $I \models \psi \triangleq eval(I, \psi) = True$.

For a given function interpretation I_f, a formula ψ can be treated as a predicate ψ^{I_f} over the domain of the formula's variables. Formally, let $x_1, \ldots, x_n$ be all variables occurring in ψ, then the corresponding predicate is defined as follows:

$$
\begin{gathered}
\psi^{I_f}(v_1, \ldots, v_n) \triangleq \\
(I_v, I_f) \models \psi \wedge I_v(x_1) = v_1 \wedge \cdots \wedge I_v(x_n) = v_n
\end{gathered}
$$

Similarly, we use notation $\psi(I_v)$ to denote the result of substitution of all variables in ψ with values specified by I_v.

2.1.2 Kripke Structures

Transition systems are modelled as Kripke structures. A Kripke structure K over a set of atomic propositions AP is defined as a quintuple $K = (AP, S, S_0, R, L)$, where

- S is a finite set of states,
- $S_0 \subseteq S$ denotes the set of initial states,
- $R \subseteq S \times S$ is a total transition relation: $\forall s \in S . \exists s' \in S . (s, s') \in R$

- $L : S \rightarrow 2^{AP}$ is a labeling function, which maps each state to the set of propositions that hold in that state.

Definition 2.1 (Simulation)
A relation $B \subseteq S \times S'$ is a *simulation* of two Kripke structures K and K', if for all elements $(s, s') \in B$:

- $K'.AP \subseteq K.AP$
- $(K.L(s) \cap K'.AP) \subseteq K'.L(s')$
- $\forall s_1.\ (s, s_1) \in K.R \longrightarrow \exists s'_1.\ (s', s'_1) \in K'.R\ \wedge\ B(s_1, s'_1)$

We say K' *simulates* K, written as $K' \succ K$, if there is a simulation relation B, such that:

$$\forall s \in K.S_0.\ \exists s' \in K'.S_0.\ B(s, s')$$

Definition 2.2 (Bisimulation)
A relation $B \subseteq S \times S'$ is a *bisimulation* of two Kripke structures K and K' if for all elements $(s, s') \in B$:

- $K.AP = K'.AP$
- $K.L(s) = K'.L(s')$
- $\forall s_1.\ (s, s_1) \in K.R \longrightarrow \exists s'_1.\ (s', s'_1) \in K'.R\ \wedge\ B(s_1, s'_1)$
- $\forall s'_1.\ (s', s'_1) \in K'.R \longrightarrow \exists s_1.\ (s, s_1) \in K.R\ \wedge\ B(s_1, s'_1)$

Two Kripke structures K and K' are called *bisimular*, written $K \approx K'$, if a bisimulation B of K and K' exists, such that:

- $\forall s \in K.S_0.\ \exists s' \in K'.S_0.\ B(s, s')$
- $\forall s' \in K'.S_0.\ \exists s \in K.S_0.\ B(s, s')$

Context of Kripke Structures

Kripke structures can be represented in ELF by specifying a context function. The context of a Kripke structure K is a set of ELF formulae unambiguously describing K. The initial states, the transition relation, and the labeling function are considered as predicates over the state variable s and over the pairs (s, s') of the state and the next state variables. For a given function interpretation I_f, we can describe a Kripke structure by the following predicates:

- $\psi_{S_0}^{I_f}$ – holds for all initial states, i.e. $\psi_{S_0}(s) \Leftrightarrow s \in S_0$

- $\psi_R^{I_f}$ – holds for all pairs of states in the transition relation, i.e. $\psi_R(s, s_1) \Leftrightarrow (s, s_1) \in R$
- $\psi_{ap}^{I_f}$ – holds for those states which possess atomic proposition ap, i.e. $\psi_{ap}(s) \Leftrightarrow ap \in L(s)$

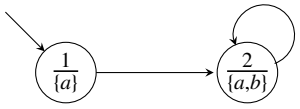
Formally, a context of a Kripke structure K parametrized by a function interpretation I_f, noted $co(K^{I_f})$, is given by the following triple:

Definition 2.3 (Kripke structure context)

$$co(K^{I_f}) \triangleq (\psi_{S_0}^{I_f}, \psi_R^{I_f}, \{\psi_{ap}^{I_f} \mid ap \in K.AP\})$$

A state s of a Kripke structure may consist of many components. We use the notation $s.x$ to denote component x in the state s. Then, the ELF formulae, which are used to describe a Kripke structure, are defined over state variables $s.x$, $s.y$ and so on. When it is clear from the context, we drop the prefix s and directly use the names x, y.

Example 2.1
Consider a Kripke structure K. It consists of two states 1 and 2 which are marked with labels a and b. State 1 is the initial state. There are two transitions: one from state 1 to state 2, and another is a loop-transition in the state 2. $co(K)$ presents the context of the Kripke structure, where s denotes the current state and s' the next state.



$$K: \quad \begin{array}{lcl} S & \equiv & \{1,2\} \\ S_0 & \equiv & \{1\} \\ R & \equiv & \{(1,2),(2,2)\} \\ AP & \equiv & \{a,b\} \\ L & \equiv & \lambda x. \begin{cases} \{a\} & : \ x = 1 \\ \{a,b\} & : \ x = 2 \end{cases} \end{array}$$

$$co(K): \quad \begin{array}{lcl} \psi_{S_0}(s) & \equiv & s = 1 \\ \psi_R(s,s') & \equiv & (s = 1 \wedge s' = 2) \vee \\ & & (s = 2 \wedge s' = 2) \\ \psi_a(s) & \equiv & s = 1 \vee s = 2 \\ \psi_b(s) & \equiv & s = 2 \end{array}$$

2.1.3 Temporal Logic

Properties of Kripke structures are described via temporal logic [CGP99]. Temporal logic is a formalism for describing sequences of transitions between states in a system, and it uses atomic propositions and boolean connectives to describe properties of

$$\begin{aligned}\psi ::= \quad & ap \in AP \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \\ & E\ \phi \mid A\ \phi\end{aligned}$$

$$\begin{aligned}\phi := \quad & \psi \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \\ & X\ \phi \mid F\ \phi \mid G\ \phi \mid \phi_1\ U\ \phi_2\end{aligned}$$

Figure 2.2: Syntax of CTL*.

states. There are many different temporal logics, and the most common ones are CTL*, ACTL*, LTL, and CTL.

In CTL* formulae describe properties of computation trees. A computation tree is computed by unwinding a Kripke structure into an infinite tree with a root from the set of initial states. CTL* formulae consist of state formulae and path formulae (Figure 2.2). A state formula ψ is either an atomic proposition, conjunction of two state formulae, negation of a formula, or a quantified path formula. In the latter case, if ϕ is a path formula, the state formula $E\ \phi$ specifies that ϕ holds along at least one path, and $A\ \phi$ specifies that ϕ holds along all paths. The operators E and A are called path quantifiers. A path formula ϕ is either a state formula, conjunction of path formulae, negation of a path formula, or application of temporal operators X, F, G, and U. CTL* is the set of all state formulae. In the following, we formally define the semantics of CTL* with respect to a Kripke structure K.

A path in K is a branch in a computation tree computed from K, i.e. it is an infinite sequence of states π such that:

$$\forall i \geq 0.\ (\pi^i, \pi^{i+1}) \in K.R$$

We write π^i to denote a sub-path of π starting at position i. We define an operator $\models$ which evaluates a given CTL* formula. We denote by $(K, s) \models \psi$ that a state formula ψ holds in the state s. Similarly, if ϕ is a path formula, $(K, \pi) \models \phi$ specifies that ϕ holds along path π. Figure 2.3 shows the definition of the operator $\models$.

Example 2.2
Let us specify several CTL* properties for the Kripke structure from Example 2.1.

1. $AF\ b$ – in all paths there is a state where b holds
2. $AG(b \longrightarrow X\ a)$ – in all paths at every state if b holds, there is a successor state where a holds, where $(b \longrightarrow X\ a)$ is a shorthand for $\neg(b \wedge \neg(X\ a))$.
3. b – in the initial state b holds

The first two properties hold for the Kripke structure, and the last one, obviously, does not hold.

$$
\begin{array}{lcll}
(K,s) \models ap & \triangleq & ap \in L(s) & \\
(K,s) \models \psi_1 \wedge \psi_2 & \triangleq & ((K,s) \models \psi_1) \wedge ((K,s) \models \psi_2) & \\
(K,s) \models \neg\psi & \triangleq & \neg((K,s) \models \psi) & \\
(K,s) \models E\ \phi & \triangleq & \exists\pi.\ \pi(0) = s\ \wedge (K,\pi) \models \phi & \text{exists} \\
(K,s) \models A\ \phi & \triangleq & \forall\pi.\ \pi(0) = s\ \wedge (K,\pi) \models \phi & \text{for all} \\
 & & & \\
(K,\pi) \models \psi & \triangleq & (K,\pi(0)) \models \psi & \\
(K,\pi) \models \phi_1 \wedge \phi_2 & \triangleq & ((K,\pi) \models \phi_1) \wedge ((K,\pi) \models \phi_2) & \\
(K,\pi) \models \neg\phi & \triangleq & \neg((K,\pi) \models \phi) & \\
(K,\pi) \models X\ \phi & \triangleq & (K,\pi^1) \models \phi & \text{next} \\
(K,\pi) \models F\ \phi & \triangleq & \exists i.\ (K,\pi^i) \models \phi & \text{finally or eventually} \\
(K,\pi) \models G\ \phi & \triangleq & \forall i.\ (K,\pi^i) \models \phi & \text{globally} \\
(K,\pi) \models \phi_1\ U\ \phi_2 & \triangleq & \exists i.\ (K,\pi^i) \models \phi_2 \wedge & \\
 & & \quad \forall j \in [0 : i[.\ (K,\pi^j) \models \phi_1 & \text{until}
\end{array}
$$

Figure 2.3: Evaluation of CTL* formulae.

ACTL* is a subset of CTL* without E path quantifier.

There are two widely used subsets of CTL*: CTL (computation-tree logic) and LTL (linear-time logic). In CTL, every temporal operator must be immediately preceded by a path quantifier. LTL consists of formulae of the form $A\ f$, where f can only contain temporal operators, atomic propositions, and boolean connectives.

It is well-known that simulation preserves ACTL* properties and that bisimulation preserves CTL* properties [CGP99].

2.2 Data Abstraction

Suppose a model description is given by a transition system, where the transition relation is defined by a predicate over the current and the next state. This predicate is described by means of equations, and does not make use of any domain specific operations, such as integer addition. Such a transition system will only input, output, propagate and compare values of variables with each other and hence all computations are *domain independent*. For example consider the following transition system $\psi_R(s,s') \equiv s'.x = ITE(s.a = s.b, s.x, s.y)$. In this case the value of the component $s.y$ changes arbitrary. Assume we want to verify the temporal property $G(s.x = s.y)$, which states that $(s.x = s.y)$ holds in each step during a run.

For this particular example assigning a domain of size three to x and y and a domain of size two to a and b is sufficient for proving or disproving the temporal property. This is the basic idea of our data abstraction algorithm, which we prove to

be sound and complete, i.e. a temporal property holds on the transition system with reduced domains, if and only if it holds on the original one. Its correctness relies on the following basic observations:

- during a run the value of x either stays the same or it takes the value of y
- the values of variables x, y are neither compared, nor assigned to the variables a or b
- the flow of data does neither depend on particular values of x and y, nor on those of a and b, only the values of occurring equalities are important
- to maintain the values of the equalities appearing in the transition predicate and in the property during a run, it suffices only to consider some finite domains for the occurring variables.

In this section we present several variations of so-called small model property, which specify a data abstraction for EL-formulae. Then, we show how this can be applied for Kripke structures which are described in EL. We also prove that abstracted and original Kripke structures are in is a bisimulation relation.

2.2.1 Small Model Property

There is a well-known definition of the small model property [PRSS02].

Definition 2.4 (Small model property)
A formula in equality logic enjoys the small model property if the formula is satisfiable/valid if and only if it is satisfiable/valid over a finite domain.

In order to utilize the mentioned property we need an algorithm which computes the finite domains of the variables, such that their sizes preserve the formula satisfiability. In general, the goal is to find a domain allocation function which maps the variable names to the finite domains, which preserve satisfiability. There is a well-known folklore theorem that specifies such a domain allocation function.

Theorem 2.1 (Folklore)
To preserve satisfiability of a formula, it is enough to give every non-boolean variable in the formula a domain of size n, where n is the number of non-boolean variables.

The proof of this theorem is quite trivial and can be found, e.g. in [PRSS02]. It is obvious that an algorithm implementing this theorem results in the state space n^n. A simple optimization is to consider equalities of the variables in the formula. The following definitions simplify the introduction of the optimized algorithm.

Definition 2.5 (Related nodes)
The function *rnodes* recursively computes a set of terms which are compared (or related) with each other in a given term/formula. Note that in this definition we are interested in related nodes of a given term/formula, all subterms/subformulas are

considered in the next definition.

$$\begin{aligned} rnodes(t_1 = t_2) &\triangleq rnodes(t_1) \cup rnodes(t_2) \\ rnodes(ITE(\psi, t_1, t_2)) &\triangleq rnodes(t_1) \cup rnodes(t_2) \\ rnodes(v) &\triangleq \{v\} \\ rnodes(c) &\triangleq \{c\} \end{aligned}$$

Definition 2.6 (Related relation)
For a given formula ψ we define a *related* relation on subformulas as:

$$R_\psi \triangleq \{(e_1, e_2) \mid \exists e \sqsubseteq \psi.\ \{e_1, e_2\} \subseteq rnodes(e)\}$$

The transitive closure of R_ψ we denote by E_ψ.

It is easy to show that E_ψ is an equivalence relation.

We use the notation $[\![e]\!]_{E_\psi}$ to represent the equivalence class containing node e. We denote by $e_1 \sim_{E_\psi} e_2$ the fact that two nodes e_1 and e_2 are in the same equivalence class.

Algorithm 2.1 (Folkore+)
A naive optimisation of the algorithm induced by the folkore theorem works as follows:

1. Compute the equivalence relation E_ψ of formula ψ.
2. Assign to each variable x a new abstract domain of the size which equals the cardinality of the corresponding equivalence class: $\mid [\![x]\!]_{E_\psi} \mid$.
3. Replace every constant occurring in the formula by a value from the new abstracted domain, according to the following rule: two originally different constants are replaced by two different abstract values. For example, let c_1 and c_2 be two constants, and let c'_1 and c'_2 be two new abstract values. c_1 and c_2 are replaced as follows: $c_1 \sim_{E_\psi} c_2 \longrightarrow (c_1 = c_2 \Leftrightarrow c'_1 = c'_2)$.

This algorithm for a given EL formula ψ computes its abstracted version ψ^{abs}. The abstracted formula ψ^{abs} differs from ψ only in constants and in domains of variables. The following example illustrates the algorithm.

Example 2.3
Consider formula $\psi \equiv z = ITE(c = a, x, y)$. Initially, the domains of all variables are infinite, e.g. $c \in \mathbb{Z}$, $a \in \mathbb{Z}$, $x \in \mathbb{N}$, $y \in \mathbb{N}$, and $z \in \mathbb{N}$. We see that variable a and constant c are not related to the variables $\{x, y, z\}$. The algorithm splits all variables and constants into two equivalence classes $\{a, c\}$ and $\{x, y, z\}$. It also assigns to variables a a domain of size two, constant c receives a value from the new domain of a, and other variables receive a domain of size three. Formally, $a' \in \{1, 2\}$, $c' = 1$, $x' \in \{3, 4, 5\}$, $y' \in \{3, 4, 5\}$, and $z' \in \{3, 4, 5\}$. The abstracted formula ψ^{abs} has the form: $\psi^{abs} \equiv z' = ITE(c' = a', x', y')$.

In the worst case, when all variables are compared with each other, the reduced state space has still size of n^n. There are other sophisticated reductions [Str02, PRSS02, BV00]. However, these algorithms work only for formulae and cannot be extended for the abstraction of Kripke structures due to the next-state computation.

Next, we note an interesting fact on how the values from the original and the reduced domains can be related.

Definition 2.7
Given an EL formula ψ, its abstraction ψ^{abs}, an original and an abstracted interpretation s and s_{abs} respectively. We define a mapping between the values of the interpretations as follows: whenever two variables are compared with each other in the formula, we require their abstracted values to be the same if and only if their original values are. The same must hold for comparisons between the variable values and the constants.

$$
\begin{aligned}
&A_{E_\psi}(s, s_{abs}) \triangleq \\
&\forall x\ y\ c. \\
&\quad s.x \sim_{E_\psi} s.y \longrightarrow (s_{abs}.x = s_{abs}.y \Leftrightarrow s.x = s.y) \wedge \\
&\quad s.x \sim_{E_\psi} c \longrightarrow (s_{abs}.x = c_{abs} \Leftrightarrow s.x = c)
\end{aligned}
$$

Given such a relation, we can establish the following fact:

Lemma 2.2
For any EL formula ψ, formula abstraction ψ^{abs} and two interpretations s and s_{abs} the following holds:

$$A_{E_\psi}(s, s_{abs}) \longrightarrow \psi(s) \Leftrightarrow \psi^{abs}(s_{abs})$$

Proof. *Recall that formulae ψ and ψ^{abs} are equivalent except the domains of the variable and the values of the constants. These differences can only affect values of the equalities in the formulae. Thus, our goal ($\psi(s) \Leftrightarrow \psi^{abs}(s_{abs})$) holds, if the values of the equalities in ψ match the values of the corresponding equalities in ψ^{abs} and vise versa. The definition of A_{E_ψ} guarantees that if two variables are compared, either directly or transitively, then their values match in the original formula if and only if their values match in the abstracted formula. This is enough to preserve the values of all equalities.*

We define an extended version of relation A for formulae which depend on two interpretations:

Definition 2.8

$$
\begin{aligned}
&A^{ex}_{E_\psi}((s, s_{abs}), (s', s'_{abs})) \triangleq \\
&\quad A_{E_\psi}(s, s_{abs}) \ \wedge A_{E_\psi}(s', s'_{abs}) \wedge \\
&\quad \forall x\ y.\ s.x \sim_{E_\psi} s'.y \longrightarrow (s_{abs}.x = s'_{abs}.y \Leftrightarrow s.x = s'.y)
\end{aligned}
$$

The following lemma is a reinterpretation of Lemma 2.2:

Lemma 2.3
For any EL formula ψ, formula abstraction ψ^{abs} and four interpretations s, s', s_{abs}, and s'_{abs} the following holds:

$$A^{ex}_{E_\psi}((s, s_{abs}), (s', s'_{abs})) \longrightarrow (\psi(s, s') \Leftrightarrow \psi^{abs}(s_{abs}, s'_{abs}))$$

2.2.2 Data Abstraction for Kripke structures

We have just presented how the domains of the variables, which occur in a *formula*, can be reduced. In this section we present an algorithm which, for a given Kripke structure, reduces the domains of all state variables by exploiting the small model property. At the moment we assume that a given Kripke structure K contains no function symbols, i.e. it has neither memories nor uninterpreted functions.

The basic idea for the domain reduction of a Kripke structure is to analyze the context of the Kripke structure and to exploit the small model property. For a given Kripke structure K the algorithm can be straightforward described as follows:

Algorithm 2.2

1. Compute the context $co(K)$ of K.
2. Add all pairs of state variables $s.x$ and the corresponding next state variables $s'.x$ to the same equivalence class.
3. Compute the equivalence relation E_ψ for each context formula ψ.
4. The equivalence classes of different formulas may intersect. We merge these intersecting equivalence classes. The result is again an equivalence relation. We denote it by E_K.
5. Assign to each variable $s.x$ a new abstract domain of the size of the cardinality of the corresponding equivalence class: $\mid [\![x]\!]_{E_K} \mid$.
6. Replace every constant, which occurs in the context, by a value from the new abstracted domain. Two originally different constants are replaced by two different abstract values: $\forall c_1.\, \forall c_2.\, c_1 \sim_{E_K} c_2 \longrightarrow (c_1 = c_2 \Leftrightarrow c'_1 = c'_2)$.
7. Construct the abstracted Kripke structure K^{abs} from the updated context.

In the worst case this algorithm computes a Kripke structure with the state space $(2 * n)^{2*n}$, where $2 * n$ is the number of state and the next state variables.

We want to show that a Kripke structure K and its abstracted version K^{abs} are in a bisimulation. Basically, we have to show that the truth values of all equalities in the description of K and K^{abs} have the same behavior. Thus, we need a suitable bisimulation relation which utilizes this fact, e.g. it should guarantee that if two state

variables are in the same equivalence class, their abstracted values during a run must be equal if and only if their original values are.

We define such a relation on the base of the relation A (Definition 2.7). Note, the formulae in the contexts of a Kripke structure range over state and next state variables. Therefore, A_{E_K} would relate the original transitions with abstracted transitions, i.e. A_{E_K} relates concrete pairs of states (s, s') with abstracted ones (s_{abs}, s'_{abs}). We derive definition of A_{E_K} from the definition of $A^{ex}_{E_\psi}$:

Definition 2.9

$$
\begin{aligned}
&A_{E_K}((s, s'), (s_{abs}, s'_{abs})) \triangleq \\
&\forall x\ y\ c. \\
&\quad s.x \sim_{E_K} s.y \longrightarrow (s_{abs}.x = s_{abs}.y \Leftrightarrow s.x = s.y) \wedge \\
&\quad s.x \sim_{E_K} c \longrightarrow (s_{abs}.x = c_{abs} \Leftrightarrow s.x = c) \wedge \\
&\quad s'.x \sim_{E_K} s'.y \longrightarrow (s'_{abs}.x = s'_{abs}.y \Leftrightarrow s'.x = s'.y) \wedge \\
&\quad s'.x \sim_{E_K} c \longrightarrow (s'_{abs}.x = c_{abs} \Leftrightarrow s'.x = c) \wedge \\
&\quad s.x \sim_{E_K} s'.y \longrightarrow (s_{abs}.x = s'_{abs}.y \Leftrightarrow s.x = s'.y)
\end{aligned}
$$

For Kripke structures the relation A_{E_K} is, in some sort, symmetric in the structure. It consists of three kinds of equalities:

1. equalities including only the current state variables
2. equalities including only the next state variables
3. equalities including the state variables and the next state variables

The first and the second groups contain identical equalities, but in the first group the rules are stated about to the current state variables and the second group about the next state variables. This is because the state and the next state variables are added pair-wise to the same equivalence class (Algorithm 2.2).

We introduce two additional predicates which relate original and abstract states of a Kripke structure:

Definition 2.10

$$
\begin{aligned}
&B_{E_K}(s, s_{abs}) \triangleq \\
&\forall x\ y\ c. \\
&\quad s.x \sim_{E_K} s.y \longrightarrow (s_{abs}.x = s_{abs}.y \Leftrightarrow s.x = s.y) \wedge \\
&\quad s.x \sim_{E_K} c \longrightarrow (s_{abs}.x = c_{abs} \Leftrightarrow s.x = c)
\end{aligned}
$$

$$
\begin{aligned}
&B^{aux}_{E_K}((s, s'), (s_{abs}, s'_{abs})) \triangleq \\
&\forall x\ y.\ s.x \sim_{E_K} s'.y \longrightarrow (s_{abs}.x = s'_{abs}.y \Leftrightarrow s.x = s'.y)
\end{aligned}
$$

We can prove that A_{E_k} can be represented in terms of B_{E_K} and $B^{aux}_{E_K}$:

Lemma 2.4

$$A_{E_k}((s, s'), (s_{abs}, s'_{abs})) =$$
$$B_{E_K}(s, s_{abs}) \wedge B^{aux}_{E_K}((s, s'), (s_{abs}, s'_{abs})) \wedge B_{E_K}(s', s'_{abs})$$

The following three lemmata define how A and B relations for Kripke structures are related with A relation for formulae. Proofs for these lemamata are based on definitions of relations A and B, Lemma 2.2, and Lemma 2.3.

Lemma 2.5
Let K be a Kripke structure. Let s and s_{abs} be an original state and the corresponding abstract state. If these states are in relation w.r.t. B_{E_K}, they are in relation w.r.t. A for initial predicate ψ_{S_0}.

$$B_{E_K}(s, s_{abs}) \longrightarrow A_{E_{\psi_{S_0}}}(s, s_{abs})$$

Lemma 2.6
Let K be a Kripke structure. Let s and s_{abs} be an original state and the corresponding abstract state. If these states are in relation w.r.t. B_{E_K}, they are in relation w.r.t. A for every atomic proposition formula ψ_{ap}.

$$B_{E_K}(s, s_{abs}) \longrightarrow \forall ap \in K.AP.\ A_{E_{\psi_{ap}}}(s, s_{abs})$$

Lemma 2.7
Let K be a Kripke structure. Let s and s_{abs} be be an original state and the corresponding abstract state. Let s' be a successor state of s and s'_{abs} be a successor state of s_{abs}. If these states are in relation w.r.t. B_{E_K}, they are in relation w.r.t. A^{ex} for transition predicate ψ_R.

$$A_{E_K}((s, s'), (s_{abs}, s'_{abs})) \longrightarrow A^{ex}_{E_{\psi_R}}((s, s_{abs}), (s', s'_{abs}))$$

Recall that a bisimulation relation for two Kripke structures requires a relation between states (see Definition 2.2). Therefore, we select B_{E_K} as the simulation relation.

Lemma 2.8
B_{E_K} is a bisimulation.

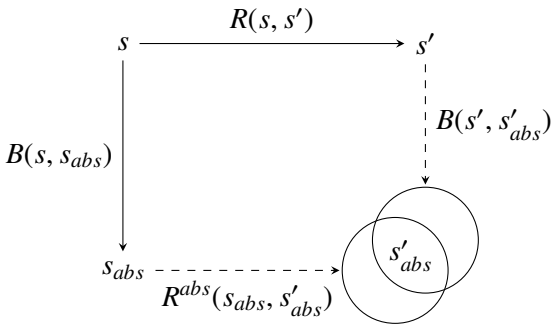


Figure 2.4: A part of bisimulation relation.

Proof. *We have to show three goals:*

1. $B_{E_K}(s, s_{abs}) \longrightarrow \forall ap \in K.AP.\, \psi_{ap}(s) \Leftrightarrow \psi^{abs}_{ap}(s_{abs})$
2. $B_{E_K}(s, s_{abs}) \longrightarrow (\forall s'.\, \psi_R(s, s') \longrightarrow \exists s'_{abs}.\, \psi^{abs}_R(s_{abs}, s'_{abs}) \wedge B_{E_K}(s', s'_{abs}))$
3. $B_{E_K}(s, s_{abs}) \longrightarrow (\forall s'_{abs}.\, \psi^{abs}_R(s_{abs}, s'_{abs}) \longrightarrow \exists s'.\, \psi_R(s, s') \wedge B_{E_K}(s', s'_{abs}))$

Case 1: *This goal is proven by the usage of Lemma 2.6 and Lemma 2.2.*

Case 2: *To prove this goal we have to find a state s'_{abs} such that there exists a transition from s_{abs} to s'_{abs} and s'_{abs} is in the bisimulation with s'. In other words having $B_{E_K}(s, s_{abs})$ and $\psi_R(s, s')$ we have to prove $\exists s'_{abs}.\, \psi^{abs}_R(s_{abs}, s'_{abs}) \wedge B_{E_K}(s', s'_{abs})$ (Figure 2.4). To prove this existence it is enough to find a s'_{abs} such that $A_{E_K}((s, s')(s_{abs}, s'_{abs}))$ holds. This is because, A_{E_K} implies $B_{E_K}(s', s'_{abs})$ (Lemma 2.4) and the existence of the transition in the abstracted model (Lemma 2.7 and Lemma 2.3). Thus, our goal is*

$$\exists s'_{abs}.\, A_{E_K}((s, s')(s_{abs}, s'_{abs}))$$

We rewrite it by the application of Lemma 2.4

$$\exists s'_{abs}.\, B_{E_K}(s, s_{abs}) \wedge B^{aux}_{E_K}((s, s'), (s_{abs}, s'_{abs})) \wedge B_{E_K}(s', s'_{abs})$$

In the assumptions we already have $B_{E_K}(s, s_{abs})$ and hence the new goal is:

$$\exists s'_{abs}.\, B^{aux}_{E_K}((s, s'), (s_{abs}, s'_{abs})) \wedge B_{E_K}(s', s'_{abs})$$

Thus, our goal is to find suitable values for variables from the next-state s'_{abs}. $B^{aux}_{E_K}$ and B_{E_K} are defined for the variables from the same equivalence class. Thus, we can select values for the variable from the abstract next-state s'_{abs} based on the equivalence classes of these variables.

The set of all values for a variable from s'_{abs} is defined by its abstract domain D_{abs}. The size of this domain is defined as the size of the corresponding equivalence

class (Algorithm 2.2, step 5). For example, let us consider some state variable $s.x$ and its equivalence class $[\![s.x]\!]_{E_K}$. Let the size of the equivalence class be n. Since we add every pair of current and next-state variables into the same equivalence class (Algorithm 2.2, step 2), this class contains $\frac{n-k}{2}$ state variables and $\frac{n-k}{2}$ next-state variables, where k is the number of constants in the class.

Now we define an algorithm which assigns suitable abstract values to variables from $[\![s.x]\!]_{E_K}$. Let P be the set of variable names from the considered equivalence class. Recall that this set contains $\frac{n-k}{2}$ names. Let freeval be the set of non-used abstract values, i.e. they are used neither in the current state s_{abs} nor as values of the abstracted constants:

$$freeval \triangleq \{v \mid v \in D_{abs} \wedge \forall x.\ s_{abs}.x \neq v \wedge\ \forall c_{abs}.\ v \neq c_{abs}\}.$$

We define function evid which constructs the sought abstract state. Function evid is defined by recursion on the number of elements in P and takes three arguments: (i) the set of variable names to be processed, (ii) the set of non-used abstract values, and (iii) sp – state placeholder; sp has the the type of Kripke structure state and initially all variables of the placeholder have undefined values. With every recursion step one variables is processed, i.e. it receives an abstract value.

We use the shorthand $\{x\} \cup P$ to denote that a non-empty set can be represented as a union of a singleton set and the rest of the set. The recursion stops when there are no more variable names left.

$$s'_{abs} \triangleq evid(P, freeval, sp)$$

$$evid(\{\}, freeval, sp) \triangleq sp$$
$$evid(\{x\} \cup P, freeval, sp) \triangleq$$
$$\text{Case 1: } \exists y.\ s'.x = s.y\ \wedge s'.x \sim_{E_K} s.y$$
$$\quad \textbf{let } sp.x := s_{abs}.y$$
$$\quad \textbf{in } evid(P, freeval, sp)$$
$$\text{Case 2: } \exists c.\ s'.x = c\ \wedge s'.x \sim_{E_K} c$$
$$\quad \textbf{let } sp.x := c_{abs}$$
$$\quad \textbf{in } evid(P, freeval, sp)$$
$$\text{Case 3: } \exists z.\ s'.x = s'.z\ \wedge s'.x \sim_{E_K} s'.z \wedge z \notin (\{x\} \cup P)$$
$$\quad \textbf{let } sp.x := sp.z$$
$$\quad \textbf{in } evid(P, freeval, sp)$$
$$\text{Case 4: else}$$
$$\quad \textbf{let } val := Some\ val.\ val \in freeval$$
$$\quad\quad sp.x := val$$
$$\quad\quad freeval' := freeval \setminus \{val\}$$
$$\quad \textbf{in } evid(P, freeval', sp)$$

In every recursion step of evid we distinguish four cases, which are processed in the given order. In the first case we test whether the transition in the original model makes the next-state variable $s'.x$ equivalent to some current-state variable $s.y$. If this is the case, we assign to the variable in the abstract next-state $s'_{abs}.x$ the value of the current one, i.e. the value of $s_{abs}.y$. This is required by $B^{aux}_{E_K}$.

In the second case we test whether $s'.x$ equals some constant. If this is the case, we assign the value of the corresponding abstract constant to $s'_{abs}.x$. This is required by B_{E_K}.

In the third case we know that variable $s'.x$ has a distinct value from all variables in the current state s. However, it can have an equivalent value with a variable in the next state s'. If there exists such a variable and we have already processed it ($z \notin (\{x\} \cup P)$), assign the corresponding value.

In the last step variable $s'.x$ has a distinct value from all variables in the current state. Moreover, it has a distinct value from all variables, which are processed so far, in the next state. Thus, we have to assign to $s'_{abs}.x$ a non-used value. These four cases guarantee the conditions required by B_{E_K}.

Note that this algorithm assumes that it can always pick up a non-used value from $freeval$. This holds because the algorithm makes $\frac{n-k}{2}$ iterations and in every iteration it can pick up one non-used values. Therefore, the domain of size $\frac{n-k}{2}$ satisfies this requirement.

The values for variables from other equivalence classes are computed by the iterative application of evid to every equivalence class.

***Case 3:** This proof is easier because the original domains, from which we have to select values during construction of s'_{abs}, are infinite. The proof argumentation is similar to the proof for case* (2). □

Theorem 2.9

$$K^{abs} \approx K$$

Proof. *We choose B_{E_K} as the bisimulation. Thus, we have to show:*

- $\forall s.\, \psi_{S_0}(s) \longrightarrow \exists s_{abs}.\, \psi^{abs}_{S_0}(s_{abs}) \wedge B_{E_K}(s, s_{abs})$
- $\forall s_{abs}.\, \psi^{abs}_{S_0}(s_{abs}) \longrightarrow \exists s.\, \psi_{S_0}(s) \wedge B_{E_K}(s, s_{abs})$

Proofs for these two goals repeat the argumentation of proofs of the previous lemma for cases two and three respectively. □

2.3 Function Symbols

In this section we present several techniques for rewriting of the operations over memory terms, i.e. rewriting of ELF-terms. Then, we introduce an algorithm which eliminates applications of function symbols and memories in Kripke structures.

2.3.1 Rewriting Algorithms

The following definitions simplify the introduction of the rewriting algorithms. We extend the function *rnodes* with memories and memory operations.

Definition 2.11 (Related nodes+)
Function *rnodes* computes set of terms (nodes) which are compared with each other. For a read operation such a node is the read-term itself. For a write operation we add into the result set the term which represents the updated memory (i.e. the write-term itself) and the original memory.

$$\begin{aligned}
rnodes(t_1 = t_2) &\triangleq rnodes(t_1) \cup rnodes(t_2) \\
rnodes(ITE(\psi, t_1, t_2)) &\triangleq rnodes(t_1) \cup rnodes(t_2) \\
rnodes(v) &\triangleq \{v\} \\
rnodes(c) &\triangleq \{c\} \\
rnodes(read(t_1, t_2)) &\triangleq \{read(t_1, t_2)\} \\
rnodes(write(t_1, t_2, t_3)) &\triangleq rnodes(t_1) \cup \{write(t_1, t_2, t_3)\} \\
rnodes(m) &\triangleq \{m\}
\end{aligned}$$

We define the set of all address terms which are used in the read operations from a given memory m in a given formula f.

Definition 2.12 (Used read addresses)

$$\begin{aligned}
RAdd_f(m) \triangleq \{a \mid \quad \exists mt.\, \exists at. \quad & read(mt, at) \sqsubseteq f \wedge \\
& [\![m]\!]_{E_f} \cap rnodes(mt) \neq \emptyset \wedge a \in rnodes(at)\}
\end{aligned}$$

Similarly, we define the set of all address terms which are used in the write operations to a given memory m in a given formula f.

Definition 2.13 (Used write addresses)

$$\begin{aligned}
WAdd_f(m) \triangleq \{a \mid \quad \exists mt.\, \exists at.\, \exists dt. \quad & write(mt, at, dt) \sqsubseteq f \wedge \\
& [\![m]\!]_{E_f} \cap rnodes(mt) \neq \emptyset \wedge a \in rnodes(at)\}
\end{aligned}$$

We define the set of all used address terms as the union of the read and the write addresses.

Definition 2.14 (Used addresses)

$$Add_f(m) \triangleq RAdd_f(m) \cup WAdd_f(m)$$

Memory equality in formulae

The semantics of memory equality is a test whether two memories have equal values in all their cells. An interesting fact is that one can replace a memory equality by a finite set of equalities of memory cells, such that the truth value of the original equality is preserved. The selection of these "representative" memory addresses (or cells) can be deduced from the formula. It includes all used addresses and one "fresh" non-constrained address term per memory equality. This term represents an arbitrary access to the rest of the memory. The need for the fresh terms can be illustrated by the following example. Let us compare two memory terms: $write(mt_1, at, dt) = write(mt_2, at, dt)$. If we only test whether these memory terms have the equal values in at, we will miss the initial values of mt_1 and mt_2. Therefore, we have to consider at least one another address which differs from at. We rewrite the original formula with the preservation of its truth value as follows:

$$write(mt_1, at, dt) = write(mt_2, at, dt) \equiv$$

$$write(mt_1, at, dt)(at) = write(mt_2, at, dt)(at) \wedge$$
$$write(mt_1, at, dt)(at_1) = write(mt_2, at, dt)(at_1)$$

where at_1 is a fresh variable (or address term), which is not used in the formula. In the following we define the rewriting algorithm formally.

We introduce a set of fresh variables $EqAdd_f(m)$. The number of these variables is defined via the size of the equivalence class $[\![m]\!]_{E_f}$. This set has two properties:

- $\mid EqAdd_f(m) \mid = \mid [\![m]\!]_{E_f} \mid -1$
- $\forall at \in EqAdd_f(m).\ at \notin f$

We add the set of the fresh variables to the set of used addresses for a memory m in a formula f:

$$AddSet_f(m) \triangleq Add_f(m) \cup EqAdd_f(m)$$

Lemma 2.10
The function AddSet computes the same set for all memories from the same equivalence class:

$$\forall m_1.\forall m_2.\ m_1 \sim_{E_f} m_2 \longrightarrow AddSet_f(m_1) = AddSet_f(m_2)$$

Algorithm 2.3
The rewriting algorithm replaces every equality e of the form $mt_1 = mt_2$ in a given formula f as follows:

- Compute the set of related nodes for the equality: $rset = rnodes(e)$
- Compute the set of used address terms. It is enough to consider any element m from $rset$: $adset = AddSet_f(m)$

- Replace the memory equality by the conjunction of the memory read operations:

$$\bigwedge_{at \in adset} read(mt_1, at) = read(mt_2, at)$$

We denote the rewritten formula ψ by ψ^{rw_1}. The algorithm 2.3 preserves the satisfiability.

Lemma 2.11
$(\exists J.\ \models \psi) \Leftrightarrow (\exists I.\ I \models \psi^{rw_1})$

To prove this lemma we define an auxiliary lemma (Lemma 2.13) which is a custom version of Lemma 2.11. First we introduce an additional notation: Let I and J be two interpretations for two disjoint sets of variables S_I and S_J respectively. We use the notation $I \cup J$ to denote the interpretation for variables from the union of the disjoint sets. Formally, $(I \cup J)(v) \triangleq$ if $v \in S_I$ then $I(v)$ else $J(v)$.

We prove a lemma which states the fact that we can always find an interpretation for fresh variables which preserves the values of memory equalities.

Lemma 2.12
Let $\{mt_0, \ldots, mt_n\}$ represent all memories from an arbitrary equivalence class of a formula ψ. Let $J = (J_v, J_f)$ be an interpretation. Let ψ^{rw_1} be the rewritten formula. Let $\{at_0, \ldots, at_{n-1}\}$ be the set of freshly introduced variables. We show that there exists an interpretation I_a for fresh variables $\{at_0, \ldots, at_{n-1}\}$ such that memory equalities in ψ and ψ^{rw_1} have the same values:

$$\exists I_a.\ \forall i, j \le n.$$
$$eval(J, mt_i = mt_j) \Leftrightarrow \bigwedge_{k<n} eval((J_v \cup I_a, J_f), read(mt_i, at_k) = read(mt_j, at_k))$$

Proof. *We prove Lemma 2.12 by induction on the number of memory equalities.*

In the induction base we have one memory equality, and hence, two memory terms mt_0 and mt_1 and one fresh variable at_0 (see Algorithm 2.3, second step). If these memory terms are equal we select such an I_a which maps at_0 to an arbitrary value. Otherwise, we select such an I_a which maps at_0 to a value which addresses a cell with different content, i.e. $eval(mt_0)(I_a(at_0)) \neq eval(mt_1)(I_a(at_0))$.

Now we make an induction step $n \mapsto n + 1$. By the induction hypothesis we know that there exists I_a such that for fresh variables $\{at_0, \ldots, at_{n-1}\}$ and memory terms $\{mt_0, \ldots, mt_n\}$ the goal holds, i.e.

$$\forall i \le n,\ j \le n.$$
$$eval(J, mt_i = mt_j) \Leftrightarrow \bigwedge_{k<n} eval((J_v \cup I_a, J_f), read(mt_i, at_k) = read(mt_j, at_k))$$

In the induction step we add a fresh variable at_n and a new memory term mt_{n+1}. Let I'_a be interpretation for the fresh variable at_n. Thus our goal is:

$$\exists I_a,\ I'_a.\ \forall i \le n+1,\ j \le n+1.$$
$$eval(J, mt_i = mt_j) \Leftrightarrow \bigwedge_{k<n} eval((J_v \cup I_a \cup I'_a, J_f), read(mt_i, at_k) = read(mt_j, at_k))$$

If mt_{n+1} is equal to any of mt_i where $i \leq n$, the proof is trivial because we can substitute mt_{n+1} by an equal memory and select for at_n any interpretation.

Now we consider the case mt_{n+1} is not equal to any of the mt_i. We instantiate I_a from the induction hypothesis in the induction step, and thus, we have to find a suitable interpretation for only at_n:

$$\exists I'_a.\ \forall i \leq n+1,\ j \leq n+1.$$
$$eval(J, mt_i = mt_j) \Leftrightarrow \bigwedge_{k \leq n} eval((J_v \cup I_a \cup I'_a, J_f), read(mt_i, at_k) = read(mt_j, at_k))$$

Now we apply the induction hypothesis, apply the fact that all memories are not equal, and drop the case where $i = j$ to rewrite the goal as follows:

$$\exists I'_a.\ \forall i \leq n.\ \neg \bigwedge_{k \leq n} eval((J_v \cup I_a \cup I'_a, J_f), read(mt_{n+1}, at_k) = read(mt_i, at_k))$$

Case 1: *Let us consider the case where for all at_k (where $k < n$) mt_{n+1} on these addresses is equal to some mt_i, i.e.*

$$\exists i \leq n.\ \forall k < n.\ eval((J_v \cup I_a \cup I'_a, J_f), read(mt_{n+1}, at_k) = read(mt_i, at_k))$$

In this case memories mt_{n+1} and mt_i must differ in at least one another cell because mt_{n+1} is not equal to any m_i. Thus, we select such an interpretation I'_a which maps fresh variable at_n to the address of this cell.

Case 2: *Here we consider the negation of the previous case:*

$$\forall i \leq n.\ \exists k < n.\ eval((J_v \cup I_a \cup I'_a, J_f), read(mt_{n+1}, at_k) \neq read(mt_i, at_k))$$

In this case for all i there exists at least one at_k which makes mt_{n+1} different from all other mt_i. Thus, our goal holds and we can select an arbitrary interpretation for I'_a.□

Lemma 2.13
Given an interpretation $I = (I_v, I_f)$ of a formula ψ. Let ψ^{rw_1} be the rewritten formula. Let I_a be an interpretation for all freshly introduced variables which preserves the value of all memory equalities (see previous lemma). Then, the following holds:

$$(I_v, I_f) \models \psi \Leftrightarrow (I_v \cup I_a, I_f) \models \psi^{rw_1}$$

This rewriting algorithm does not depend on the formula structure but only on the memory equalities. Thus, we can apply this rewriting on a set of formulae, where all memory equalities are rewritten simultaneously. Lemma 2.13 holds for every rewritten formula.

Memory equality in Kripke structures

The rewriting of a Kripke structure K consists of several steps:

- Compute the context of K
- Apply the memory rewriting to the context
- Translate the rewritten context to the Kripke structure K^{rw_1}

To show that the rewritten Kripke structure K^{rw_1} simulates the original K, we have to find a simulation relation (Definition 2.1). A suitable simulation relation B^{rw_1} consists of two parts: the relation over the original state variables and the fresh address variables. We relate the original state variables via the identity mapping. The fresh variables do not exist in the original Kripke structure K. They are only present in K^{rw_1}. Therefore, we can select their values according to interpretation I_a which preserves values of all memory equalities (see the proof for Lemma 2.11 for details).

Definition 2.15
Let P be the set of fresh variable names in K^{rw_1}.

$$B^{rw_1}(s, s^{rw_1}) \triangleq \forall x \notin P.\ s^{rw_1}.x = s.x\ \wedge \forall x \in P.\ s^{rw_1}.x = I_a(x)$$

Lemma 2.14
B^{rw_1} is a simulation relation.

To prove this lemma we have to show two goals (Definition 2.1):

1. $\forall ap \in K.AP.\ B^{rw_1}(s, s_{rw_1}) \wedge \psi_{ap}(s) \longrightarrow \psi_{ap}^{rw_1}(s_{rw_1})$
2. $B^{rw_1}(s, s_{rw_1}) \wedge \psi_R(s, s') \longrightarrow \exists {s'}_{rw_1}.\ \psi_R^{rw_1}(s_{rw_1}, {s'}_{rw_1}) \wedge B^{rw_1}(s_1, {s'}_{rw_1})$

Proofs for both goals are based on Lemma 2.13, Lemma 2.12, and partially repeat the formal argumentation of the proof for Lemma 2.12, therefore, we omit these proofs.

The rewriting algorithm guarantees the simulation relation between an original and the rewritten Kripke structures, with B^{rw_1} as the simulation relation.

Theorem 2.15
$K \succ K^{rw1}$

Memory update rewrites

This rewriting technique eliminates combined read-write operations. We rewrite all terms of the form $read(write(mt, at_1, dt), at_2)$ by propagating the read operation:

$$read(write(mt, at_1, dt), at_2) \equiv ITE(at_1 = at_2, dt, read(m, at_2)).$$

Obviously, this rewriting technique preserves satisfiability of a formula. Application of this technique to a Kripke structure guarantees the bisimulation.

The combination of this techniques with the one presented above allows us to eliminate all memory equalities and memory write operations. The rewritten formula/Kripke structure can only contain memory read operations. We denote the rewritten formula ψ by ψ^{rw} and rewritten Kripke structure K by K^{rw}. The rewritten K^{rw} only simulates the original K because the first rewriting technique can only guarantee simulation relation.

2.3.2 Elimination of Function Symbols

Often, the actual definition of a function is irrelevant for the truth value of a formula, i.e. this formula is valid for all interpretations of the function. In such a case one could drop the function interpretations, and, hence, the functions are represented by uninterpreted function symbols. This allows to reduce the description size of the model and to abstract part of the model behavior. However, this can increase the state space of the model, because the function symbols are modelled as mappings. In this section, we present an algorithm which transforms an ELF-formula, without memory writes and memory equalities, into a more simple EL-formula without function symbols. We describe how this transformation can be applied to Kripke structures. We also prove the soundness of the former transformation, i.e. the transformed Kripke structure simulates the original one.

Application to formulae

Ackermann [Ack54] proposed a methodology for elimination of the function symbols in an ELF formula. His proposal is to replace each function application $f(x)$ by a fresh variable f_x, which is not used in the formula. Furthermore, he added several restrictions to impose functional consistency: if the arguments of the original function applications are equal, the fresh variables are equal too. He also proved that this transformation preserves formulae satisfiability (and validity).

Burch and Dill's approach to function elimination relies on the usage of ITE constructs [BD94]. For each syntactically different occurrence of a function application a new ITE is added. For example, the three function applications $f(x)$, $f(y)$, and $f(z)$ are replaced by f_x, $ITE(x = y, f_x, f_y)$, and $ITE(y = z, ITE(x = y, f_x, f_y), f_z)$.

As the base for our algorithm we will use the Burch and Dill approach. We denote the above outlined ITE-transformation of a formula ψ, to one without function symbols, by ψ^*.

Lemma 2.16
Given a formula ψ and an interpretation $I = (I_v, I_f)$. Let I_v^* be an interpretation for the fresh variables, which maps the fresh variables to the value produced by the corresponding function application, e.g. $I_v^*(f_t) = I_f(f)(eval(I, t))$. For the ITE-transformed formula the following holds:

$$I_v \cup I_v^* \models \psi^* \Leftrightarrow (I_v, I_f) \models \psi$$

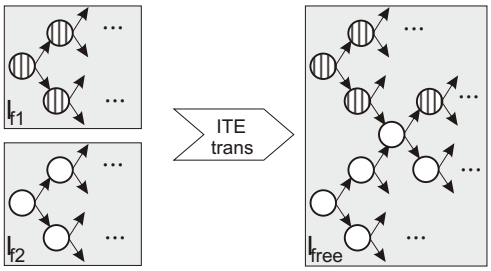


Figure 2.5: The lack of temporal function consistency leads to more interleaving runs.

This lemma captures exactly those interpretations of the fresh variables which correspond to the result of the corresponding function application.

It is important to note that, the ITE-transformation only changes and depends on the function applications in a formula, not on its structure. Hence, we can also apply it to a set of formulae $\{\psi_1, \ldots, \psi_n\}$, where function applications in all formulae are substituted simultaneously. For each ψ_i^* of the resulting set the property of Lemma 2.16 holds.

Application to Kripke Structures

We apply the function elimination to a Kripke structure K, by first translating it to a context. Then, we rewrite memory equalities followed by application of ITE-transformation to the context. Finally, we translate the transformed context back to a Kripke structure. The resulting Kripke structure is denoted by K^*.

The good news is that K^* contains neither uninterpreted function applications nor memory applications. The trade off is that K^* contains the fresh domain variables, which represent the result of the corresponding function applications. Note, since memories only occur in the read and the write operations, and because these operations are also eliminated we do not longer need to model the memories in the state space. The temporal behavior of these fresh domain variables is not specified. Thus, they can change non-deterministically and functional consistency over time is lost. In particular, the memory semantics is not preserved any more. The ITE-transformed Kripke structure K^* contains more transitions than a "union" of all Kripke structures K^{I_f} for all possible interpretations I_f (Figure 2.5). For example, let f_3 be a fresh variable which represents the function application $f(3)$. Then, at some point in time t in a run of K^*, it can have value $f_3^t = 3$ but at the next point something else $f_3 = 4$. Thus, the "interpretation" of f in K^* has been changed during the model run, that is not possible in the non-transformed Kripke structure K^{I_f}.

Nevertheless, we can prove that K^* simulates K:

Theorem 2.17
$K^* \succ K$

We can build a Kripke structure which preserves more behaviour of K than K^*

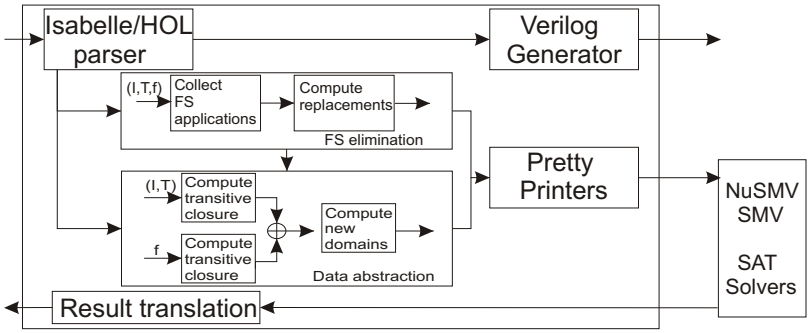


Figure 2.6: The IHaVeIt structure.

by the introduction of Temporal Functional Constraints (TFC). These constraints restrict the behavior of the fresh domain variables, e.g. if the arguments of a function application are equivalent at t and $t + 1$, the result of the function application should be the same. We define TFC formally as follows:

$$\forall a \in Add_{K^*}(m).\ \forall b \in Add_{K^*}(m).\ a' = b \longrightarrow f'_a = f_b$$

with $Add_{K^*}(m) \triangleq \bigcup_{f \in co(K^*)} Add_f(m)$.

These constraints are added to the transition predicate of the transformed Kripke structure K^*.

To conclude this section we note that an application of the presented rewriting techniques eliminates all memory/function operations. However, this has its price: the transformed Kripke structures are only simulations of the original ones. Therefore, false negatives are possible.

2.4 Summary: Who Has It?

The answer to this question is: IHaVeIt [Tve05a] (Isabelle Hardware Verification Infrastructure). IHaVeIt is a design and a verification environment, which is based on the interactive theorem prover Isabelle/HOL. It is implemented in the Standard ML language and is built in Isabelle/HOL as an oracle. The tool is available at the project homepage [Tve05b].

The IHaVeIt task is the efficient automatic verification of temporal and combinational properties from Isabelle/HOL [NPW02]. The main contribution of the tool is the introduction of a symbolic level pre-processor for Kripke structures. This preprocessor implements the transformation algorithms, which have been described above. Thus, IHaVeIt itself can not prove any lemmas, it transforms a given formula/Kripke structure and passes it to an external tool. IHaVeIt employs automatic tools such as model checkers (NuSMV [CCG+02], Cadence SMV [McM99]), and SAT solvers as back ends.

IHaVeIt has a modular structure and has the following component (Figure 2.6):

- parser for a subset of Isabelle/HOL language,
- elimination of function symbols (Section 2.3),
- data abstraction (Section 2.2.2),
- pretty-printers for NuMSV, SAT, Cadence SMV[1], and Verilog[2].

2.4.1 Hardware in Isabelle/HOL

The higher order logic (HOL) gives the user the freedom to define almost anything she/he has in mind. However, the hardware designs in this thesis have to be synthesizable into real hardware, e.g. designs have to be translatable into Verilog. Moreover, these designs are inputs for the external automatic tools, which we use to reduce the user's work. In this section, we present a suitable subset of Isabelle/HOL for description of the desired hardware designs.

Types

We employ a fragment of the Isabelle/HOL language which consists of expressions involving the following types: booleans, bit vectors, naturals, integers, lists, functions, finite enumerations, and records. We shrink the infinite types, e.g. naturals and lists, by means of *predicate sets*. A predicate set defines the set of all elements satisfying a given predicate. We defined in Isabelle/HOL a library of predicate sets for the supported types[3], e.g. $bv\_n(n)$ and $arr\_of(n, t)$ specify the set of bit vectors of the length n and the set of arrays of the length n with the elements of subtype t respectively.

Expressions

We model the combinational circuits in Isabelle/HOL via expressions. These expressions are built up as usual Isabelle/HOL expressions which are either standard operators (e.g. plus, list head and tail) or user-defined functions. We support four kinds of user-defined functions: non-recursive functions, sub-typed lambda expressions, recursive functions, and uninterpreted functions. Non-recursive functions are built up via combination of the Isabelle/HOL operators. Sub-typed lambda expression is a wrapper for Isabelle's lambda term which is only defined if the argument for the lambda expression is of the correct sub-type. We use function *AbsSubtyped* to denote such an expression, for example:

$$AbsSubtyped((\lambda x.\ x + 1), nat\_range(0, 4)) \triangleq \lambda x.\ \begin{cases} x + 1 & : \ x \in nat\_range(0, 4) \\ arbitrary & : \ else \end{cases}$$

[1] This is joint work with Hristo Tzigarov.
[2] This is joint work with Andrey Shadrin.
[3] See Appendix A for the full description.

The expression $AbsSubtyped((\lambda x.\ x + 1), nat\_range(0, 4))(y)$ is only defined if y is in a natural number between 0 and 4.

The user can define recursive functions either by recursion on natural numbers or the length of a list. During the translations to the external tools or Verilog, all applications of recursive functions are unrolled into the set of non-recursive function calls. Based on the early experience in hardware design and verification, these two types of recursion are enough to build and verify huge and complex designs, e.g. the VAMP processor [Krö01, Dal06, Bey05] and the work in this thesis.

Uninterpreted functions are introduced by defining the function name and its signature. The signature consists of a set of the subtypes for the inputs and a subtype for the output of the function. These functions are mostly used for the verification purposes, and they allow abstraction of the model without changing it. We demonstrate their usage in Section 2.4.2. These functions are translated to Verilog as modules without bodies, e.g. the user can insert in there the hardware designs which are not modelled in Isabelle/HOL.

Theorems

IHaVeIt targets two kinds of theorems: combinational and temporal ones. A combinational theorem is an expression of boolean type where all free variables have to be quantified over their subtypes, e.g.

$$\forall\ a \in\ arr\_of(4, bv\_n(8)).\ P(a),$$

where $arr\_of(4, bv\_n(8))$ specifies an array with four elements and each element is a bit vector of length eight.

A temporal theorem is an LTL or a CTL formula[4] stated over a Kripke structure, e.g.

$$K \models_{ltl} ltl\_formula.$$

Kripke structure K is a triple (S, I, T), where S is a predicate set, I is an initial predicate, and T is a transition predicate defined in terms of the current and the next state.

2.4.2 Functional Abstraction

A pure application of data abstraction to a given model and a given property may only slightly reduce the model state. This is because the functions, which actually are not relevant for the property, may violate the data independency of the model. If we drop definitions of all defined functions (as we did in Section 2.3), we may abstract too much behavior of the original model. Hence, it could lead to many false negatives. Our approach is to drop the definitions of only those functions which are indeed irrelevant to the property. IHaVeIt allows the user to mark these functions. The

[4] Appendix B.1 and Appendix B.2 describe the syntax and semantics of LTL and CTL supported by IHaVeIt.

user should only add the name of the function and its subtype to the assumptions of the theorem, and IHaVeIt will treat this function as an uninterpreted one. Thus, the user abstracts the model without actually doing any changes on it. Note, in most of the related work this step is done informally. In this thesis, we verify huge and detailed systems in Isabelle/HOL, where sub-systems are verified (semi-)automatically. This requires a strong formalization of every proof step, also including the introduction of the uninterpreted functions. For example, let us consider the following theorem:

$$\begin{array}{l} \forall a \in bv\_n(5).\ Q(a) \in bv\_n(32) \\ \Longrightarrow \\ (S, I, T) \models_{ltl}\ ltl\_formula \end{array}$$

In this case, IHaVeIt treats the function Q as an uninterpreted one, which takes a bit vector of length five and produces a bit vector of length 32. Thus, the theorem will be proven for all interpretations of Q. Now, if we want to have this theorem for a concrete definition of Q, we have to check whether this definition indeed satisfies the specified signature, i.e. $\forall a \in\ bv\_n(5).\ Q(a) \in bv\_n(32)$. IHaVeIt can also be used to prove such theorems automatically.

2.4.3 Benchmarks

We demonstrate the efficiency of our environment via an automatic verification of two different bit-level hardware designs: liveness of a pipelined machine and correctness of a memory management unit. We run all experiments on a Dual AMD Opteron 2.4 GHz with 4 Gb RAM.

Processor

The processor is a 5-stage pipelined DLX-like machine which implements 36 32-bit instructions (e.g. shifts, store, jumps, branch instructions). The data path width can be set to *any integer value* and the verification time of our proofs is independent of this value. The processor comprises 32 general purpose registers (GPR), 31 internal registers (such as PC, DPC, instruction register), and the memory has type $2^{30} \rightarrow 2^{32}$. The model includes a delayed PC (*DPC*), as well as a three-stage forwarding and stalling mechanism [MP00, Krö01]. The model description is 2778 line of code and contains *bit-level* description of all parts, e.g. ALU, shifters. Our DLX machine is similar to the most complex benchmark used in the recent related work (e.g. Manolios et al.[MSV06]). We verified the processor liveness property which requires that an instruction is *eventually* fetched and it *eventually* leaves the processor. We verify this property by proving the liveness of the stalling logic of the processor.[5] For the five-stage processor, we have five stall signals. They control when the stage should be stalled, i.e. no instructions in upper stages can proceed. We directly formulate this

[5]Here, we assume that the instruction memory and the data memory (memory busy signals) are alive.

theorem [Krö01] using LTL:

$$(S_{MA}, I_{MA}, \delta_{MA}) \models_{LTL}$$
$$GF(\neg IMbusy) \wedge\ GF(\neg DMbusy) \longrightarrow \forall 1 \leq i \leq 5.\, GF(\neg stall_i)$$

This theorem states that if *IMbusy* and *DMbusy* signals are finitely often true, the stall signal of every stage is finitely often true (i.e. every stage will be stalled for a finite number of consecutive cycles). We also checked a version of the DLX processor with a bug in the stalling engine. Table 2.1 shows the verification time of the correct machine (dlx5) and the buggy machine (dlx5b). The results are obtained by applying NuSMV with and without preprocessing by IHaVeIt. The preprocessing makes the application of this model checker feasible. Similarly, we compare IHaVeIt and SMV. We run SMV with counterexample-based abstraction turned on. Here, we achieve several time speed-up over plain usage of SMV.

Memory Management Unit

Memory Management Unit (MMU) is the hardware support for a virtual memory. The virtual memory mechanism relies on a main memory (e.g. RAM) and a swap memory (e.g. hard drive disk). An MMU is usually placed in the processor just before the memory interface and it translates virtual addresses into physical ones. A processor with an MMU runs in either the user mode or the system mode. In the system mode, the MMU is not used. In this mode processor directly accesses the main memory. In the user mode, the address translation has to be done, i.e. a virtual address is translated to a physical one. Then, the physical address is used to access the main memory. There are several cases when the translation is impossible, e.g. the accessed data are not in the main memory, violation of access rights. In these cases MMU rises an exception, the processor enters the system mode and executes a page fault handler. For more details on virtual memory and MMU we refer the reader to [DHP05].

In this section we consider an MMU model, which is an optimization of the MMU presented in [DHP05], and it has been implemented by Dalinger. The optimized MMU has a translation look-aside buffer (TLB), which caches the recently used translations to speed-up the following translations. The MMU design is not really big. However, to verify it the interfaces to/from the TLB, the processor, and the memory have to be modelled. These interfaces increase the model state space drastically, e.g. the memory has type $2^{29} \rightarrow 2^{64}$.

The MMU correctness can be split according to the access type: read/write translated/untranslated with/without exception. The correctness includes the following assumptions: (i) $proc\_if\_correct$ – the processor interface to the MMU has a correct behavior (e.g. it doesn't start a new request while a previous one is still running), and (ii) $mem\_if\_correct$ – the main memory interface to the MMU is correct (e.g. the memory is alive and its content is stable). We describe the correctness of the untranslated read access [Dal06] without exception directly in LTL as follows:

	NuSMV	NuSMV + IHaVeIt	SMV	SMV + IHaVeIt
dlx5	>2days	23 h	2.63 s	0.35s
dlx5b	>2days	1 h 09 m	0.46 s	0.27s
mmu(read, weak)	1 m 24.27 s	0.43 s	n.a	n.a.
mmu(write, weak)	1 m 24.27 s	0.43 s	n.a	n.a.
mmu(read, strong)	>2days	1.61 s	n.a	n.a.
mmu(write, strong)	>2days	2.14 s	n.a	n.a.

Table 2.1: Comparison of verification time with and without preprocessing by IHaVeIt; n.a. – not applicable because of LTL past operators in the formula.

$$(S_{MMU}, I_{MMU}, \delta_{MMU}) \models_{LTL} proc\_if\_correct \wedge\ mem\_if\_correct \longrightarrow$$
$$G\ (uread\_req\_start \wedge\ (X \neg mmu\_out\_excp) \longrightarrow$$
$$\neg end\_req\ U\ (end\_req \wedge mmu\_out\_data\_correct))$$

This lemma states that if there is a read request from the processor and the MMU doesn't generate an exception then (i) eventually the end of the request is signalled by the MMU and the output data is the content of the memory cell on the translated address, and (ii) in between there was no request end, i.e. no requests are lost by MMU. Table 2.1 shows verification time for the untranslated read and write accesses. We tried two different property formulations: a weak formalization requires some additional user work to reuse it in the processor verification; a strong one is the formalization given in [Dal06]. In comparison to the plain application of NuSMV we get several order magnitude speed-up. We could not apply the SMV because our LTL formulae contain past temporal operators, which are not supported by SMV.

Cache System

A cache system is a part of a processor, which is responsible for the management of the data transfer from/to the memory and its intermediate storage. The main purpose of the cache system is to speed up most memory accesses without changing their semantics. Thus, the task of a correct cache system is to simulate the behavior of a memory and be transparent for the processor. Therefore, for each memory access to some address the cache system should always return the last written data to this address. If there was no write accesses to this address, some initial memory content has to be returned. This property is often called data consistency.

Müller [Mül07] implemented and verified a simple cache system in Isabelle/HOL with the help of IHaVeIt. This cache system consists of interconnected instruction and data caches. Thus, it allows fetching a new instruction and reading/writing data simultaneously. Müller automatically verified liveness of the whole cache system, and control and data automata. He also noticed that the application of the algorithm

for elimination of uninterpreted functions makes automatic verification of the data consistency impossible. It is expected because the latter algorithm does not preserve memory semantics (see Figure 2.5). This part of the proof he interactively carried out in Isabelle/HOL.

Bus Controller

In a Verisoft subproject an automotive system was developed and verified. The basic element of this system is an electronic control unit (ECU), which consists of a processor and a bus interface. Several ECUs, which are connected via a bus, represent the automotive system at the gate level. The implementation of this bus interface is called automotive bus controller (ABC). The goal of the ABC is to provide

- a data exchange with the processor
- a mechanism for the message transmission
- a scheduling mechanism initiating the message transmission and providing a common time base.

IHaVeIt has been heavily used in the verification of the ABC [ABK08], e.g. for verification of the purely digital (non-real time) parts of the design.

Computer System

In the following section we present and verify a computer system, which consists of a pipelined processor with out-of-order execution and external devices. Unfortunately, the state-of-the-art automatic methods cannot automatically verify systems of such complexity. Our strategy is to verify the system interactively in Isabelle/HOL and to support our proofs by usage of IHaVeIt. In Section 3.4 we give a comparison of our approach with a fully interactive one.

Acknowledgment

I would like to thank Eyad Alkassar for the joint work on [TA08], which is the base for Section 2.2 and Section 2.3.

Chapter 3

VAMP^{XT}

The work in this chapter is a further development of the VAMP processor. The base for the VAMP processor was designed and verified by Kröning [Krö01]. Jacobi [Jac02] developed and verified the VAMP's floating point units. The VAMP memory unit and caches were designed and verified by Beyer [Bey05]. He also instantiated the base model developed by Kröning with the floating point untis and the memory unit. Dalinger [Dal06] extended the memory system with a memory management unit (MMU), which is a hardware support for the virtual memory mechanism. In this chapter, we consider the VAMP processor as the base of a computer system. The latter requires several modifications in the design to allow the communication with other parts of the system, i.e. with the external devices. This, of course, requires re-considering the correctness proofs. The main contribution of this chapter is the new version of the verified VAMP processor, which is designed to be used in a computer system.

This chapter is organized as follows. In the following section we present a specification of the VAMP processor and extend it with the interfaces for the external devices. We introduce the VAMP on the gate level in Section 3.2. Section 3.3 presents the VAMP correctness criterion, which was formally proved in the Isabelle/HOL system.

3.1 The VAMP Specification

A processor specification is usually modelled as an automaton as seen by the programmer. A configuration of this automaton represents a processor state. It has the regular components such as a memory, a register file, and a program counter. Every automaton step corresponds to the execution of one instruction pointed by the program counter. This automaton is usually called Instruction Set Architecture (ISA). We also call it the processor specification.

The VAMP specification state consists of two program counters, three register files and a memory. The two program counters realize a so-called *delaed-PC* mechanism with one delay slot. In this mechanism the current PC update will not affect the next

instruction but the instruction after the next one. These two program counters are $PC \in \mathbb{B}^{32}$ and $DPC \in \mathbb{B}^{32}$. A complete formal description of the mechanism is given by Müller and Paul [MP00].

The VAMP provides three sorts of register files: (i) general purpose registers *GPR*; it consists of 32 registers of 32-bit width each; register 0 always contains zero, i.e. 0^{32}. (ii) floating point registers *FPR*; it can be accessed as a register file with 32 32-bit registers or as a register file with 16 64-bit registers. (iii) special purpose registers *SPR*; it consists of 17 32-bit registers. These registers are used for various processor management purposes, e.g. storing the interrupt causes and a number of exception data, holding data for the virtual memory mechanism. The description of every *SPR* register is given by Dalinger [Dal06, Chapter 3].

The VAMP specification memory $M : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$ is a partial mapping. from 29-bit addresses to 64-bit data. We denote by *MA* the set of all addresses where the memory *M* is defined.

Formally, an ISA configuration c_P is a 6-tuple:

$$c_P \triangleq (PC, DPC, GPR, FPR, SPR, M)$$

We use the notation $c_P.F$ to denote the state of one of the six configuration components. We also introduce type C_P to denote the type of the ISA configuration.

The ISA communicates with the external devices in two ways. First, ISA may access the devices by reading or writing words over the so-called device input and output interfaces, which are named *difi* and $difo_s$ respectively.[1] The naming conventions *difi* and $difo_s$ are from the device point of view. Second, the devices may interrupt the processor by signaling the external events on the $eev \in \mathbb{B}^{19}$ channel.

We model $difi \in Difi$ as the following tuple:

$$difi \triangleq (req, w, a, din)$$

where the active value of the boolean flag $difi.req \in \mathbb{B}$ signals the presence of a request; the active value of the boolean flag $difi.w \in \mathbb{B}$ corresponds to a write access and the inactive value to a read access; $difi.a \in \mathbb{B}^{30}$ specifies the accessed device and its port; $difi.din \in \mathbb{B}^{32}$ provides the data for a write access. The device address $difi.a$ has the following format (Appendix C.2). The bits $difi.a[29:13]$ have to be set to ones and they specify that the devices are mapped above physical memory. The bits $difi.a[12:10]$ specify the accessed device and $difi.a[9:0]$ specify the accessed device port. Thus, we support up to eight devices with up to 1024 ports of width 32 bits.

The processor accesses a device when it executes a regular load/store access to/from a device address $difi.a$, which is associated with the accessed device. This access will be placed on the *difi* channel and the device answer will be read from the $difo_s \in \mathbb{B}^{32}$ channel. We denote by *DA* the set of all device addresses. Note that the accesses to the device address space do not affect the processor memory.

[1] The *s* in $difo_s$ stands for *specification*. We use *s* in order to distinguish the specification data from the *difo* which is used in the implementation (Section 3.2.2) and has a slightly different type.

In previous work the memory was treated as a total mapping. The introduction of the memory mapped I/O devices takes away a part of the memory address space. Thus, *DA* and *MA* represent the set of defined addresses. The processors can be used with many or no devices. Therefore, we consider every instruction which accesses a memory outside *MA* as an instruction with a legal device access. Thus, the absence of accesses to the undefined addresses has to be guaranteed by the programmer. We introduce a suitable software condition at the end of this chapter.

Now we introduce the next-state function Δ_P of the processor specification. This function is a revision of the function presented by Dalinger [Dal06]. The revision is due to the introduction of the external devices. Whenever we use Dalinger's definitions we up-scribe them with the tag *old*, e.g. we denote by f^{old} Dalinger's definition of the function f.

The Δ_P function computes the next processor state $c'_P \in C_P$ based on a given previous state $c_P \in C_P$, an external interrupt vector $eev \in \mathbb{B}^{19}$, and data provided by the devices $difo_s \in \mathbb{B}^{32}$:

$$c'_P \triangleq \Delta_P(c_P, eev, difo_s)$$

In the following we develop the definition of Δ_P.

The processor operates in one of two modes: *user* or *system* mode. These two modes are distinguished by the value of the *SPR* register number 16 (the *MODE* register). If bit zero has value 1, the processor operates in the user mode; otherwise the processor is in system mode. We denote the value of this bit by $vmode(c_P) \triangleq c_P.SPR[MODE][0]$. In the user mode the processor uses address translations. In this mode the addresses of all memory accesses (load, store, and instruction fetch) are treated as virtual ones. They have to be first translated into the corresponding physical addresses. The physical addresses are then used to execute the memory/device accesses.

The main data structure for address translation is the page table. Every entry in this table contains the address of a memory page and the flags which specify the allowed access mode. This page table is saved in the processor memory. It can be located via the page table origin (PTO) and the page table length (PTL). Both are saved in the register bank *SPR*, PTO is the register number and 9 and PTL is number 10. We access these registers as $c_P.SPR[PTO]$ and $c_P.SPR[PTL]$.

The address translation of a virtual address $va \in \mathbb{B}^{32}$ is done in two steps. In the first step one computes the page table entry address $ptea \in \mathbb{B}^{30}$. Function *compute_ptea* computes *ptea* based on a given processor configuration c_P, and a virtual address *va*.

$$ptea \triangleq compute\_ptea(c_P, va)$$

In the second step one reads the page table entry, computes the physical address, and checks the access mode.

The whole translation is computed by the function *decodeitr* [Dal06, Chapter 2]. This function, for a given processor configuration c_P, a flag *mw* signaling the type of

the access, and a virtual address *va*, computes the corresponding physical address. It raises an exception, if the translation cannot be done.

$$(pa, excp) \triangleq decodeitr(c_P, mw, va)$$

The semantics of the outputs is as follows: if the boolean flag *excp* has value 0, $pa \in \mathbb{B}^{32}$ is the target physical address. If flag *excp* has value 1, the translation can not be done (e.g. due to the page fault). In this case the *pa* output is a bit vector filled with zeros and is considered to be invalid. We employ the function $ipa(c_P)$ to denote the physical address for fetch access.

Definition 3.1

$$ipa(c_P) \triangleq decodeitr(c_P, 0, c_P.DPC).pa$$

Note that function *ipa* is only used in the user mode. In the system mode the processor does not use the virtual memory mechanism. All memory addresses are directly interpreted as physical ones.

The fetch of an instruction can cause exceptions. Whenever the fetch address is not word-aligned, we raise the instruction misalignment interrupt (*imal*).

Definition 3.2

$$imal(c_P) \triangleq \quad c_P.DPC \bmod 4 \neq 0$$

We raise the instruction page fault exception *ipf* whenever the translation of the fetch address can not be done or the fetch address points outside the processor memory (i.e. instructions can only reside in *MA*). We also raise this interrupt if the page table entry address *ptea* is not in the physical memory.

Definition 3.3

$$\begin{array}{rll} ipf(c_P) \triangleq & vmode(c_P) \wedge & (decodeitr(c_P, 0, c_P.DPC).excp \vee \\ & & ipa(c_P)[31:3] \notin MA \vee \\ & & compute\_ptea(c_P, c_P.DPC)[29:1] \notin MA) \vee \\ & \neg vmode(c_P) \wedge & c_P.DPC[31:3] \notin MA \end{array}$$

The function *IR* computes (or fetches) an instruction which has to be executed in the current configuration c_P:

Definition 3.4
Let *PCdata* be the double word in the memory pointed to by the current program counter, i.e.

$$PCdata \triangleq \begin{cases} c_P.M[c_P.DPC[31:3]] & : \neg\, vmode(c_P) \wedge \neg\, imal(c_P) \wedge \neg\, ipf(c_P) \\ c_P.M[ipa(c_P)[31:3]] & : vmode(c_P) \wedge \neg\, imal(c_P) \wedge \neg\, ipf(c_P) \\ 0^{64} & : otherwise \end{cases}$$

According to the second bit we select which part of the read double word we use as the fetch result:

$$IR(c_P) \triangleq \begin{cases} PCdata[63:32] & : c_P.DPC[2] \\ PCdata[31:0] & : otherwise \end{cases}$$

The result of the application of *IR* is a bit vector which encodes the instruction. There are many functions which decode the instruction characteristics (Appendix C). For example, $RS\,1(c_P)$ computes the address of the first operand in the register file, $RD(c_P)$ computes the address of the destination register in the register file etc. We introduce the predicate $mem?(c_P)$ which holds for all memory instructions, the predicate $word?(c_P)$ which holds for the memory instructions with a word access (i.e., reading or storing a 32-bit word), and the predicate $mw?(c_P)$ which holds for all memory instructions with a write access.

The data memory accesses (loads and stores from/to memory or devices) are based on the computation of the effective address.

Definition 3.5

$$ea(c_P) \triangleq c_P.GPR[RS\,1(c_P)] +_{32} sext(imm(c_P))$$

where $imm(c_P)$ is the immediate constant of the instruction $IR(c_P)$ and *sext* computes 32-bit sign-extension version of $imm(c_P)$.

If the processor operates in the system mode and a memory instruction has to be executed, $ea(c_P)$ computes a physical address. If the processor operates in the user mode, $ea(c_P)$ defines a virtual address for the memory access. We employ the shorthand $dpa(c_P)$ to denote the result of the address translation for the data access:

$$dpa(c_P) \triangleq decodeitr(c_P, mw?(c_P), ea(c_P)).pa$$

Note that function *dpa* is only used in the user mode.

There are two exceptions which can take place during a memory access. The data misalignment exception (*dmal*) occurs whenever the access address is not aligned with the access type, e.g. if we have a word access but the address is not word aligned (for the formal definition see [Dal06, p. 42]).

The data page fault exception (*dpf*) is raised if the address translation can not be done, the page table entry is outside the processor memory, or physical address is outside the processor memory.

Definition 3.6

$$\begin{aligned} dpf(c_P) \triangleq\ & mem?(c_P) \wedge vmode(c_P) \wedge \\ & (decodeitr(c_P, mw?(c_P), ea(c_P)).excp \vee \\ & compute\_ptea(c_P, ea(c_P))[29:1] \notin MA) \vee \\ & mem?(c_P) \wedge vmode(c_P) \wedge dpa(c_P)[31:3] \notin MA \wedge \neg word?(c_P) \vee \\ & mem?(c_P) \wedge \neg vmode(c_P) \wedge ea(c_P)[31:3] \notin MA \wedge \neg word?(c_P) \end{aligned}$$

Note that *dpf* is also raised if the device access is not word-aligned. We use *dpf*, instead of *dmal*, to signal a misaligned device access because (i) this test may require address translation, and (ii) this translation can already generate *dpf* exception.

Now we can define the step function for the memory.

Definition 3.7
We define an auxiliary function $byteupd(bw, a, b) \in \mathbb{B}^{8*n}$, with $n \in \mathbb{N}$. It takes a set of bytes indices, which is encoded as a bit vector, $bw \in \mathbb{B}^n$, and two bit vectors $a \in \mathbb{B}^{8*n}$, and $b \in \mathbb{B}^{8*n}$. The result is a bit vector composed from bytes of a and b as follows:

$$| byteupd(bw, a, b) |_i \triangleq \begin{cases} | b |_i & : bw[i] \\ | a |_i & : otherwise \end{cases}$$

where $0 \leq i < n$.

Definition 3.8
Let $bw \in \mathbb{B}^n$ be the set of indices of bytes which have to be written.[2] Let $dest \in \mathbb{B}^{64}$ be the data to be written to the memory.

The updated memory is computed as follows:

$$c'_{\mathrm{P}}.M \triangleq \begin{cases} c_{\mathrm{P}}.M(ea(c_{\mathrm{P}})[31:3] := byteupd(bw, c_{\mathrm{P}}.M(ea(c_{\mathrm{P}})[31:3]), dest)) \\ \qquad : mw?(c_{\mathrm{P}}) \wedge \neg vmode(c_{\mathrm{P}}) \wedge \\ \qquad \neg dmal(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}})[31:3] \in MA \\ \\ c_{\mathrm{P}}.M(dpa(c_{\mathrm{P}})[31:3] := byteupd(bw, c_{\mathrm{P}}.M(dpa(c_{\mathrm{P}})[31:3]), dest)) \\ \qquad : mw?(c_{\mathrm{P}}) \wedge vmode(c_{\mathrm{P}}) \wedge \neg dmal(c_{\mathrm{P}}) \wedge \\ \qquad \neg dpf(c_{\mathrm{P}}) \wedge dpa(c_{\mathrm{P}})[31:3] \in MA \\ \\ c_{\mathrm{P}}.M \quad : otherwise \end{cases}$$

In previous work the *GPR* and the *FPR* register files can only be updated by the data which are already stored in the processor configuration: either in the register files or the specification memory. Our model can additionally accept data from the external devices, which are only used for load word operation. This corresponds to the load operations on addresses from *DA*, or more exactly not from *MA*:

[2] See Appendix C, Table C.1, and Table C.4 for details.

Definition 3.9

$$c'_{\mathrm{P}}.GPR[RD(c_{\mathrm{P}})] \triangleq \begin{cases} difo_s & : lw?(c_{\mathrm{P}}) \wedge \neg\, dmal(c_{\mathrm{P}}) \wedge \neg\, dpf(c_{\mathrm{P}}) \wedge \\ & (\neg\, vmode(c_{\mathrm{P}}) \longrightarrow ea(c_{\mathrm{P}})[31:3] \notin MA) \wedge \\ & (vmode(c_{\mathrm{P}}) \longrightarrow dpa(c_{\mathrm{P}})[31:3] \notin MA) \\ & \\ c_{\mathrm{P}}.M[ea(c_{\mathrm{P}})[31:3]][31:0] & \\ \quad : lw?(c_{\mathrm{P}}) \wedge \neg\, vmode(c_{\mathrm{P}}) \wedge & \\ \quad \neg\, dmal(c_{\mathrm{P}}) \wedge \neg\, dpf(c_{\mathrm{P}}) \wedge & \\ \quad ea(c_{\mathrm{P}})[31:3] \in MA \wedge \neg ea(c_{\mathrm{P}})[2] & \\ & \\ \quad \vdots \qquad \vdots & \\ c_{\mathrm{P}}.GPR[RS\,1] +_{32} c_{\mathrm{P}}.GPR[RS\,2] & : add?(c_{\mathrm{P}}) \\ \quad \vdots \qquad \vdots & \\ c_{\mathrm{P}}.GPR[RD(c_{\mathrm{P}})] & : otherwise \end{cases}$$

where

- the predicate *lw*? signals the execution of a load-word instruction
- the predicate *add*? signals the execution of the fixed-point addition of two registers.

Let the function $FPRupdate^{old}(c_{\mathrm{P}})$ computes the new state of the *FPR* register file as specified by Jacobi [Jac02]. We extend the update of *FPR* register file for the case of loading one word from the devices.

Definition 3.10
Let *data* be the old value of the destination register, i.e. $data = FPR[RD(c_{\mathrm{P}})]$. Let $data'$ be the data which to be written into the destination register in case of a load instruction, i.e.

$$data'[31:0] = \begin{cases} difo_s & : \neg RD(c_{\mathrm{P}})[0] \\ data[31:0] & : otherwise \end{cases}$$

$$data'[63:32] = \begin{cases} difo_s & : RD(c_{\mathrm{P}})[0] \\ data[63:32] & : otherwise \end{cases}$$

Note that $fdouble(c_{\mathrm{P}})$ holds if the instruction $IR(c_{\mathrm{P}})$ executes a double word access

and $fload?(c_P)$ holds for a load operation to a *FPR* register.

$$c'_P.FPR[RD(c_P)[4:1]] \triangleq \begin{cases} data' & : fload?(c_P) \wedge \neg\, dmal(c_P) \wedge \neg\, dpf(c_P) \wedge \\ & (\neg\, vmode(c_P) \longrightarrow ea(c_P)[31:3] \notin MA) \wedge \\ & (vmode(c_P) \longrightarrow dpa(c_P)[31:3] \notin MA) \\ \\ FPRupdate^{old}(c_P)[RD(c_P)[4:1]] & : otherwise \end{cases}$$

The program counters are updated with respect to the delayed *PC* mechanism. For example, for all instructions except branches, jumps, return from exception (*rfe*), and in case of an interrupt, the *DPC* receives the value of the *PC*. The *PC* is incremented by 4. Thus, it points to the next instruction.

$$c'_P.DPC = c_P.PC$$
$$c'_P.PC = c_P.PC +_{32} 4$$

The update of the program counters for other instructions is formally specified in [Dal06, Section 3.1].

The step functions for other components are modelled in exactly the same way as in the previous work. The updates of the *SPR* and the interrupt semantics are described in detail by Dalinger [Dal06]. This concludes the formal definition of the specification step function Δ_P.

Now we define the ISA output function Ω_P. This function, for a given processor state c_P, computes the processor request to the devices *difi*. The request consists of two flags *req* and *w* specifying the type of the request, the access address *a*, and the data *din* to be written in case of a write access.

Definition 3.11
Let $dest \in \mathbb{B}^{32}$ be the data to be written to the device; it can be the content of a register from the *GPR* register file or a non-double register from the *FPR* register file. Let predicates *lw*? and *sw*? signal a load-word instruction or a store-word instruction respectively. Let $load\_dev?(c_P)$ hold if processor reads from a device, i.e. $load\_dev?(c_P) \triangleq lw?(c_P) \vee (fload?(c_P) \wedge \neg fdouble(c_P))$. Similarly, let $write\_dev?(c_P)$ hold if processor writes to a device, i.e. $write\_dev?(c_P) \triangleq sw?(c_P) \vee (fstore?(c_P) \wedge$

$\neg fdouble(c_{\mathrm{P}})$).

$$\Omega_{\mathrm{P}}(c_{\mathrm{P}}).difi \triangleq \left[\!\!\left[\begin{array}{lcl} req & = & (load\_dev?(c_{\mathrm{P}}) \vee write\_dev?(c_{\mathrm{P}})) \wedge \\ & & \neg\, dmal(c_{\mathrm{P}}) \ \wedge \ \neg\, dpf(c_{\mathrm{P}}) \wedge \\ & & (\neg vmode(c_{\mathrm{P}}) \longrightarrow ea(c_{\mathrm{P}})[31:3] \notin MA) \wedge \\ & & (vmode(c_{\mathrm{P}}) \longrightarrow dpa(c_{\mathrm{P}})[31:3] \notin MA) \\ w & = & sw?(c_{\mathrm{P}}) \\ a & = & \begin{cases} dpa(c_{\mathrm{P}})[31:2] & : vmode(c_{\mathrm{P}}) \\ ea(c_{\mathrm{P}})[31:2] & : otherwise \end{cases} \\ din & = & dest \end{array} \right]\!\!\right]$$

We can prove that a step without any device accesses ignores the given $difo_s$:

Lemma 3.1

$$\begin{array}{l} \forall c_P,\ eev,\ difo1_s,\ difo2_s. \\ \quad \neg\Omega_P(c_P).req \longrightarrow \\ \qquad \Delta_P(c_P, eev, difo1_s) = \Delta_P(c_P, eev, difo2_s) \end{array}$$

The VAMP Specification Run

While running, the ISA executes instructions and communicates with external devices. We model this communication via the input and the output sequences. An input sequence is a mapping from a step number to the processor input. This input consists of an external event (eev) and the data requested by the processor ($difo_s$). We denote by $ISA^n.eev$ the external event vector for the processor step n. Similarly, we denote by $ISA^n.difo_s$ the data from the device to the processor for step $n \mapsto n+1$.

We model a run of the *ISA* as a mapping from a step number to the processor state. This run is defined recursively by the application of the step function Δ_{P} for a given number of steps (or instructions). Running *ISA* from initial configuration c_{P}^{init} for n steps results in the configuration $ISA^n.c_{\mathrm{P}}$.

Definition 3.12

$$ISA^n.c_{\mathrm{P}} \triangleq \begin{cases} c_{\mathrm{P}}^{init} & : n = 0 \\ \Delta_{\mathrm{P}}(ISA^{n-1}.c_{\mathrm{P}}, ISA^{n-1}.eev, ISA^{n-1}.difo_s) & : otherwise \end{cases}$$

The output sequence, which is a mapping from a step number to the processor output, is computed by the application of the output function to the corresponding processor configuration:

Definition 3.13

$$ISA^n.difi \triangleq \begin{cases} difi^\epsilon & : n = 0 \\ \Omega_P(ISA^{n-1}.c_P) & : otherwise \end{cases}$$

where $difi^\epsilon$ is defined as follows:

$$difi^\epsilon \triangleq \left[\!\!\left[\begin{array}{lcl} req & = & 0 \\ w & = & 0 \\ a & = & 0^{30} \\ din & = & 0^{32} \end{array} \right]\!\!\right]$$

The following lemma specifies that the *ISA* model only considers the input from devices at the steps with device accesses. It is a generalization of Lemma 3.1.

Lemma 3.2
Let ISA_v and ISA_s be two ISA models. Let them have equal initial states. Let them receive the same eev inputs during a run, i.e. $\forall i.\ {ISA_v}^i.eev = {ISA_s}^i.eev$. Let ISA_v and ISA_s models receive the same difo inputs for all instructions with device accesses, i.e. $\forall i < I.\ {ISA_v}^{i+1}.difi.req \longrightarrow {ISA_v}^i.difo_s = {ISA_s}^i.difo_s$. Then, these two models have equal runs, i.e. they produce the same states and outputs.

$$\begin{array}{l} {ISA_v}^0.c_P = {ISA_s}^0.c_P \wedge \\ \forall i < I.\ {ISA_v}^i.eev = {ISA_s}^i.eev \wedge \\ \forall i < I.\ {ISA_v}^{i+1}.difi.req \longrightarrow {ISA_v}^i.difo_s = {ISA_s}^i.difo_s \\ \Longrightarrow \\ \forall i \leq I.\ {ISA_v}^i.c_P = {ISA_s}^i.c_P \wedge \\ \forall i \leq I + 1.\ {ISA_v}^i.difi = {ISA_s}^i.difi \end{array}$$

Proof. *First we assume that $\forall i \leq I.\ {ISA_v}^i.c_P = {ISA_s}^i.c_P$ holds. Having this we can prove $\forall i \leq I + 1.\ {ISA_v}^i.difi = {ISA_s}^i.difi$ because difi outputs depend only on previous processor state.*

Now, we prove the assumed statement by induction on the run length and using Lemma 3.1. □

3.2 The VAMP Implementation

Pipelined microprocessors allow the overlapping execution of different instructions. These processors are kind of assembly lines. Every pipeline stage computes a part of the instruction result and passes it on the next stages in the pipeline. A typical pipeline consists of five stages: (i) fetch, (ii) decode, (iii) execution, (iv) memory,

and (v) write back. In the first stage the instruction is loaded from the memory. In the second stage the instruction is decoded, and all its operands are read from the processor's registers. In the third stage the instruction is executed, i.e. the result of the instruction is computed. In the fourth stage memory accesses, if needed, are executed. Finally, the computed result is written back to the register file.

The execution of several instructions can overlap because the different pipeline stages can process different instructions. The pipelined processors can be classified according to the kind of the instruction execution: in-order execution and out-of-order execution (OOO). An in-order processor executes one instruction after the other in the order they are fetched from the memory. The pipeline of such a processor can simultaneously process several instructions. However, if one stage has to be stalled, some or all upper stages have to be stalled as well. For example, the execution stage can be stalled if a memory access requires several cycles, and, hence, the stage fetch and decode are stalled as well. The processors with OOO execution can have a better performance. They can have several data paths in their pipelines, e.g. several function units. Therefore, these processors can execute instructions in a different order than the order they are fetched. For example, if the results of the instructions I_i and I_{i+1} do not depend on each other, the processor may execute instruction I_{i+1} before the earlier fetched I_i.

The VAMP processor is a pipelined processor with OOO execution. In the following subsections we present the VAMP architecture and the description of the VAMP design, and highlight the VAMP main features. We also present the modifications which are needed for the communication between the VAMP and the external devices.

3.2.1 The VAMP Architecture

Figure 3.1 gives an overview of the data path of the VAMP processor [Bey05]. The core of the VAMP is the Tomasulo algorithm [Tom67], which implements out-of-order execution. The VAMP supports a mechanism for precise interrupts. This mechanism requires that if an instruction is interrupted, the results of all previous instructions are written back, but all later instructions are flushed. This mechanism is implemented via a reorder buffer (ROB), which is tailed to the implementation of the Tomasulo algorithm. The goal of this buffer is to guarantee that the instructions are written back in the same order they are fetched.

The main data structure in the Tomasulo implementation is the so-called producer table. This table extends every register in the register file with two additional fields: valid and tag fields. The active value of the valid field signals that there is no instruction in the pipeline which updates the register. Therefore, the content of the register is valid. The inactive value of the valid field signals that there is an instruction in the pipeline which updates this register. In this case, the tag field points to the last such instruction. The tag filed is updated during the instruction issue. The value of this field stays unique until the corresponding instruction leaves the processor.

The instruction fetch does not belong to the Tomasulo algorithm and takes place

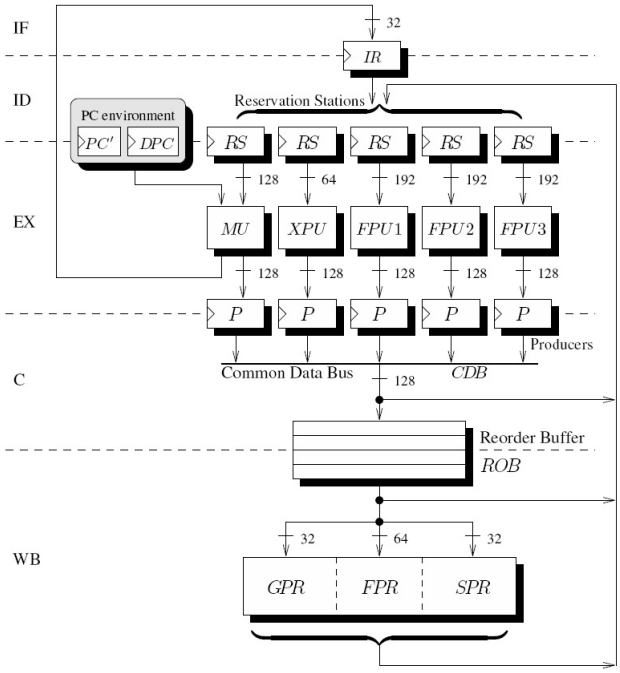


Figure 3.1: Data path of the VAMP processor.

in order. The Tomasulo algorithm starts with the instruction decode, which is directly followed by the instruction issue. During the instruction issue all available operands are read, for all non-available ones the tags from the register file are inserted. These tags are used to find and insert the missing operands. The issue phase is completed by storing the decoded instruction with its operands/tags in a reservation station (RS). Simultaneously with the end of the issue phase, a non-used tag is assigned to the instruction, the valid field of the destination registers of the instruction are set to the inactive value, the tag field receives the instruction's tag, and a new entry in the ROB is allocated for the instruction.

Every instruction stays in the RS until all operands are available, and the target functional unit can accept a new instruction. The missing operands are found by observing (snooping) the common data bus (CDB). The snooping is a simple comparison of the required tag with the tags appearing on the bus. When all operands are available, the instruction can be dispatched out of order to a functional unit. Note that every instruction is always traveling through the pipeline with its tag. Therefore, it can be identified in any pipeline stage.

After an instruction is executed by a functional unit, the instruction and the execution result are moved to the functional unit's output register. These registers are called producers (P) of functional units. Note that the function units can execute instructions out of order but the tags allow the unique identification of instructions.

All producers are connected to the CDB. The CDB has an arbiter which decides which instruction can be completed, i.e. can be moved into the reserved entry in the ROB. When an instruction is moved in the ROB, the result field of the corresponding entry is marked as valid. During the instruction movement in the ROB, the execution result can be forwarded in the reservation stations via the snooping mechanism.

As soon as the oldest instruction result in the ROB becomes valid, it can be written back to the register file. The write back is the last step of the instruction execution. It also signals that the instruction is leaving the processor. A more detailed and formal description of the VAMP implementation of the Tomasulo algorithm can be found in Kröning's PhD thesis [Krö01].

3.2.2 The VAMP Configuration

The VAMP design comprises five functional units: a fixed point arithmetic unit (XPU), three floating point arithmetic units (FPU1– addition/subtraction, FPU2– multiplication/division, FPU3–conversion/testing), and a memory unit (MU). The design and the verification of the FPU's and the XPU are omitted and can be found in [Jac02] and [Krö01] respectively. The memory unit was developed and verified by Dalinger [Dal06]. In Section 3.2.5, we briefly describe the structure of the MU and present the changes due to the support of the external devices.

The VAMP implementation of the Tomasulo algorithm has eight entries in reservation station. Every entry in a RS is constantly assigned to the corresponding functional unit. Four entries are fixed for the XPU, one entry for every FPU, and one entry for the MU. There are several instructions which do not compute anything, e.g. a jump or a data movement inside the register file. They can be directly issued in the ROB bypassing the functional units. However, these instructions have to be stalled in the decode stage until all their operands are available, because they can not snoop on the CDB.

The instructions in the VAMP have up to six operands, e.g. the double precision floating point operations and the memory operations. The result of the instruction execution consists of up to four 32-bit words: two words due to the double precision FPU result, one word for the exception causes (CA), and one word for the exception data (EData). The latter two words are ignored as long as no interrupts occur. In case of an interrupt they are used in the computation of the next state of the *SPR* [Bey05, Section 4.1].

The VAMP register file consists of three register banks which are extended by the producer tables. These banks are the general purpose registers (*GPR*), the special purpose registers (*SPR*), and the floating point registers (*FPR*). The *FPR* can be accessed as 16 64-bit registers or as 32 32-bit registers.

The fetch mechanism implements the delayed-*PC* architecture [MP00] where all *PC* computations are delayed by one cycle. This implies that the instruction, which is followed directly after a branch instruction, is executed before the branch destination. The fetch result, i.e. the new instruction, is stored in the register $S1.IR$. There can be two exceptions during the fetch: the instruction misalignment and the instruction

page fault. They are stored in $S1.imal$ and $S1.ipf$ respectively.

The interrupts are split into two groups: the internal interrupts (e.g. *imal*, overflow) and the external ones (e.g. reset, external devices). The internal interrupts are part of the instruction result, namely CA. The external interrupts are an additional input bus of the VAMP. In previous work this bus is always sampled at the instruction write back [Bey05]. In the next section we show that sampling at the write back leads to an unclear semantics of the processor-device communication. We present an elegant solution for this problem. If during the instruction execution an interrupt (exception) occurs, the processor does not handle it at once but at the interrupt handling point. The VAMP considers the interrupts at the write back of an instruction. If an interrupt has to be served, the VAMP raises the signal *JISR* (jump to the interrupt service routine). If *JISR* is raised, the VAMP cancels any computations by resetting all implementation registers (the RS, the ROB, the functional units, and the producers). The processor also marks all registers in the register file as valid. Additionally, the program counters are reset to the initial values. These values are pointing to the beginning of the interrupt service routine.

To summarize, the processor implementation configuration $h_P \in H_P$ is 16-tuple

$$\begin{aligned} h_P \triangleq \quad & (PC, DPC, GPR, FPR, SPR, S1, RS, P, ROB, \\ & ROBhead, ROBtail, ROBcount, MU, FPU1, FPU2, FPU3) \end{aligned}$$

The *ROBhead* and the *ROBtail* are used to point at the head and the tail of the reorder buffer respectively [SP88]. The *ROBcount* contains a current number of the reserved entries in the ROB. This counter is used to distinguish between the empty and the full ROB since in these cases $h_P.ROBhead = h_P.ROBtail$. We do not have any component for the XPU since it is combinational.

Note that we do not have any component for the memory. The memory is not a part of the processor implementation but it is rather an external component. However, we can define the content of the memory by considering the memory interfaces of the processor and the memory initial state [Bey05, Section 1.5]. Before we give this definition we define the VAMP communication interfaces.

3.2.3 The VAMP Communication Interfaces

The VAMP can communicate with two "off-the-chip" components: the external devices and the memory. The processor employs the following communication buses (Figure 3.2):

mifi – memory interface input; the processor puts the memory requests on this bus; this is a processor output

mifo – memory interface output; the processor receives the memory answers on this bus; this is a processor input

difi – device interface input; the processor puts the device requests on this bus; this is a processor output

difo – device interface output; the processor receives the device answers on this bus; this is a processor input

eev – external event vector; the devices signal their interrupts on this bis; this is a processor input

Note that the naming convention is from the memory/devices point of view.

We model inputs to the VAMP as mappings from hardware cycles to the corresponding data. We use the notation $VAMP^t.mifo$, $VAMP^t.difo$, and $VAMP^t.eev$ to denote the states of the memory, the device buses, and the external event bus at cycle t respectively. Similarly, we employ the notation $VAMP^t.mifi$, and $VAMP^t.difi$ to denote the state of the VAMP outputs at cycle t. In Section 3.2.4 we formally define these outputs.

The processor-memory communication obeys a bus protocol [MP00, Bey05]. The processor places its request on $mifi \in Mifi$, which is modelled as the following tuple:

$$mifi \triangleq (req, w, a, din, bwb, burst)$$

The processor accesses the memory by setting $mifi.req$. The accessed address is set on $mifi.a \in \mathbb{B}^{29}$. The component $mifi.bwb \in \mathbb{B}^8$ specifies the type of the access. It defines which bytes of the memory cell should be written. If all bits of $mifi.bwb$ have inactive values, the processor makes a read access. The component $mifi.din \in \mathbb{B}^{64}$ provides the data for the write access. The boolean flag $mifi.burst \in \mathbb{B}$ signals that during this request several words will be read/written. The memory answers are placed on $mifo \in Mifo$:

$$mifo \triangleq (reqp, brdy, dout)$$

The memory signals by setting $mifo.reqp \in \mathbb{B}$ that it is processing a request and can not accept a new one. The active value of $mifo.brdy \in \mathbb{B}$ signals that in the next cycle the request is finished and the data on $mifo.dout \in \mathbb{B}^{64}$ will be valid. We say that the memory interface is busy if and only if *reqp* holds or *brdy* holds, i.e. $busy(mifo) \triangleq mifo.reqp \vee mifo.brdy$.

The *mifi*-bus logically and physically splits the processor and the memory chip into two independent parts. It also introduces one cycle delay in the communication between the VAMP and the memory: the VAMP places the request data and in the next cycle, in the best case, receives an answer from the memory. The above described bus-protocol guarantees that these parts can correctly communicate with each other [MP00, Bey05].

The communication with devices is done in a similar fashion. The processor places a request on *difi* and expects the result on *difo*. We model *difi* in the same manner as for the specification (Section 3.1). The channel $difo \in Difo$ is modelled similarly to *mifo*, i.e.

$$difo \triangleq (reqp, brdy, dout)$$

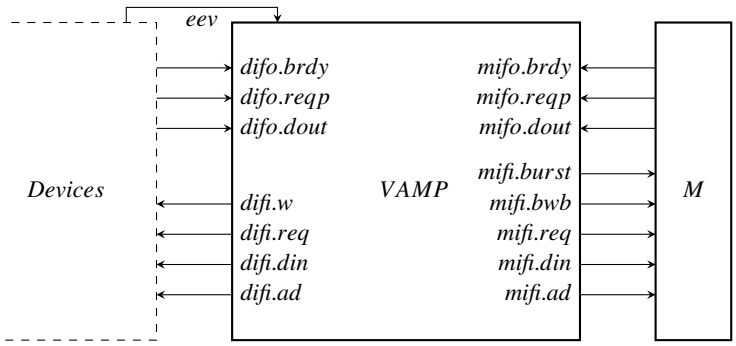


Figure 3.2: The external interfaces of the VAMP.

with $difi.dout \in \mathbb{B}^{32}$. We define $difo^{\epsilon} \in Difo$ as the idle value with $\neg difo^{\epsilon}.reqp$, $\neg difo^{\epsilon}.brdy$, and $difo^{\epsilon}.dout = 0^{32}$.

We introduce an extra notion to denote the busy state of the device interface buses (*difi* and *difo*), i.e. *difi* and *difo* are busy if and only if *reqp* holds or *brdy* holds:

Definition 3.14

$$busy(difo) \triangleq difo.reqp \vee difo.brdy$$

We slightly extend the bus protocol to support devices which produce results already at the next cycle after the request. Thus, these devices set neither *reqp* nor *brdy* signals. An example of such a device is an automotive bus controller [KP07]. Therefore, if at req^t holds and $busy^{t+1}$ is inactive, the request is finished and the data $dout^{t+1}$ must be valid. We introduce a signal *da_end* which holds at the cycle when a device access is finished:

Definition 3.15

$$\begin{aligned} da\_end(VAMP, t) \triangleq\ & (busy(VAMP^{t-1}.difo) \wedge \neg busy(VAMP^{t}.difo.)) \vee \\ & (VAMP^{t-1}.difi.req \wedge \neg busy(VAMP^{t}.difo)) \end{aligned}$$

The Figure 3.3 and Figure 3.4 show typical processor-device communications.

There are special states of the VAMP processor which we call initial configurations. In these states all internal registers (the RS and ROB registers, the functional units, and the producers) are reset. All registers in the register file are marked as valid and the program counters have some initial values. Note that these states are similar to the

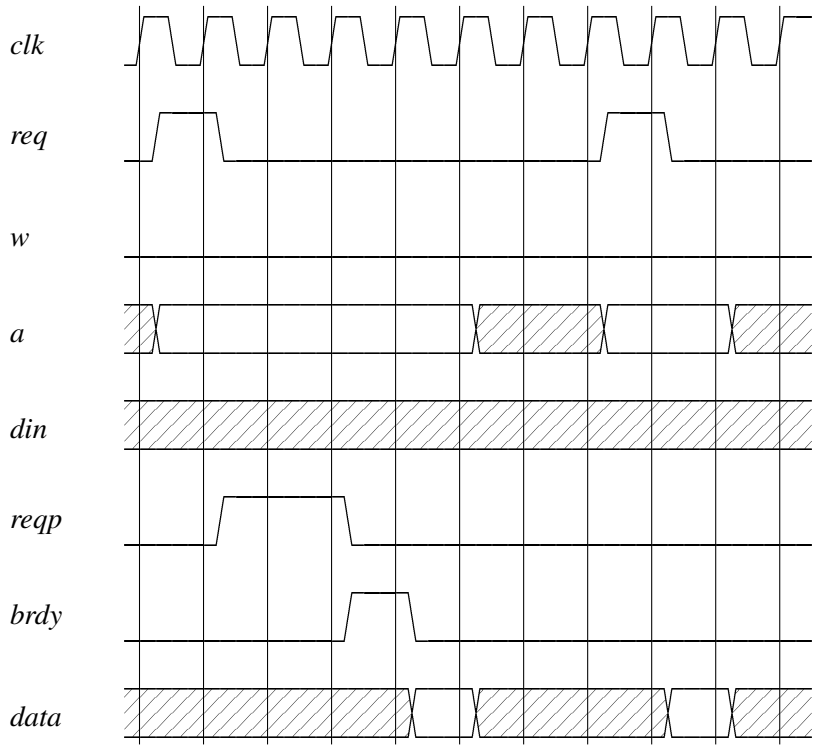


Figure 3.3: Timing of the device interface: read and fast read accesses.

one after a jump to the interrupt service routine is done, i.e. one cycle after the *JISR* signal is raised.

3.2.4 The VAMP Implementation Run

Now we define the VAMP processor as a Moore automaton. The next step function δ_P computes the next state $h'_P = \delta_P(h_P, mifo, difo, eev, reset)$ for a given processor state h_P and the states of the input buses. The output function ω_P, for a given processor state h_P, computes the state of the interface input buses: $(mifi, difi) = \omega_P(h_P)$.

A run of such an automaton, for a given number of cycles t, is defined by the recursive application of the step function:

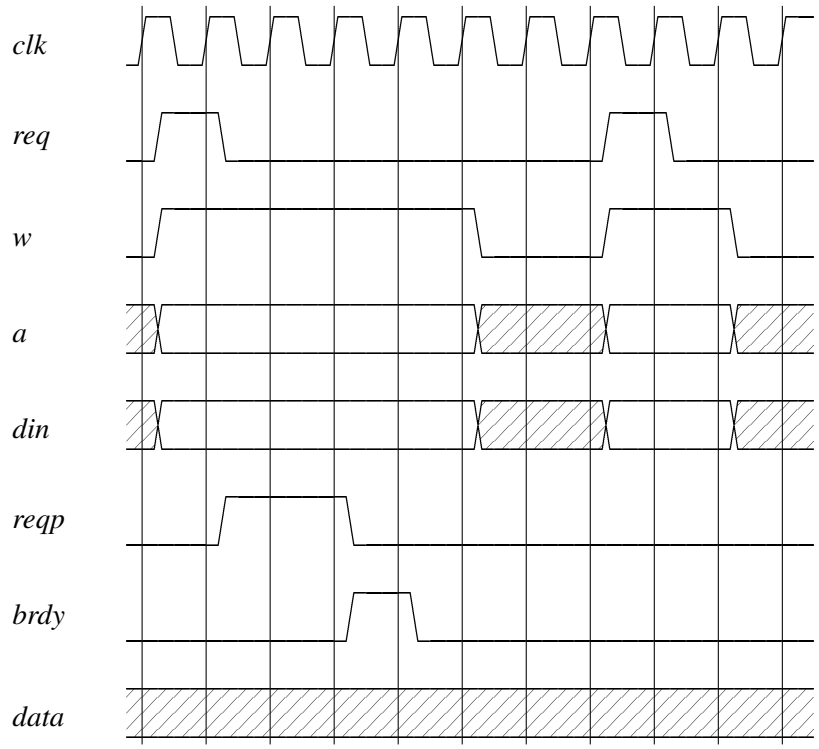


Figure 3.4: Timing of the device interface: write and fast write accesses.

Definition 3.16

$$VAMP^t.h_{\mathrm{P}} \triangleq \begin{cases} h_{\mathrm{P}}^{init} & : t = 0 \\ \delta_{\mathrm{P}}(VAMP^{t-1}.h_{\mathrm{P}}, VAMP^{t-1}.mifo, & \\ \quad VAMP^{t-1}.difo, VAMP^{t-1}.eev, VAMP^{t-1}.reset) & : otherwise \end{cases}$$

For the rest of the thesis we assume that the reset signal is never set: $\forall t.\neg VAMP^t.reset$.

The outputs of the VAMP are computed by the application of the output function:

Definition 3.17

$$(VAMP^t.mifi, VAMP^t.difi) \triangleq \begin{cases} (mifi^\epsilon, difi^\epsilon) & : t = 0 \\ \omega_{\mathrm{P}}(VAMP^t.h_{\mathrm{P}}) & : otherwise \end{cases}$$

where $mifi^\epsilon$ and $difi^\epsilon$ are some idle values.

We define the state of every cell of the external memory solely by observing the memory interfaces [Bey05, Chapter 4]. However, we have to assume that the physical memory[3] (i) can only be changed by the processor (no DMA requests by the devices) and (ii) is correct (e.g. it keeps the saved data).

Definition 3.18
Let *init_mem* be the initial memory content. We introduce a function $bw(t, dold) \in \mathbb{B}^{64}$ which for a given cycle $t \in \mathbb{N}$ and a given old content of the memory cell $dold \in \mathbb{B}^{64}$ computes the value to be written in the memory, i.e.

$$bw(t, dold) \triangleq byteupd(VAMP^t.mifi.bwb, dold, VAMP^t.mifi.din)$$

The memory content $M(VAMP, t)$ at cycle t is recursively defined as follows:

$$M(VAMP, t)(a) \triangleq \begin{cases} init\_mem(a) & : t = 0 \\ bw(t, M(VAMP, t-1)(a)) & \\ & : VAMP^{t-1}.mifi.a = a \wedge \\ & \quad \neg busy(VAMP^{t-1}.mifo) \\ M(VAMP, t-1)(a) & \\ & : otherwise \end{cases}$$

We use this definition as the specification for the implementation of the memory system.

3.2.5 The VAMP Memory Unit

The memory unit (MU) executes all memory and device accesses. It also manages the external interfaces. The MU implements two independent functionalities: the instruction fetch and the memory/device accesses.

The instruction fetch is based on the given $h_P.DPC$. After the possible address translation (via MMU), the fetch request is forwarded to the cache system of the VAMP. Note that the cache system provides the fetch and the data interfaces; it consists of two interconnected caches and communicates with the external memory [Bey05, Mül07]. When a fetch request is finished[4], the result and the possible exceptions are saved in the VAMP fetch stage (the IF stage in Figure 3.1). A memory access is executed similarly. The difference is that the access result is saved in the MU's producer.

A new feature of the MU is the support of the accesses to the external devices. The external devices are mapped into the memory address space, i.e. they can be accessed via the regular load and store instructions. Recall that we place the device address space above the address space of the physical memory. Every address which starts with 17 ones belongs to the device addresses.

$$IsDevAddr(ad) \triangleq ad[31:15] = 1^{17}$$

[3] The chip which implements memory.

[4] The request can last one cycle, if the fetch result has been previously cached.

The rest of a device address (15 bits) is used to specify a particular device and its registers (Figure C.2). If a device access has to be executed, MU places the request on the above introduced *difi* interface, and the access result is expected on the *difo* interface. Note that all device accesses bypass the cache system, because the devices can progress independently of the processor, and hence, the cached data may become out-dated.

The support of the precise interrupts requires: if an instruction is interrupted, the effect of all later fetched instructions has to be rolled back. However, if instruction i is interrupted, instruction $i + n$ could already modify the memory. This modification cannot be rolled back. This situation can be avoided [BJK+05]. One starts a write access if and only if the corresponding instruction is the oldest one in the pipeline. The same problem arises for all device accesses because both read and write accesses can change states of accessed devices. Therefore, every device access has to be stalled until the corresponding instruction becomes the oldest one in the pipeline.

External Interrupt Sampling

Recall that the VAMP handles the internal interrupts at the instruction write back stage. The old VAMP samples the external interrupt bus (the state of the *eev* bus) at the write back as well [BJK+05]. However, the sampling at the write back may be a problem for the processor-device communication. Let us consider the following scenario.

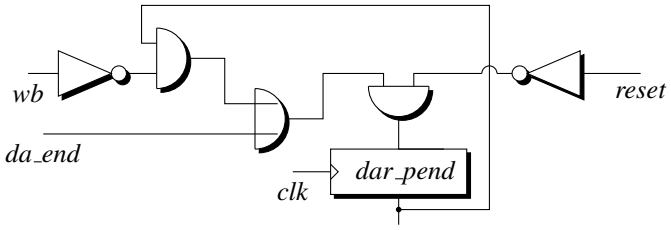
An instruction accesses a device, and its goal is to deactivate the device interrupt. Let us assume that at cycle t the device signals the end of the request, i.e. the interrupt is deactivated. At cycle t the instruction is still in the MU. The instruction path to the write back stage takes several cycles. Meanwhile the device can progress and may activate the interrupt. Thus, at the instruction write back the VAMP samples this re-activated interrupt. However, from the assembly programmer point of view this interrupt belongs to, or should affect, the next instruction.

Thus, the sampling of *eev* at the write back does not allow a precise ordering of external interrupts with instructions.

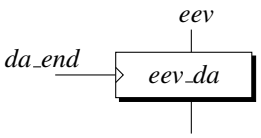
Let us consider all possible device accesses of the VAMP. The processor allows three sorts of device accesses:

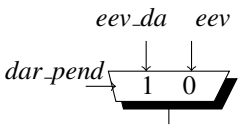
- an active read access – a load instruction with a device address
- an active write access – a store instruction with a device address
- a passive read access – the sampling of the external interrupts

This interpretation reveals the fact that *every* instruction, with external interrupts enabled, executes at least one device access. The instructions without active device accesses passively access devices at the write back stage. The instructions with a device access do it *twice*: the first time in the MU and the second time in the WB



(a) Computation of the device-access result pending





(b) The external event vector for device access (c) The interrupt selection in the WB stage

Figure 3.5: The extensions of the MU and the WB stage due to the support of the precise interrupts.

stage. Moreover, these accesses are spread over time. Every instruction must access the external devices only once to avoid any "shadow" scenarios.

Obviously, the instructions without any active device access already satisfy this requirement. We change the sampling of the external interrupts for all instructions with an active device access: we sample them exactly at the time when the device places the result on the bus.[5] This guarantees that the result of a device access and the external interrupts are read at the same cycle. We also add a test in the WB stage. If an instruction without *any* active device access leaves the processor, we use the current state of the *eev* bus. Otherwise, we use the interrupts which have been sampled in the MU stage. This solution requires several small modifications of the memory unit and the write back stage.

Memory unit and write back stage extensions

We introduce three auxiliary notations:

- da_end^t – signals the end of a device access at cycle t, if any access is pending (Definition 3.15)
- wb^t – signals the write back of an instruction at cycle t
- eev^t – represents the state of the external event bus at cycle t
- $reset^t$ – signals reset at cycle t

[5] Smith and Pleszkun [SP88] proposed to sample external interrupts before instruction issue. MIPS-R3000 Family [Brü91] samples the interrupt before the memory stage. These solutions still allow instructions accessing the external devices twice.

We add to the MU stage a 1-bit register *dar_pend* (device-access result pending) and a 19-bit register *eev_da* (external event vector for device access). The register *dar_pend* contains the value 1 if:

- the accessed device produced the result for a processor request, i.e. the device access is finished
- this result is not yet written back, i.e. the corresponding instruction is not yet in WB stage

Recall that an instruction can only start a device access if it is the oldest instruction in the pipeline. Thus, the next write back signals that such an instruction is leaving the processor. This allows an easy determination of the cycles when *dar_pend* should be set up. We set *dar_pend* between the request end and the write back of the result of the device request (*wb*): The necessary hardware is presented in Figure 3.5:

$$dar\_pend^{t+1} \triangleq ((\neg wb^t \wedge\ dar\_pend^t) \vee da\_end^t) \wedge \neg reset^t$$

Additionally, we save the state of the *eev* bus at the request end in *eev_da*. The value of this register is kept stable at least as long as the instruction is not written back, i.e. as long as *dar_pend* is set. Thus, we only update the register at the end of device accesses:

$$eev\_da^{t+1} \triangleq \begin{cases} eev^t & : da\_end^t \\ eev\_da^t & : otherwise \end{cases}$$

The last modification is in the WB stage. We insert a multiplexer which selects either the current state of the *eev* bus or the interrupts which have been saved in *eev_da*. The control signal of this multiplexer is *dar_pend*, which is provided by the memory unit. After reset we initialise the register *dar_pend* with zero and the initial value of *eev_da* does not matter.

Correctness of the new hardware

The signal *dar_pend* can only be activated between the request end and the next write back:

Lemma 3.3

$\forall t.\ dar\_pend^t \longrightarrow \exists t1 < t.\ da\_end^{t1} \wedge \forall t2 \in [t1 : t[.\ \neg wb^{t2}$

If the signal *dar_pend* is set, the state of the register *eev_da* equals the state of the *eev* bus at the end of the last device access. If the signal *dar_pend* is not set, the state of the register *eev_da* does not matter.

Lemma 3.4

$\forall t.\ dar\_pend^t \longrightarrow eev\_da^t = eev^{last_{hw}(t,da\_end)}$

where $last_{hw}(t, da\_end)$ *denotes the last cycle below t when da_end hold.*

We have proven both lemmata automatically, by the tool chain which we have presented in Section 2.4.

Beyer has proven that the signal *JISR* and running memory write access cannot be active at the same cycle (See [Bey05, p.156, Lemma 4.5.4]). Since all device accesses are implemented in the same way as memory writes, we can derive that the signal *JISR* and the end of a device access cannot be active simultaneously. Moreover, since *JISR* implies *wb* we formulate this property as follows:

Lemma 3.5

$\forall t.\ \neg(wb^t \ \wedge da\_end^t)$

3.3 The VAMP Correctness Criterion

A typical correctness criterion for pipelined machines is a simulation relation. The criterion states that every run of the implementation can be simulated by a run of a specification with respect to the selected simulation relation. Thus, a simulation relation relates the time and the configurations between the implementation and the specification models. We use the scheduling function concept [SH98, MP00] to relate the time notion of both models. The comparison of the configurations is based on the concept of the programmer visible registers.

In the following subsections we develop the VAMP correctness criterion with respect to its sequential specification, the ISA model. We define the simulation relation. Since the VAMP and the ISA communicate with external devices, we define a relation which synchronizes these inputs.

3.3.1 State Relation

The VAMP implementation has more registers than the ISA specification. The registers, which are present in both models, are called the *visible registers*. This name is given from the programmer's point of view. Examples of these registers are the program counters (*PC* and *DPC*), and the general purpose registers (*GPR*). All other implementation registers are called the *invisible registers*. They are used to save the partial results of the instruction execution, e.g. the internal registers of the functional units. We define a predicate *Rconf* which relates the visible registers of the VAMP processor and the corresponding components of the ISA model.

Definition 3.19

$$
\begin{array}{lcl}
Rconf(h_{\mathrm{P}}, c_{\mathrm{P}}) & \triangleq & \\
c_{\mathrm{P}}.PC & = & h_{\mathrm{P}}.PC \\
c_{\mathrm{P}}.DPC & = & h_{\mathrm{P}}.DPC \\
c_{\mathrm{P}}.GPR & = & \lambda x.\ h_{\mathrm{P}}.GPR[x].data \\
c_{\mathrm{P}}.FPR & = & \lambda x.\ h_{\mathrm{P}}.FPR[x].data \\
c_{\mathrm{P}}.SPR & = & \lambda x.\ h_{\mathrm{P}}.SPR[x].data
\end{array}
$$

Note that we consider memory as part of the correctness theorem which we present later in this chapter.

3.3.2 Scheduling Function

A scheduling function maps a given hardware cycle to the number of instructions, which have been processed by the processor so far. Thus, one can consider a scheduling function as an instruction counter. The definition of a scheduling function is based on some special hardware events. These events signal the progress of the instruction execution. For example, the event *write back* signals that an instruction has been completely executed and leaves the processor. Thus, the write back of an instruction causes irreversible changes on the processor state, e.g. update of the *GPR*.

We use the scheduling function *sI* which is based on two processor events: the write back (*wb*) and the jump to interrupt service routine (*JISR*). Both events signal that an instruction leaves the processor. Signal *JISR* implies *wb* and has an additional effect: it forces the processor to drain (or to flush) the pipeline since an interrupt has occurred. Hence, the processor has to abort any computations and execute an interrupt service routine. The scheduling function is defined recursively on the hardware cycles:

Definition 3.20

$$sI(t) \triangleq \begin{cases} 0 & : t = 0 \\ sI(t-1) + 1 & : wb^{t-1} \vee JISR^{t-1} \\ sI(t-1) & : otherwise \end{cases}$$

An interesting property of *sI* is that its result can be interpreted in two different ways: (i) $sI(t)$ is the index of the next instruction to be written back, where the first instruction has index 0, or (ii) $sI(t)$ is the number of instructions which are completely processed by the processor, i.e. left the processor.

It is easy to prove that *sI* is a monotonic function.

Lemma 3.6

$$t < t' \Longrightarrow sI(t) \leq sI(t')$$

The definition of *sI* guarantees that if there were no write backs during some interval, this interval does not affect the result of *sI*.

Lemma 3.7

$$\forall t'' \in [t : t'[\,.\, \neg wb^{t''} \Longrightarrow sI(t) = sI(t')$$

We can also prove that for every counted instruction there is a hardware cycle when it has been written back.

Lemma 3.8

$$\forall i < sI(T).\ \exists t < T.\ wb^t \wedge i = sI(t)$$

This scheduling function is suitable for a correctness criterion of *GPR*, *SPR*, and *FPR* register files at any hardware cycle and for a correctness criterion of *PC*, *DPC*, and memory for drained pipeline. However, if one wants to state anything about the invisible registers, one needs several scheduling functions: one scheduling function per pipeline stage ([MP00, Section 4.5.2]; [Krö01, Section 6.5]). For example, if we run the ISA model for $i = sI(t)$ steps, the program counter $ISA^i.DPC$ will point at the instruction i. However, the program counter in the implementation may point to some instruction $i+n$ (with $n \geq 0$) due to the pipelined execution. Examples of a correctness criterion with all implementation registers can be found in [MP00, Krö01, Bey05].

3.3.3 Inputs/Outputs from/to External Environment

The ISA model and the VAMP implementation communicate with the external environment. The runs of these models can only be compared if the models receive equivalent inputs. Now, we define two predicates which relate input sequences for the VAMP and the ISA models.

The relation *Reev* relates the input sequences of the external interrupts. In the implementation we consider the external interrupts at the write back of an instruction. These interrupts are either the data saved in MU register *eev_da* or the current state of the external interrupt bus.

Definition 3.21

$$\begin{aligned}
&Reev(T, VAMP.eev, ISA.eev) \triangleq \\
&\quad \forall i.\ \forall t < T.\ t = (The\ x.\ i = sI(x)\ \wedge\ (wb^x \vee\ JISR^x)) \longrightarrow \\
&\quad ISA^i.eev = \begin{cases} VAMP^t.h_P.eev_da & :\ VAMP^t.h_P.dar_pend \\ VAMP^t.eev & :\ otherwise \end{cases}
\end{aligned}$$

where $(The\ x.\ sI(x) = i \wedge (wb^x \vee JISR^x)$ specifies the hardware cycle when instruction i leaves the processor.

We define the predicate *Rdifo* which relates the answers from the external devices. These data have to be only related at the end of the processor-device interaction. Recall that a device access can only be started if this instruction is the oldest in the pipeline, i.e. it is the next instruction to be written back. We also know that $sI(t)$

specifies the index of the next instruction to be written back. Thus, if we started a device access at some cycle t, $sI(t)$ specifies the index of the instruction in MU. This fact has been proven by Beyer [Bey05][6] for the instructions with memory write access, and we can reuse this fact because all device accesses are implemented in the same way.

Definition 3.22

$$
\begin{aligned}
&Rdifo(T, VAMP.difo, ISA.difo_s) \triangleq \\
&\quad \forall i.\ \forall t < T.\ t = (The\ x.\ i = sI(x)\ \wedge da\_end(VAMP, x) \longrightarrow \\
&\quad ISA^i.difo_s = VAMP^t.difo.data
\end{aligned}
$$

This definition relates the data which appear on the *difo*-bus at cycle t with the data designated for the instruction i.

The ISA and the VAMP models produce outputs for the external devices. We define predicate *Rdifi* to relate these outputs. We derive the definition for this predicate from the correctness criterion of the outputs of the memory unit [Bey05, Dal06]. We can reuse this correctness criterion, because the external devices are mapped in to the memory address space.

The predicate *Rdifi* consists of three conjuncts:

- if there is a request in *VAMP*, the output data *difi* of *VAMP* must match the output data *difi* of *ISA*,
- if a written back instruction accesses a device in the *ISA* model, there must be a hardware cycle when *VAMP* started the corresponding device access,
- if we write back an instruction with a device access, there must be a prior hardware cycle when *VAMP* started device access with correct data.

Definition 3.23

$$
\begin{aligned}
&Rdifi(T, VAMP.difi, ISA.difi) \triangleq \\
&\quad (\forall t \leq T.\ VAMP^t.difi.req \longrightarrow VAMP^t.difi = ISA^{sI(t)+1}.difi) \wedge \\
&\quad (\forall i < sI(T).\ ISA^{i+1}.difi.req \longrightarrow \\
&\qquad \exists t \leq T.\ i = sI(t)\ \wedge VAMP^t.difi = ISA^{i+1}.difi) \wedge \\
&\quad (wb^T \wedge ISA^{sI(T)+1}.difi.req \longrightarrow \\
&\qquad \exists t \leq T.\ VAMP^t.difi = ISA^{sI(T)+1}.difi \wedge sI(T) = sI(t))
\end{aligned}
$$

[6] See [Bey05, p. 155] first case of the proof for Lemma 4.5.3.

Note that the VAMP keeps the data on the bus stable during the whole request (see [Bey05, p. 145]).

Lemma 3.9

$$VAMP^t.difi.req \wedge t' = next_{hw}(t, da\_end)$$
$$\Longrightarrow$$
$$\forall t'' \in [t : t'].\ VAMP^t.difi.\{w, a, din\} = VAMP^{t''}.difi.\{w, a, din\}$$

where $next_{hw}(t, da\_end)$ *denotes the next cycle after t when da_end holds.*

3.3.4 Software Conditions

Sometimes it is impossible or too expensive to implement the handling of some exceptions in the hardware. These special cases restrict the software which can be executed on the developed hardware. We call these restrictions *software conditions* [BJK+05, DHP05]. We present three software conditions for the VAMP processor:

- software guarantees for the self-modifying code, which can generate the read-after-write (RAW) hazards
- software guarantees for the address translation
- software guarantees for accesses to the defined memory address space

Beyer et al. [BJK+05] proposes a software condition which excludes the RAW hazards for the self-modifying assembly code. Dalinger [Dal06] extends this condition with the support of the virtual memory. This software condition is based on the so-called *sync*-instruction. The execution of the sync-instruction should require the write back of all instructions in the pipeline, i.e. draining the pipeline. Thus, the sync-instruction should stall the fetch of the new instructions as long as the pipeline is not empty. Moreover, such an instruction should not modify the semantics of the program. Hence, the execution effect of the sync-instruction has to be equivalent to the one of a *nop* instruction. In the VAMP there are two instructions with a sync-semantics: *movs2i IEEEf R0*, *rfe*, and any interrupted instruction (an instruction which causes *JISR*).[7] The first software condition requires that between any two instructions which produce a RAW hazard there is at least one sync instruction. Dalinger presents a formal definition of the software condition [Dal06, p. 76]. We describe it informally here:

Definition 3.24 (*synced_code_i*?)
The assembly code is synced if:

[7] Hillebrand [Hil05, Section 5.1.4] observed that *rfe* and interrupted instructions can be used as a sync-instruction.

1. whenever the processor operates in the system mode (i.e. $\neg vmode(c_P)$) there is a sync instruction between any fetch from and the last modification of a physical address $a \in MA$,

2. whenever the processor operates in the user mode (i.e. $vmode(c_P)$) there is a sync instruction between any fetch of a translated address $a \in MA$ with the corresponding page table entry address $ptea \in MA$ and the latest modification among ad and $ptea$.

Usually, modern processors contain translation look-side buffers (TLB). A TLB caches the recently used translations to speed-up the following translations. If the processor allows that the cached data in the TLB may become inconsistent with the data in the memory, then we need an additional software condition. The software condition *stable_PT*? eliminates this inconsistency by prohibiting the assembly programs, which are executed in the user mode, from modifying the page table entries. We introduce this condition because the next version of the MU of the VAMP contains TLBs.[8]

Definition 3.25 (*stable_PT*?)
Let PT be the set of addresses which belong to the page table, i.e. the $PTL + 1$ number of addresses starting from the page table origin (PTO):

$$PT(c_P) \triangleq \{x \mid c_P.SPR[PTO] * 4096 \leq x \wedge\ x \leq c_P.SPR[PTO] * 4096 +_{32} c_P.SPR[PTL]\}$$

Then

$$\forall i.\ vmode(ISA^i.c_P) \wedge\ mw?(ISA^i.c_P) \longrightarrow$$
$$decodeitr(ISA^i.c_P, mw?(ISA^i.c_P), ea(ISA^i.c_P)).pa \notin PT(ISA^i.c_P)$$

In the presence of external devices, we need an additional software condition which guarantees the absence of accesses to the undefined address space. The main issue is the liveness because neither memory nor devices respond to an access to the undefined address space, and hence, such an access never terminates.

Definition 3.26 (*def_addr*?)
For a defined device address space DA, all address of memory instructions have to be either in MA or in DA.

$$\forall i.\quad \neg\ dmal(c_P)\ \wedge\ \neg\ dpf(c_P)\ \wedge (lw?(ISA^i.c_P) \vee sw?(ISA^i.c_P)) \wedge$$
$$(\neg\ vmode(c_P)\ \longrightarrow ea(c_P)[31:3] \in MA \vee ea(c_P)[31:2] \in DA) \wedge$$
$$(vmode(c_P)\ \longrightarrow dpa(c_P)[31:3] \in MA \vee dpa(c_P)[31:2] \in DA)$$

[8] We experienced that the new MU is roughly three times faster than the previous version [Dal06]. This new version is currently being verified by Dalinger and Alekhin [Ale08].

3.3.5 The VAMP Correctness Theorem

The correctness criterion of the VAMP can be split into two parts: a criterion for non-interrupted runs and a criterion for interrupted runs [Bey05, Dal06, Krö01]. The former criterion is very strong and it states the correctness of *every* implementation register, e.g. [Krö01, Section 6.7], [Bey05, Section 4.3.2]. However, since the run of any real processor contains interrupts, this criterion can not be used as *the* correctness criterion. The reason is that some processor components, e.g. program counters, may have values which never occur in the specification with interrupts. Therefore, *the* correctness criterion of the VAMP is stated after every interrupt [Bey05, Dal06]. This criterion covers *all* visible registers, and it also allows us to use only one scheduling function.

Even though we verified the criterion for non-interrupted runs in Isabelle/HOL, we do not present it in this thesis. This is because, this criterion is only used as an auxiliary lemma in the verification of the criterion for the interrupted runs. The proof of the criterion for non-interrupted runs is based on the PVS proofs which are described in [Krö01, Bey05, Dal06].

Theorem 3.10
Let the assembly code fulfill the software conditions. Let the inputs of the specification and the implementation models be synchronized with respect to Reev and Rdifo. Let also the initial states of the implementation and the specification be in the simulation relation.

We show that the state of the VAMP processor is in the simulation relation with the ISA state after every interrupt.

$$
\begin{aligned}
&\mathit{synced\_code\_i}? \wedge\ \mathit{stable\_PT}? \wedge\ \mathit{def\_addr}? \wedge \\
&\mathit{Reev}(T, \mathit{VAMP.eev}, \mathit{ISA.eev}) \wedge\ \mathit{Rdifo}(T, \mathit{VAMP.difo}, \mathit{ISA.difo}_s) \wedge \\
&\mathit{Rconf}(T, \mathit{VAMP}^0.h_P, \mathit{ISA}^0.c_P) \wedge M(\mathit{VAMP}, 0) = \mathit{ISA}^0.c_P.M \\
&\Longrightarrow \\
&\quad (\mathit{JISR}^{T-1} \longrightarrow \mathit{Rconf}(\mathit{VAMP}^T.h_P, \mathit{ISA}^{sI(T)}.c_P)\ \wedge M(\mathit{VAMP}, T) = \mathit{ISA}^{sI(T)}.c_P.M) \wedge \\
&\quad \mathit{Rdifi}(T, \mathit{VAMP.difi}, \mathit{ISA.difi})
\end{aligned}
$$

The proof for this theorem assembles the proofs presented by Kröning [Krö01], Beyer [BJK+05], and Dalinger [Dal06]. The proof strategy consists of several steps. First, we show the correctness of the implementation of the Tomasulo algorithm. Then, we introduce an intermediate correctness for the VAMP processor *without* interrupts and prove that if there are no interrupts, the VAMP fulfills the correctness criterion. As the next step, we show that a non-interrupted run followed by an interrupt implies the VAMP correctness criterion. Finally, we prove that every run of the VAMP processor can be composed of sub-runs, which consist of a non-interrupted run followed by a step with an interrupt. We have applied induction over the number of such sub-runs.

For the detailed description of the proof, we refer the reader to the following sources:

VAMP(no FPUs, MU)	Person years	Theorems	Proof steps
in Isabelle	1.5	1206	20455
in PVS	3	966	37666

Table 3.1: Verification efforts: PVS vs. Isabelle/HOL+IHaVeIt.

- the proof of the implementation of the Tomasulo algorithm was described by Kröning [Krö01]
- Beyer [Bey05] instantiated the Tomasulo implementation and added user visible memory
- Dalinger [Dal06] proved correctness of the VAMP processor with address translation

3.4 Summary

In this chapter, we presented a new more advanced version of the VAMP processor. The previous version of the VAMP was extended with a *full* support of the external devices. The new version was considered as a part of a computer system, in contrast to previous works where the VAMP was considered as a stand-alone system. This point of view, together with the formal verification of the combined system (Chapter 5), allowed us to establish a clean semantics of the external interrupts.

Recall that the original VAMP project is carried out in the interactive theorem prover PVS. Now, we compare the verification efforts in PVS and in Isabelle/HOL (Table 3.1 [9, 10]). This comparison can not be 100% "fair", because (i) the nature of the systems is too different, and, in our opinion, the PVS proof commands are more powerful than those of Isabelle, (ii) our work in Isabelle/HOL is based on the proof strategy developed in PVS. The second argument explains why we could finish the proofs in less time.

We believe that the most fair comparison would be the comparison of the size of the proofs, i.e. the number of interactive commands applied by the user. Recall that the proofs in PVS are purely interactive ones, and the proofs in Isabelle/HOL are partially automated. Therefore, such a comparison gives an idea how much user work

[9] We developed and verified the VAMP model which has a full support for floating point instructions including *IEEE* flags [IEE85]. However, we did not implement and verify any floating point functional units (FPUs) in Isabelle/HOL, because in the Verisoft project there is no software exploiting these units. If they are needed, one could instantiate FPUs (e.g. units developed and verified by Jacobi [Jac02]) in our VAMP model.

[10] The MU used in the presented VAMP model has been developed and verified by Dalinger [Dal06]. Currently, Dalinger and Alekhin [Ale08] are verifying an optimized version of the MU in Isabelle/HOL, and they expect to complete the verification in the near future.

can be saved thanks to IHaVeIt. Table 3.1 illustrates that IHaVeIt allowed us to save about 30%–40% of the work.

The greater amount of theorems in Isabelle/HOL is also due to the automation of the proofs. This is because for every "boring" subgoal of a theorem (i) we introduce a separate lemma, (ii) then, we prove it automatically, (iii) and finally, we used the proven lemma in the original proof. While in PVS, proofs for these subgoals are part of the theorem proofs.

We conclude this chapter by listing typical examples where the application of IHaVeIt reduced the user work (with respect to the previous work in PVS):

- verification of all basic hardware constructs, e.g. shifters, decoders, arithmetic logic unit,
- verification of different automata, e.g. Lemma 3.3, Lemma 3.4, control and data automaton of caches [Mül07] and MMU (Section 2.4.3),
- verification of huge combinational formulae which were parts of the induction step of the VAMP correctness theorem.

Chapter 4

Generic Device Theory

Computer systems have many built-in devices, such as hard disk drives, and network adapters. These devices are the only way a computer system, its processor(s) and programs, can communicate with the external environment. For example, a network adapter allows an operating system the communication with the network. Obviously the devices are placed between the processor(s) and the external environment.

In this chapter, we develop a generic device theory where *generic* refers to the independence from the number of devices and any device specific features. We need the latter property to instantiate this theory with different devices and to build different computer systems using the same device theory.

The device theory consists of two parts. First, it provides a device model on two levels of abstraction: devices as seen in hardware and devices as seen by an assembly programmer. Second, a correctness criterion guaranteeing a simulation relation between these two models is specified.

We call the device model, as seen in hardware, *device implementation* (*DI*). This model allows the parallel running of many devices. The device model at the assembly level is called *device specification* (*DS*). *DS* is defined in an interleaved way, and, hence, at most one device can progress with every step.

The device independence is achieved by defining *DI* and *DS* on the base of several assumptions. Thus, all concrete devices which satisfy these assumptions can be instantiated in the developed theory.

This chapter is organised as follows. In the following section, we describe the device implementation. Section 4.2 presents the device specification. In Section 4.3 we present a set of requirements which specify a set of admissible concrete devices. In this section, we also show how any run of the implementation model can be simulated by a run of the specification model. Finally, we state a simulation theorem and present a correctness proof for this theorem.

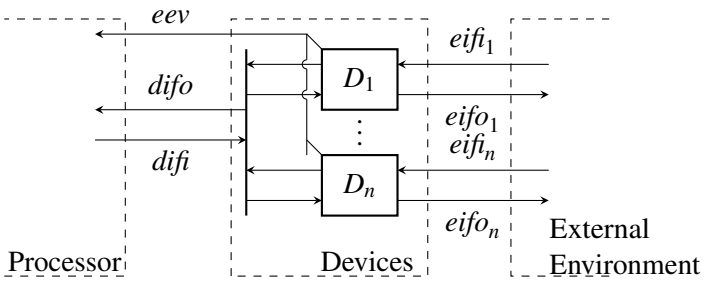


Figure 4.1: Device model: Implementation.

4.1 Device Implementation

We consider the device implementation (*DI*) as an automaton communicating with a processor on one side and with the external environment on another side. Figure 4.1 shows an overview of the *DI* model.

The *DI* model can contain many devices. Every device has a unique identifier *idx* from the set of device identifiers *DevN*. A device with identifier *idx* is characterized via:

- H_{D}^{idx} – configuration type,
- $Eifi_{idx}$ – type of the input data from the external environment of the device,
- $Eifo_{idx}$ – type of the output data to the external environment of the device.

The device configuration includes all devices in the model. It is modelled as a mapping $h_{\mathrm{D}} \in DevN \to \bigcup_{idx \in DevN} H_{\mathrm{D}}^{idx}$ which maps device identifier $idx \in DevN$ to the device configuration $h_{\mathrm{D}}(idx) \in H_{\mathrm{D}}^{idx}$. We use the notation H_{D} to denote the configuration type of the *DI* model, i.e. $H_{\mathrm{D}} \triangleq DevN \to \bigcup_{idx \in DevN} H_{\mathrm{D}}^{idx}$.

The devices have three communication channels: two with a processor and one with the external environment. The channels to the processor serve two functions:

- bidirectional data exchange between processor and devices
- unidirectional signaling of device events (interrupts) to the processor

The third channel allows communication with the external environment.

The bidirectional communication channel with the processor can pass data in both directions and is modelled as:

- $difi \in Difi$ (device interface input) – data from the processor to the device (input for devices). These data consists of request and write flags, accessed address, and data for the write access.

- *difo* ∈ *Difo* (device interface output) – data to the processor from the device (output from devices). These data are the bus protocol control flags and the data for the read access.

These interfaces have been fully described in Section 3.2.2. Figures 3.3 and 3.4 show typical processor-device communications.

The communication channel with the external environment is modelled via input and output interfaces:

- $Eifis \triangleq DevN \to \bigcup_{idx \in DevN} Eifi_{idx}$ – type of the input data from the external environment *to all* devices. *eifis* ∈ *Eifis* maps a device identifier *idx* ∈ *DevN* to the device input $eifis(idx) \in Eifi_{idx}$.

- $Eifos \triangleq DevN \to \bigcup_{idx \in DevN} Eifo_{idx}$ – type of the output data *from all* devices to the external environment. *eifos* ∈ *Eifos* maps a device identifier *idx* ∈ *DevN* to the device output $eifos(idx) \in Eifo_{idx}$.

The step function δ_{D} updates the device configurations and manages the presented communication channels. This function takes an input from the external environment *eifis* ∈ *Eifis*, an input from the processor *difi* ∈ *Difi*, a device configuration $h_{\mathrm{D}} \in H_{\mathrm{D}}$, and a reset signal *reset* ∈ $\mathbb{B}$. It produces a new device configuration $h'_{\mathrm{D}} \in H_{\mathrm{D}}$, an output *difo* ∈ *Difo* to the processor, an output *eifos* ∈ *Eifos* to the environment, and a vector of external events for the processor *eev* ∈ $\mathbb{B}^{19}$.

$$\delta_{\mathrm{D}} : Eifis \times Difi \times H_{\mathrm{D}} \times \mathbb{B} \to H_{\mathrm{D}} \times Difo \times Eifos \times \mathbb{B}^{19}$$

We don't put any restrictions on the *DI* model, except for the handshakes with the processor (Section 3.2.2).

Device Implementation Run

A run of the *DI* model is defined recursively by the application of the step function for a given number of hardware cycles. During a run the devices communicate with the processor and external environment via input sequences. The data flow from the processor to the devices is a mapping from the hardware cycles to the data, i.e. $DI^t.difi$ represents the input from the processor at cycle *t*. In a similar style, we model the data flow from the external environment: $DI^t.eifis$ is the input from the external environment at cycle *t*. Recall that in this thesis we assume that the reset signal is never set $\forall t \geq 0.\ \neg DI^t.reset$.

A run for *t* cycles starting from an initial state DI^{init} results in a new configuration $DI^t.h_{\mathrm{D}}$, outputs to the processor $DI^t.difo$ and $DI^t.eev$, and outputs to the external environment $DI^t.eifos$.

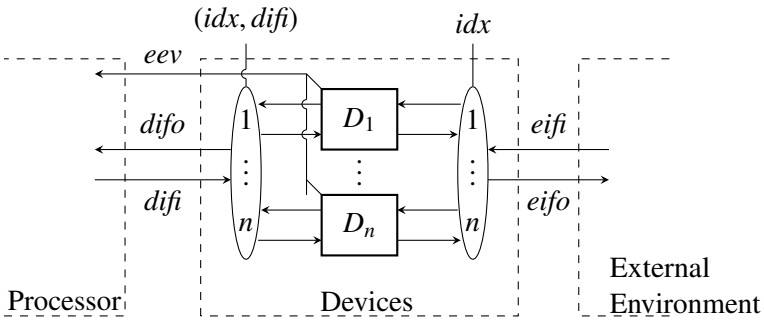


Figure 4.2: Device model: Specification.

Definition 4.1

$$DI^t \triangleq \begin{cases} DI^{init} & : t = 0 \\ \delta_{\mathrm{D}}(DI^{t-1}.eifis, DI^{t-1}.difi, DI^{t-1}.h_{\mathrm{D}}, DI^{t-1}.reset) & : otherwise \end{cases}$$

At the beginning (cycle zero) the system is in an initial state. At all other cycles we recursively apply the step function δ_{D}.

4.2 Device Specification

The implementation model can run all devices in parallel, i.e. all devices can progress at every hardware cycle. In contrast, the device specification (*DS*) is a purely sequential model, i.e. at every specification step one device at most can progress. This sequentialisation imposes some modelling differences with respect to the implementation model. For example, for the external environment the implementation model receives inputs for all devices, but the specification needs only one, for the device that progresses. The same holds for the outputs to the environment. The differences discussed above are considered in details in the next subsections.

The Specification Model

A specification device with index $idx \in DevN$ is characterized via its configuration type C_{D}^{idx} and types $Eifi_{idx}$ and $Eifo_{idx}$ which are introduced in the previous section. Configuration of the *DS* model includes all devices in the model. It is modelled as a mapping $c_{\mathrm{D}} \in DevN \rightarrow \bigcup_{idx \in DevN} C_{\mathrm{D}}^{idx}$ which maps device identifier $idx \in DevN$ to the device configuration $c_{\mathrm{D}}(idx) \in C_{\mathrm{D}}^{idx}$. We use notation C_{D} to denote the configuration type of the *DS* model, i.e. $C_{\mathrm{D}} \triangleq DevN \rightarrow \bigcup_{idx \in DevN} C_{\mathrm{D}}^{idx}$. A device in the *DS* model can progress either due to a processor access or due to an input from

the external environment. In order to distinguish between a processor access and a routine device step we extend the set of device identifiers by the processor identifier *P*. We denote this extended set by *PD*.

Definition 4.2

$$PD \triangleq \{P\} \cup DevN$$

The main feature of the *DS* model is that the communication with the processor and the environment is executed with zero delay. This requires slight changes of the channel modelling. We model the unidirectional channel for passing external event signals in the same fashion as for the implementation. The processor requests to the devices are modelled with the help of the interface *difi* ∈ *Difi* introduced above. The device responses consist solely of the data provided to the processor $difo_s \in \mathbb{B}^{32}$. The reader can notice that we drop all auxiliary protocol flags since they are no longer necessary here (see Section 3.1).

While in the implementation all devices can communicate with the external environment simultaneously, in the specification only one device can communicate with its environment. Thus, this communication channel contains input/output data depending on the progressing device.

The input from the external environment *Eifi* consists of the data for the devices: $Eifi \triangleq \bigcup_{idx \in DevN} Eifi_{idx}$. Similarly we define the output to the external environment: $Eifo \triangleq \bigcup_{idx \in DevN} Eifo_{idx}$.

Now, we introduce the *DS* model transition function Δ_D, which specifies the interaction of the devices with the processor and the external environment. It takes a processor-device identifier *idx* ∈ *PD*, an input from the external environment *eifi* ∈ *Eifi*, an input from the processor *difi* ∈ *Difi*, as well as a device configuration $c_D \in C_D$. It returns a new device configuration $c_D' \in C_D$, an output to the processor $difo_s \in \mathbb{B}^{32}$, an external output *eifo* ∈ *Eifo*, and a vector of external events $eev \in \mathbb{B}^{19}$. Thus, the transition function Δ_D has the following signature:

$$\Delta_D : PD \times Eifi \times Difi \times C_D \to C_D \times \mathbb{B}^{32} \times Eifo \times \mathbb{B}^{19}$$

We assume the following semantics of $\Delta_D(idx, eifi, difi, c_D)$:

- If $idx \neq P$, a step of a device with the identifier *idx* is triggered by the external input *eifi*. In this case Δ_D ignores the given *difi*.
- If $idx = P \wedge difi.req$, a device step is triggered by a processor request (load/store instruction). In this case Δ_D ignores the given *eifi*. The device being accessed as well as the access-type are specified by the given *difi*.
- Otherwise, no device is selected and the processor does not access any device. In this case, Δ_D does nothing.

These conditions guarantee that *one* device at most can progress. These and other assumptions *axiomatically* specify the next-state function of the *DS* model. We also

introduce a set of assumptions relating the transition functions of the *DI* and *DS* models. The concrete instantiations of the transition functions must satisfy these assumptions.

Device Specification Run

Runs of the *DS* model are defined via *computational sequences*. A computational sequence σ is a finite sequence of processor-device identifiers. It defines which device can progress in every specification step.

The specification model receives data from the processor and environment via *difi* and *eifi* channels, respectively. The data flow from the processor is described by a *difi*-sequence *DS.difi*, e.g. $DS^{j}.difi$ is the processor input for the specification step j with the identifier $\sigma(j)$. Similarly, the data flow from the external environment is represented as an *eifi*-sequence, i.e. $DS^{j}.eifi$ is the input from the external environment for the specification step j with the identifier $\sigma(j)$.

A run with a given computational sequence σ for $j \leq len(\sigma)$ steps starting from some initial state DS^{init} results in a new configuration $DS^{(j,\sigma)}.c_{\mathrm{D}}$, an output to the processor $DS^{(j,\sigma)}.difo_s$, an output sequence to the external environment $DS^{(j,\sigma)}.eifo$, and a vector of external events $DS^{(j,\sigma)}.eev$. The output sequence to the external environment $DS^{(j,\sigma)}.eifo$ accumulates *eifo*–outputs for all specification steps before j.

Definition 4.3

$$
\begin{array}{l}
DS^{(j,\sigma)} \triangleq \\
\quad \textbf{let} \\
\qquad (c_{\mathrm{D}}, difo_s, eifo, eev) := \Delta_{\mathrm{D}}(\sigma(j-1), DS^{j-1}.eifi, DS^{j-1}.difi, DS^{(j-1,\sigma)}.c_{\mathrm{D}}) \\
\quad \textbf{in} \\
\qquad \begin{cases} DS^{init} & : \; j = 0 \\ (c_{\mathrm{D}}, difo_s, DS^{(j-1,\sigma)}.eifo \circ eifo, eev) & : \; otherwise \end{cases}
\end{array}
$$

We write DS^{σ} to denote $DS^{(len(\sigma),\sigma)}$. Thus, $DS^{\sigma}.c_{\mathrm{D}}$, $DS^{\sigma}.eifo$, $DS^{\sigma}.difo_s$, $DS^{\sigma}.eev$ denote the configuration, the outputs to external environment, and the outputs to the processor after processing the complete sequence respectively.

Now we define several assumptions which guarantee the sequential behavior of the step function of the specification Δ_{D}. For readability's sake, we define them in terms of one step of *DS* model.

Assumption 4.1
Let DS_v and DS_s be two instances of the device specification model. A step with a device identifier ignores the input from processor and provides an idle output to the

processor:

$$\begin{aligned}
&\sigma(j) \neq P \ \wedge \ {DS_s}^{(j,\sigma)}.c_D = {DS_v}^{(j,\sigma)}.c_D \wedge \\
&{DS_s}^{(j,\sigma)}.eifi = {DS_v}^{(j,\sigma)}.eifi \ \wedge {DS_s}^{(j,\sigma)}.eifo = {DS_v}^{(j,\sigma)}.eifo \longrightarrow \\
&\quad {DS_s}^{(j+1,\sigma)}.c_D = {DS_v}^{(j+1,\sigma)}.c_D \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.difo_s = {DS_v}^{(j+1,\sigma)}.difo_s = {difo_s}^{\epsilon} \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.eifo = {DS_v}^{(j+1,\sigma)}.eifo \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.eev = {DS_v}^{(j+1,\sigma)}.eev
\end{aligned}$$

where ${difo_s}^{\epsilon}$ *is some idle value.*

Assumption 4.2
Let DS_v *and* DS_s *be two instances of the device specification model. A step with the processor identifier and with a processor request ignores the input from the external environment:*

$$\begin{aligned}
&\sigma(j) = P \ \wedge \ {DS_s}^{j}.difi.req \ \wedge \ {DS_s}^{(j,\sigma)}.c_D = {DS_v}^{(j,\sigma)}.c_D \wedge \\
&{DS_s}^{(j,\sigma)}.difi = {DS_v}^{(j,\sigma)}.difi \ \wedge {DS_s}^{(j,\sigma)}.eifo = {DS_v}^{(j,\sigma)}.eifo \longrightarrow \\
&\quad {DS_s}^{(j+1,\sigma)}.c_D = {DS_v}^{(j+1,\sigma)}.c_D \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.difo_s = {DS_v}^{(j+1,\sigma)}.difo_s \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.eifo = {DS_v}^{(j+1,\sigma)}.eifo \wedge \\
&\quad {DS_s}^{(j+1,\sigma)}.eev = {DS_v}^{(j+1,\sigma)}.eev
\end{aligned}$$

Assumption 4.3
A step with the processor identifier without a request does not have any effect on the devices:

$$\begin{aligned}
&\sigma(j) = P \wedge \ \neg {DS}^{j}.difi.req \longrightarrow \\
&\quad {DS}^{(j+1,\sigma)}.c_D = {DS}^{(j,\sigma)}.c_D \wedge \\
&\quad {DS}^{(j+1,\sigma)}.difo_s = {difo_s}^{\epsilon} \wedge \\
&\quad {DS}^{(j+1,\sigma)}.eifo = {DS}^{(j,\sigma)}.eifo \circ {eifo}^{\epsilon} \wedge \\
&\quad {DS}^{(j+1,\sigma)}.eev = {DS}^{(j,\sigma)}.eev
\end{aligned}$$

where ${eifo}^{\epsilon}$ *is some idle value.*

Lemma 4.4
Let DS_v *and* DS_s *be two instances of the device specification model. Let us assume they have the same initial state. Let them receive the same input sequences from the external environment. If for every processor request they receive equivalent inputs difi,*

under Assumptions 4.1, 4.2, and 4.3, their runs are equivalent.

$$
\begin{aligned}
& {DS_v}^{(0,\sigma)} = {DS_s}^{(0,\sigma)} \wedge \\
& \forall\, j < len(\sigma).\ {DS_v}^{j}.eifi = {DS_s}^{j}.eifi \wedge \\
& \forall\, j < len(\sigma).\ \sigma(j) = P \longrightarrow \\
& \quad ({DS_v}^{j}.difi.req = {DS_s}^{j}.difi.req) \wedge \\
& \quad ({DS_v}^{j}.difi.req \longrightarrow {DS_v}^{j}.difi = {DS_s}^{j}.difi) \\
& \Longrightarrow \\
& \forall\, j \leq len(\sigma).\ {DS_v}^{(j,\sigma)}.c_D = {DS_s}^{(j,\sigma)}.c_D \wedge \\
& \qquad\qquad {DS_v}^{(j,\sigma)}.eev = {DS_s}^{(j,\sigma)}.eev \wedge \\
& \qquad\qquad {DS_v}^{(j,\sigma)}.difo_s = {DS_s}^{(j,\sigma)}.difo_s \wedge \\
& \qquad\qquad {DS_v}^{(j,\sigma)}.eifos = {DS_s}^{(j,\sigma)}.eifos
\end{aligned}
$$

Proof. *We prove this lemma by induction on the sequence length. The base case is trivial. In the induction step, $\sigma \circ [idx]$, we basically have to show the following:*

$$
\begin{aligned}
& \Delta_D(idx, {DS_v}^{\sigma}.c_D, {DS_v}^{len(\sigma)}.difi, {DS_v}^{len(\sigma)}.eifi) = \\
& \Delta_D(idx, {DS_s}^{\sigma}.c_D, {DS_s}^{len(\sigma)}.difi, {DS_s}^{len(\sigma)}.eifi)
\end{aligned}
$$

The equality of the inputs from the environment (i.e. ${DS_v}^{len(\sigma)}.eifi = {DS_s}^{len(\sigma)}.eifi$) is guaranteed by the lemma premises. The equality of the states is guaranteed by the induction hypothesis, i.e. ${DS_v}^{\sigma}.c_D = {DS_s}^{\sigma}.c_D$. Thus, we have only to consider the inputs from the processor (i.e. the difi–inputs). Let us make the case distinction on idx.

Case 1: *$idx \neq P$, we know that in this case the next-state function Δ_D ignores the given input from the processor (Assumption 4.1).*

Case 2: *$idx = P$ and there is no request, i.e. $\neg DS_s.difi.req$. The lemma premises guarantee that the request bit is equivalent for both models. Assumption 4.3 guarantees that a given processor input is ignored, and hence, the subgoal holds.*

Case 3: *$idx = P$ and there is a request, i.e. $DS_s.difi.req$. The premises of the lemma guarantee the equality of the inputs from processor, i.e. ${DS_v}^{len(\sigma)}.difi = {DS_s}^{len(\sigma)}.difi$. The induction hypothesis guarantees the equality of the states. Finally, Assumption 4.2 guarantees the equality of the outputs.* □

4.3 Correctness Criterion

Recall that our implementation model is a parallel one and the specification model is a sequential one. It implies that one step in the implementation may correspond to

several consecutive specification steps or no steps at all. For example an implementation step $t \mapsto t+1$ may correspond to the specification sequence $j \mapsto \ldots \mapsto j+k$, where $k \in \{0 \ldots \mid DevN \mid\}$ is the number of progressing devices:

- $k = 0$ – no device makes a step and hence $h_{\mathrm{D}}^{t} = h_{\mathrm{D}}^{t+1}$
- $k =\mid DevN \mid$ – all devices make a step, and some of them may be accessed by the processor

We employ the scheduling function concept which we have used in Chapter 3 to relate the time notion of these models.

4.3.1 Scheduling Function

In Section 3.3.2, we introduced a scheduling function which abstracts hardware cycles and stages to instruction numbers. The definition of a scheduling function is based on some special hardware events. We have used write back event signalling that an instruction is processed, and hence, this scheduling function basically counts the number of processed instructions. Thus, we can relate the hardware state at a given cycle t with the specification state after the execution of the processed instructions.

The scheduling function presented in this section is used in the correctness criterion for the device model and for the combined processor-devices model, which is presented in the next chapter. Therefore, we split the trigger events into two groups: (i) processor-sided and (ii) external-environment-sided events.

Processor-sided events

Consider an instruction which neither writes to the memory nor accesses a device. The result of such an instruction is finally computed at the write back stage, i.e. at the cycle when the instruction is leaving the processor. Thus, the execution effect of such an instruction can be rolled back by the processor at any hardware cycle before it is written back.[1] We use the notation wb^t to denote the value of the hardware signal write-back at cycle t.

An instruction writing to the memory or accessing a device forces *irreversible* changes of the memory/device at the access cycle. The access cycle precedes the write back cycle of the instruction. Thus, the memory/device is altered before the instruction leaves the processor. Therefore, the memory and devices correctnesses have to be treated specially. Beyer [Bey05] introduced a correctness criterion for a user visible memory. His criterion captures the implementation memory state just after the memory instruction has left the memory unit of the processor (i.e. after the memory access is finished). He compares this memory state against the specification memory after the processor specification has *completely* finished the execution of that memory instruction. However, this approach is not sufficient for a model with a

[1] This rollback is possible, if for every instruction the processor can restore the program counters which have been used to fetch it. The VAMP supports such a rollback.

processor and devices, because the devices can act on their own and the memory can not.

In order to built a precise refinement mapping, we consider four events for instructions with device accesses. We list these events in the chronological order:

1. the start of a device access as visible on the processor side,

2. the end of the device access as visible on the device side,

3. the end of the device access as visible on the processor side, and

4. the write back of the instruction.

The first event is the first cycle when the processor places request on the bus, i.e. the first cycle when *difi.req* is activated (see Figure 3.3). The second event is the cycle when the accessed device puts the result on the bus. We introduce a predicate $da : H_{\mathrm{D}} \times Difi \times Eifis \to \mathbb{B}$ which for a given device state $h_{\mathrm{D}} \in H_{\mathrm{D}}$, input from processor $difi \in Difi$, and external environment $eifis \in Eifis$ signal this event. For clarity's sake we denote this event at cycle t by da^t.

From the processor and accessed device points of view *da* has almost the same semantics as *wb*, because once a device access is done its effect can not be rolled back. The third event takes places when the processor samples the device access result. For our processor model it is the signal *da_end*, which is presented in Definition 3.15. Note that between *da* and *da_end* events there is one cycle delay due to the communication bus. Finally, the write back event we have already discussed above.

For the purpose of this chapter a formal definition of *da* is not important. However, we need two properties of *da* for the correctness proof of a combined processor-device model, which is presented in the next chapter. We specify these properties axiomatically. Note that these axioms have to be proved for every concrete device model (for example see Chapter 6).

Assumption 4.5
If da^t holds there must be a preceding cycle t' when the request is started.

$$\forall t.\, da^t \longrightarrow \exists t' < t.\quad DI^{t'}.difi.req \wedge \forall t'' \in]t' : t[.\, \neg DI^{t''}.difi.req \wedge \neg da^{t''}$$

We also require the device model to have liveness property, i.e. all requests are finally processed.

Assumption 4.6

$$\forall t.\, DI^t.difi.req \longrightarrow \exists t' > t.\, da^{t'}$$

External-environment-sided events

Device steps can be triggered by the external environment. We introduce a function *DevIds* which, for a given device state $h_{\mathrm{D}} \in H_{\mathrm{D}}$ and the external inputs $eifis \in Eifis$, computes a computational sequence σ consisting of only device identifiers.

$$\sigma = DevIds(h_{\mathrm{D}}, eifis)$$

This sequence σ corresponds to the devices making progress due to that input in one step of the implementation.

The definition of this function depends on the concrete device model and how the gate-level device model has to be abstracted. Here we present possible examples of the *DevIds* semantics:

- In a very simple case we can model every implementation step in the specification. In this case *DevIds* always returns a computational sequence which contains all device identifiers without duplications.

- With the help of this function we can purge the steps of a gate-level device which have no effect. If the state of the device has not been changed due to the external environment, its identifier is not in the result of *DevIds*.

- We can also ignore some steps of the gate level model, i.e. we will not model these steps in the specification. Let us consider a device which has a n-bit counter. Every time the counter reaches $2^n - 1$, the device generates an interrupt and places it on *eev*. In the specification we could abstract it as a device which from time to time generates an interrupt.

We put one restriction on *DevIds*: it should not return a sequence with duplicated device identifiers.

Assumption 4.7

$$\begin{aligned} &\forall h_D,\ eifis,\ \sigma.\ \sigma = DevIds(h_D, eifis) \longrightarrow \\ &\quad \forall i < len(\sigma), j < len(\sigma).\ i \neq j \longrightarrow \sigma(i) \neq \sigma(j) \end{aligned}$$

Now we can define the scheduling function sI_{PD}. It builds a computational sequence σ for a given number of hardware cycles. Figure 4.3 depicts a part of the gate-level run of a system with processor and three devices. It also presents how this subrun can be abstracted into a computational sequence. We define function sI_{PD} by recursion on hardware cycles:

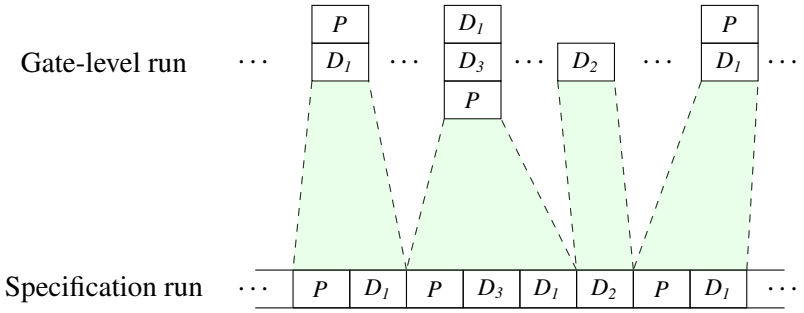


Figure 4.3: Abstraction of hardware cycles to a computational sequence via sI_{PD}.

Definition 4.4

$$
\begin{array}{l}
sI_{\mathrm{PD}}(0) \triangleq [] \\
sI_{\mathrm{PD}}(t+1) \triangleq \\
\quad \textbf{let} \\
\qquad \sigma^t \quad := \quad sI_{\mathrm{PD}}(t) \\
\qquad was\_da \quad := \quad \exists t' < t.\, da^{t'} \wedge \forall t'' \in]t' : t[.\, \neg wb^{t''} \\
\qquad devids \quad := \quad DevIds(DI^t.h_{\mathrm{D}}, DI^t.eifis) \\
\quad \textbf{in } \sigma^t \circ \begin{cases} devids \circ [P] & : \quad da^t \\ [P] \circ devids & : \quad wb^t \wedge \neg was\_da \wedge \neg da^t \\ devids & : \quad otherwise \end{cases}
\end{array}
$$

At the beginning, sI_{PD} returns an empty sequence. With every step the previous sequence is extended by the identifiers of the changed system parts. Here we distinguish three cases:

1. da^t – The device access is finished. The sequence is extended by the identifiers of the progressed devices followed by the processor identifier. The processor identifier is added because:

   - *da* signals that the access is finished on the device side
   - the processor can not roll back the access effect

   Therefore, the specification must make a step any way. The processor identifier is added at the tail of the sequence, because at the end of cycle t the valid data are put on the *difo*-bus. The *eev*-bus is also updated at the end of cycle t by all devices. The implementation processor reads these data in the next cycle. Therefore, the processor in the specification makes its step after the devices.

2. $wb^t \wedge \neg was\_da \wedge \neg da^t$ – There is the write back of an instruction without a device access. In this case the processor and the devices progress independently. We add the processor identifier and all identifiers of the changed devices to the sequence. The processor identifier is added at the head of the subsequence since:

   - wb^t signals the instruction completion on the processor side
   - the processor reads the *eev* bus, and these data are computed in the previous cycle; we have one cycle delay due to the bus communication.

   Therefore, the processor in the specification also reads the data which are provided by the previous device configuration.

3. Default case – In this case only the devices, specified by *DevIds*, are progressing. We don't add the processor identifier to the sequence because either

   - there is no processor-device communication, or
   - if wb^t is set, an instruction with a device access leaves the processor; the processor identifier, which corresponds to this instruction, was added earlier (see first case), or
   - if wb^t is not set, there is no write back of any instruction.

We use the notation σ^t to denote the computational sequence up to cycle t, i.e.

$$\sigma^t \triangleq sI_{\mathrm{PD}}(t)$$

We will denote the computational subsequence between cycles ts and te by σ^{te}_{ts}, i.e. $\sigma^{te} = \sigma^{ts} \circ \sigma^{te}_{ts}$

It is easy to prove that sI_{PD} is a monotonic function with respect to the length of the output sequence.

Lemma 4.8

$$\begin{array}{l} t < T \\ \Longrightarrow \\ len(\sigma^t) \leq len(\sigma^T) \ \wedge \sigma^t \prec \sigma^T \end{array}$$

The next function for a given index of a sequence element computes the hardware cycle at which the element has been added into the sequence.

Definition 4.5

$$AddedAt(j) \triangleq max\{t \mid j \geq len(\sigma^t)\}$$

The algorithm for the construction of the computational sequence guarantees that for every element in the sequence there is a unique hardware cycle when it has been added.

Lemma 4.9

$$\forall\, j < len(\sigma^T).\ \exists t < T.\ t = AddedAt(j)\ \wedge\ \forall t'.\ t' \neq t \longrightarrow t' \neq AddedAt(j)$$

We can also prove that if two indices point to equal elements in the computational sequence and these elements have been added at the same cycle, these indices are equal.

Lemma 4.10

$$\begin{array}{l}\forall i < len(\sigma^T).\ \forall j < len(\sigma^T).\\ \quad \sigma^T(i) = \sigma^T(j)\ \wedge AddedAt(i) = AddedAt(j)\ \longrightarrow i = j\end{array}$$

4.3.2 Input and Output Relations

We can only state and prove the relations between the *DI* and the *DS* models if they receive the same inputs. Therefore, we need a way to relate or to synchronise input sequences for these two models.

We define a predicate *sync_eifis* which tests whether inputs from the external environments are synchronised. The predicate *sync_eifis* works as follows. If a device with the identifier $\sigma^{te}(j)$ makes a step at $t \mapsto t+1$, this device makes this step with the input $eifis^t(idx)$. We compute this hardware cycle t as $t = AddedAt(j)$. Then, we compare implementation input for the device $eifis^t(idx)$ with the specification input for step j, i.e. with seq_eifi^j. Thus, predicate *sync_eifis* guarantees that devices in the implementation and the specification receive the same inputs.

Definition 4.6
Let *seq_eifis* be a sequence of the implementation inputs from the external environment. Let *seq_eifi* be a sequence of the specification inputs from the external environment.

$$\begin{array}{l}sync\_eifis(ts, te, seq\_eifis, seq\_eifi) \triangleq\\ \forall\, j < len(\sigma^{te}).\\ \quad \texttt{if}\ \sigma^{te}(j) \neq P \wedge (\exists t \in [ts : te[.\ t = AddedAt(j))\\ \quad \texttt{then}\ seq\_eifis^{AddedAt(j)}(\sigma^{te}(j)) = seq\_eifi^j\\ \quad \texttt{else}\ seq\_eifi^j = eifi^{\epsilon}\end{array}$$

We apply a similar strategy to synchronise the inputs from the processor. The function *sync_difi* implements the synchronization.

Definition 4.7
Let R be a predicate which tests whether a given step number j was added in the computational sequence at the transition $t \mapsto t+1$ due to a device access: $R(j,t) = da^t \ \wedge t = AddedAt(j)$. Let seq_difi_I be a sequence of the implementation inputs from the processor. Let seq_difi_S be a sequence of the specification inputs from the processor.

$$
\begin{array}{l}
sync\_difi(ts, te, seq\_difi_I, seq\_difi_S) \triangleq \\
\forall j < len(\sigma^{te}). \\
\quad \texttt{if } \sigma^{te}(j) = P \wedge (\exists t \in [ts : te[.\ R(j,t)) \\
\quad \texttt{then } (seq\_difi_I{}^{(The\ x.\ R(j,x))}.\{w,a,din\} = seq\_difi_S{}^{j}.\{w,a,din\}) \ \wedge seq\_difi_S{}^{j}.req \\
\quad \texttt{else } \neg seq\_difi_S{}^{j}.req
\end{array}
$$

This definition is based on the end of a processor-device interaction. If there was the end of a device access, we compare the implementation data to the specification ones at the end of the request. Since at the end of a request the processor deactivates the request flag, according to the bus protocol (see Figure 3.3), we require the active value of $seq\_difi_S{}^{i}.difi.req$. If the specification step j does not have processor-device interaction, we only require that the boolean flag *req* of the specification must be inactive and other fields do not matter.

Recall that the processor keeps the request data stable until the end of request (Lemma 3.9), and, hence, we can map them to the specification ones. From the device point of view this property has the form:

Corollary 4.11

$$
\begin{array}{l}
\forall t.\ da^t \longrightarrow \\
\quad \exists t' < t.\ DI^{t'}.difi.req \wedge \\
\quad\quad \forall t'' \in [t' : t].\ DI^{t''}.difi.\{w,a,din\} = DI^{t'}.difi.\{w,a,din\} \wedge \\
\quad\quad \forall t'' \in]t' : t].\ \neg DI^{t''}.difi.req
\end{array}
$$

Similarly to the synchronisation of inputs we define a predicate *sync_eifos*. It tests for synchronising of a sequence of implementation outputs to the external environment and a sequence of specification outputs to the external environment.

Definition 4.8
Let K be a predicate which tests whether (i) a given step number j was added in the computational sequence at the transition $t \mapsto t+1$, and (ii) this is a step of a device with

a given identifier $idx \in DevN$. Formally, $K(j,t,idx) \triangleq (t = AddedAt(j)) \wedge \sigma^t(j) = idx$. Let *seq_eifos* be a sequence of the implementation inputs from the external environment. Let *seq_eifo* be a sequence of the specification inputs from the external environment.

$$\begin{aligned}
&sync_eifos(ts, te, seq_eifos, seq_eifo) \triangleq \\
&\forall t \in]ts : te].\ \forall idx \in DevN.\ (\exists j < len(\sigma^{te}).\ K(j,t,idx)) \longrightarrow \\
&\quad seq_eifos^{t+1}(idx) = seq_eifo^{(The\ x.\ K(x,t,idx))+1}
\end{aligned}$$

This definition states that whenever an implementation device makes some transition $t \mapsto t+1$ and there is a corresponding transition in the specification, then the outputs *eifo* of these transitions must match. We relate seq_eifos^{t+1} and seq_eifo^{j+1} because we need the outputs of the transition $t \mapsto t+1$ and after step j, respectively.

4.3.3 Admissible Step Functions

In the previous sections we introduced two device models *DI* and *DS*. They are based on the generic step functions δ_{D} (for the implementation) and Δ_{D} (for the specification). In order to state any simulation relation between runs of *DI* and *DS* models, we have to specify how the results of these generic step functions are related with each other.

First, for every device with identifier *idx* we introduce a predicate $sim_{idx} : H_{\mathrm{D}}^{idx} \times C_{\mathrm{D}}^{idx} \to \mathbb{B}$ which tests whether the implementation state of the device is in a relation with its specification state. We introduce a predicate sim_D which holds if the states of all devices are in relations with their specification states.

Definition 4.9

$$sim_D(h_{\mathrm{D}}, c_{\mathrm{D}}) \triangleq \forall idx \in DevN.\ sim_{idx}(h_{\mathrm{D}}(idx), c_{\mathrm{D}}(idx))$$

We introduce a "one-step" assumption. This assumption specifies a relationship between one step in the implementation and a sequence of steps in the specification. In other words, one step of the *DI* model (single application of δ_{D}) must correspond to a sequential execution of specification steps computed by sI_{PD}. We define this assumption for *any* initial state and *any* input sequences.

Assumption 4.12

$$
\begin{aligned}
&sim_D(DI^0.h_D, DS^{\sigma^0}.c_D) \wedge \\
&sync\_eifis(0,1,DI.eifis,DS.eifi) \wedge\ sync\_difi(0,1,DI.difi,DS.difi) \longrightarrow \\
&\quad sim_D(DI^1.h_D, DS^{\sigma^1}.c_D) \wedge \\
&\quad sync\_eifos(0,1,DI.eifos,DS^{\sigma^1}.eifo) \wedge \\
&\quad DI^1.difo.dout = DS^{\sigma^1}.difo_s \wedge \\
&\quad DI^1.eev = DS^{\sigma^1}.eev
\end{aligned}
$$

Note that we can use the $DS^{\sigma^1}.difo_s$ output as the correct value for $DI^1.difo.dout$ because:

- If da^0, the implementation produces an output to the processor. The corresponding specification step is the last in the computational sequence σ^1 (Definition 4.4).
- If $\neg da^0$ and σ^1 contains no processor identifier, then Assumption 4.1 guarantees that the DS model produces the idle output. Hence, the *difo* output of δ_{D} must also be the idle output.
- If $\neg da^0$ and σ^1 contains the processor identifier, then the processor identifier is the first element in σ^1 (Definition 4.4). In this case $sync\_difi(0,1,DI.difi,DS.difi)$ guarantees that $\neg DS.difi.req$. Assumption 4.3 guarantees that the DS model produces the idle output. Hence, the *difo* output of δ_{D} must also be the idle output.

Now, we derive from Assumption 4.12 a generic lemma for a step $t \rightarrow t+1$:

Lemma 4.13

$$
\begin{aligned}
&sim_D(DI^t.h_D, DS^{\sigma^t}.c_D) \wedge \\
&sync\_eifis(t,t+1,DI.eifis,DS.eifi) \wedge\ sync\_difi(t,t+1,DI.difi,DS.difi) \\
&\Longrightarrow \\
&\quad sim_D((DI^t)^1.h_D, (DS^{\sigma^t})^{\sigma_t^{t+1}}.c_D) \wedge \\
&\quad sync\_eifos(t,t+1,DI.eifos,(DS^{\sigma^t})^{\sigma_t^{t+1}}.eifo) \wedge \\
&\quad (DI^t)^1.difo.dout = (DS^{\sigma^t})^{\sigma_t^{t+1}}.difo_s \wedge \\
&\quad (DI^t)^1.eev = (DS^{\sigma^t})^{\sigma_t^{t+1}}.eev
\end{aligned}
$$

The proof of Lemma 4.13 requires several auxiliary lemmata stating the decomposition property of runs, scheduling function, and synchronization predicates. We present these lemmata in the following subsection.

Decomposition lemmata

Lemma 4.14
To run the DI model for $t1 + t2$ cycles is the same as to run it first for $t1$ cycles and then for $t2$ cycles:

$$DI^{t1+t2} = (DI^{t1})^{t2}$$

Lemma 4.15
To run the DS model with the computational sequence $\sigma 1 \circ \sigma 2$ is the same as to run it first with sequence $\sigma 1$ and then with sequence $\sigma 2$:

$$DS^{\sigma 1 \circ \sigma 2} = (DS^{\sigma 1})^{\sigma 2}$$

The scheduling function sI_{PD} has to preserve the decomposition property as well.

Lemma 4.16
The computational sequence for $t1 + t2$ cycles can be decomposed into two subsequences. One starting from 0 and going to $t1$ and the other from $t1$ up to $t2$:

$$\sigma^{t1+t2} = \sigma_0^{t1} \circ \sigma_{t1}^{t2}$$

The proofs for Lemma 4.14, Lemma 4.15, and Lemma 4.16 were carried out by induction over hardware cycles, or the length of computational sequences.

We also need the decomposition property for inputs and outputs. Hence, the functions *sync_eifis*, *sync_difi*, and *sync_eifos* have to preserve it.

Lemma 4.17

$$\begin{array}{l}\forall t.\ \forall st \\ \quad sync\_eifis(0, st + t, seq\_eifis, seq\_eifi) \longrightarrow \\ \quad sync\_eifis(st, st + t, seq\_eifis, seq\_eifi)\end{array}$$

Lemma 4.18

$$\begin{array}{l}\forall t.\ \forall st. \\ \quad sync\_difi(0, st + t, seq\_difi_I, seq\_difi_S) \longrightarrow \\ \quad sync\_difi(st, st + t, seq\_difi_I, seq\_difi_S)\end{array}$$

Lemma 4.19

$$\begin{array}{l}\forall t.\ \forall te \\ sync\_eifos(0, t, seq\_eifos, seq\_eifo) \wedge \\ sync\_eifos(t + 1, t + te, seq\_eifos, (\lambda x.\ seq\_eifo(x + len(\sigma^t))) \longrightarrow \\ \quad sync\_eifos(0, t + te, seq\_eifos, seq\_eifo)\end{array}$$

These lemmata are proven by using the definitions of *sync_eifis*, *sync_difi*, and *sync_eifos* as well as Lemma 4.9 and Lemma 4.10.

4.3.4 The Simulation Theorem

As the correctness criterion we have selected a simulation relation. We want to show that every run of the implementation model *DI* can be simulated by a run of the specification model *DS* with respect to sI_{PD}.

Theorem 4.20
Let σ^T be the computation sequence up to cycle T and let all inputs be synchronised. Let the initial configurations be in the simulation relation.

We show that the state of the DI model after T cycles and the state of the DS model after executing sequence σ^T are equivalent. Moreover, these two models produce equal outputs.

$$
\begin{aligned}
& sim_D(DI^0.h_D, DS^0.c_D) \wedge \\
& sync\_eifis(0, T, DI.eifis, DS.eifi) \wedge \\
& sync\_difi(0, T, DI.difi, DS.difi) \\
& \Longrightarrow \\
& sim_D(DI^T.h_D, DS^{\sigma^T}.c_D) \wedge \\
& sync\_eifos(0, T, DI^T.eifos, DS^{\sigma^T}.eifo) \wedge \\
& DI^T.difo.dout = DS^{\sigma^T}.difo_s \wedge \\
& DI^T.eev = DS^{\sigma^T}.eev
\end{aligned}
$$

Theorem 4.20 states the correctness criterion after a given number of hardware cycles T. This theorem also covers the correctness for all input sequences and initial states since we have not restricted them. The assumptions of Theorem 4.20 filter out the specification runs which can not exist in the implementation model.

Proof. *The proof is carried out by induction on hardware cycles. The induction base is trivial: we know that the initial configurations are the same and the output buses are initialised with the same idle values.*

In the induction step we have to show the following:

$$
\begin{aligned}
IH: \quad & sync\_eifis(0, t+1, DI.eifis, DS.eifi) \wedge \\
& sync\_difi(0, t+1, DI.difi, DS.difi) \wedge \\
& sim_D(DI^t.h_D, DS^{\sigma^t}.c_D) \wedge \\
& sync\_eifos(0, t, DI.eifos, DS^{\sigma^t}.eifo) \\
\Longrightarrow & \\
IS: \quad & sim_D(DI^{t+1}.h_D, DS^{\sigma^{t+1}}.c_D) \wedge \\
& sync\_eifos(0, t+1, DI.eifos, DS^{\sigma^{t+1}}.eifo) \wedge \\
& DI^{t+1}.difo.dout = DS^{\sigma^{t+1}}.difo_s \wedge \\
& DI^{t+1}.eev = DS^{\sigma^{t+1}}.eev
\end{aligned}
$$

We start the proof by decomposing the application of the scheduling function $sI_{PD}(t+1)$ *into two regions: (i) from* 0 *to* t *and (ii) from* t *to* $t+1$ *. Lemma 4.16 provides us with the necessary decomposition rule:* $\sigma^{t+1} = \sigma^t \circ \sigma_t^{t+1}$.

By applying Lemma 4.15 we split the specification run into two subruns: (i) one run with σ^t *and (ii) the second run with* σ_t^{t+1}. *Together with Lemma 4.14 and Lemma 4.19 we rewrite the induction step as follows:*

$$
\begin{aligned}
& sim_D(((DI^t)^1).h_D, ((DS^{\sigma^t})^{\sigma_t^{t+1}}).c_D) \wedge \\
& sync\_eifos(t, t+1, DI.eifos, (DS^{\sigma^t})^{\sigma_t^{t+1}}.eifo) \wedge \\
& (DI^t)^1).difo.dout = ((DS^{\sigma^t})^{\sigma_t^{t+1}}).difo_s \wedge \\
& ((DI^t)^1).eev = ((DS^{\sigma^t})^{\sigma_t^{t+1}}).eev
\end{aligned}
$$

Now to finish the proof we have to apply Lemma 4.13. This lemma has three assumptions: (i) The equivalence of the states, i.e. $sim_D(DI^t.h_D, DS^{\sigma^t}.c_D)$*, this is provided by the induction hypothesis. (ii) The input sequences from the processor must be synchronised for the step* $t \mapsto t+1$. *From the induction hypothesis we know that they are synchronized from* 0 *until* $t+1$. *From the latter and Lemma 4.18 we can derive the needed synchronization for the step* $t \mapsto t+1$. *(iii) The input sequences from the external environment must be synchronised for the step* $t \mapsto t+1$. *We employ the argumentation as in the previous case together with Lemma 4.17. .* □

4.4 Summary

In this chapter we presented a generic device theory. The theory contains one parallel model and one sequential model. The parallel model is meant to be the implementation of the devices at the gate level, where at every hardware cycle each device can progress. The sequential model is used as the specification of the devices at the assembly level, where the devices progress one after another. We consider both models not as standalone systems but as parts of computer systems, which consist of processors and external environments.

We stated and proved the simulation theorem stating that every run of the implementation can be simulated by a run of the specification.

This theory can also be easily instantiated with concrete devices. This possible thanks to the generic types for configuration of devices and the interface types to the external environment.

The generality of the device theory is based on *seven* assumptions and the protocol specifying communication between processor and devices (Section 3.2.2). Thus, any concrete implementation and specification models, which agree on these assumptions, are automatically correct.

Chapter 5

VAMP^{XT} & Devices: The Computer System

In this chapter, we develop a computer system which consists of two parts: a processor and integrated memory-mapped devices. Similarly to the previous chapters, we present this system on the gate level and the assembly-language level. Our strategy is quite simple: we put together all models presented earlier. Thus, the gate-level model is a combination of the VAMP processor and the *DI* model and the specification is a combination of the *ISA* and *DS* models. We also present and prove a correctness criterion stating a simulation relation between these two combined models.

5.1 VD-System Implementation

The implementation of the combined VAMP-Devices system (*VDI*) integrates the VAMP and the generic device model on the gate level. The processor and devices communicate via the device interface buses *difi* and *difo*, which are internal in this model. These buses introduce one cycle delay in the communication (Section 3.2.2), i.e. the processor places the request on the *difi* and at the next cycle the device model reads these data. Therefore, we introduce the state of the communication bus $h_B \in H_B$ which holds the data from the processor $h_B.difi \in Difi$ and the data to the processor $h_B.difo \in Difo$ and external event vector $h_B.eev \in \mathbb{B}^{19}$:

$$H_B \triangleq Difi \times Difo \times \mathbb{B}^{19}$$

Formally, the configuration of the *VDI*, $h_{\mathrm{PD}} \in H_{\mathrm{PD}}$, has the following type:

$$H_{\mathrm{PD}} \triangleq H_{\mathrm{P}} \times H_{\mathrm{D}} \times H_B$$

where $h_{\mathrm{PD}}.h_{\mathrm{P}}$ denotes the VAMP configuration, $h_{\mathrm{PD}}.h_{\mathrm{D}}$ denotes the configuration of the devices, and $h_{\mathrm{PD}}.h_B$ denotes the state of the bus.

The next-state function of *VDI* updates the processor and the devices by the application of their next-state functions. It also defines the communication between

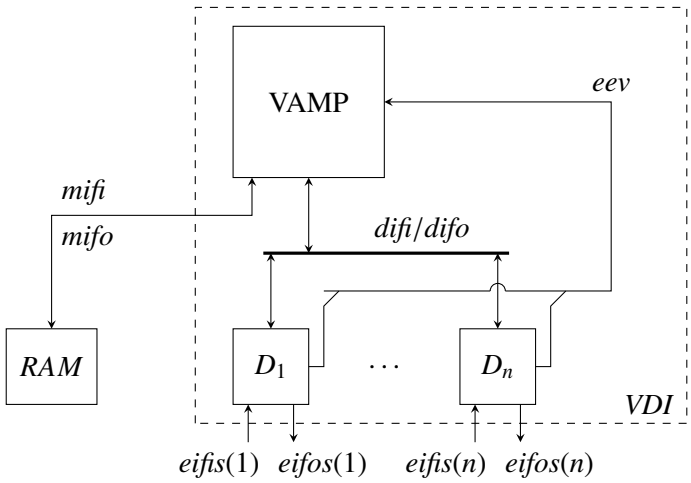


Figure 5.1: Architecture of the implementation of the combined model.

the system components and the external world. For the *VDI* model the external world consists of the memory chips and the external environment of the devices. Formally, the step function of *VDI* computes the next state $h'_{\mathrm{PD}} \in H_{\mathrm{PD}}$ based on a given system state $h_{\mathrm{PD}} \in H_{\mathrm{PD}}$, inputs from the device environment $eifis \in Eifis$ and the memory $mifo \in Mifo$. It also computes the outputs of the system to the external environment $eifos \in Eifos$ of the devices and to the memory $mifi \in Mifi$.

Definition 5.1

$$
\begin{array}{l}
\delta_{\mathrm{PD}}(h_{\mathrm{PD}}, eifis, mifo, reset) \triangleq \\
\quad \textbf{let} \\
\qquad h'_{\mathrm{P}} := \delta_{\mathrm{P}}(h_{\mathrm{PD}}.h_{\mathrm{P}}, mifo, h_{\mathrm{PD}}.h_B.difo, h_{\mathrm{PD}}.h_B.eev, reset) \\
\qquad (h'_{\mathrm{D}}, difo', eifos, eev') := \delta_{\mathrm{D}}(eifis, h_{\mathrm{PD}}.h_B.difi, h_{\mathrm{PD}}.h_{\mathrm{D}}, reset) \\
\qquad (mifi, difi') := \omega_{\mathrm{P}}(h_{\mathrm{PD}}.h_{\mathrm{P}}) \\
\qquad h_B{}' := (difi', difo', eev') \\
\qquad h'_{\mathrm{PD}} := (h'_{\mathrm{P}}, h'_{\mathrm{D}}, h_B{}') \\
\quad \textbf{in } (h'_{\mathrm{PD}}, eifos, mifi)
\end{array}
$$

A run of the *VDI* model is defined recursively by the application of the next-state function for a given number of hardware cycles. The communication with the external environment is modelled in a similar fashion as for the device model (Section 4.1): we employ $VDI^t.eifis$ to denote the device inputs from the external environment at cycle t. We denote the input from the external memory[1] at cycle t as $VDI^t.mifo$. A run

[1]This external memory is only specified but not implemented on gate-level.

for t cycles starting from a configuration h_{PD}^{init} results in a new configuration $VDI^t.h_{\mathrm{PD}}$, outputs to the external environment $VDI^t.eifos$, and outputs to the external memory $VDI^t.mifi$. Formally:

Definition 5.2

$$VDI^t.(h_{\mathrm{PD}}, eifos, mifi) \triangleq \begin{cases} (h_{\mathrm{PD}}^{init}, eifos^{\epsilon}, mifi^{\epsilon}) & : \quad t = 0 \\ \delta_{\mathrm{PD}}(VDI^{t-1}.h_{\mathrm{PD}}, VDI^{t-1}.eifis, & \\ \qquad VDI^{t-1}.mifo, VDI^{t-1}.reset) & : \quad otherwise \end{cases}$$

where $eifos^{\epsilon}$ and $mifi^{\epsilon}$ are the idle values.

For clarity's sake, we use the shorthand $VDI^t.h_{\mathrm{P}}$ to denote the configuration of the VAMP at cycle t, i.e. $VDI^t.h_{\mathrm{P}} \triangleq VDI^t.h_{\mathrm{PD}}.h_{\mathrm{P}}$. Similarly, use the shorthand $VDI^t.h_{\mathrm{D}}$, $VDI^t.eev$, $VDI^t.difi$, and $VDI^t.difo$ to access configuration of devices, external event vector, input for devices, and output for devices respectively.

Recall that in Chapter 3 we have assumed the *VAMP* input signal *reset* to be inactive for all hardware cycles $t \geq 0$. We make the same assumption for the signal *reset* of the *VDI* model.

The Memory of the *VDI* Model

Similarly to the processor *VAMP*, the *VDI* model does not have any detailed implementation component for the memory. We define the state of the *VDI* memory by observing the memory interfaces as we have done it for *VAMP* (see Definition 3.18).

Definition 5.3
Let *init_mem* be the initial memory content. We introduce a function $bw(t, dold) \in \mathbb{B}^{64}$ which for a given cycle $t \in \mathbb{N}$ and a given old content of the memory cell $dold \in \mathbb{B}^{64}$ computes the value to be written in the memory, i.e.

$$bw(t, dold) \triangleq byteupd(VDI^t.mifi.bwb, dold, VDI^t.mifi.din)$$

The memory content $M(VDI, t)$ at cycle t is recursively defined as follows:

$$M(VDI, t)(a) \triangleq \begin{cases} init\_mem(a) & : t = 0 \\ bw(t, M(VDI, t-1)(a)) & \\ \qquad : VDI^{t-1}.mifi.a = a \wedge & \\ \qquad \neg busy(VAMP^{t-1}.mifo) & \\ M(VDI, t-1)(a) & \\ \qquad : otherwise & \end{cases}$$

Decomposition of the *VDI* Model

The combined model *VDI* can be easily decomposed into *VAMP* and *DI*. We introduce an additional notation which allows us to compactly describe the decomposition lemmata.

Definition 5.4
We define relations on states, inputs, and outputs of the device model *DI* and its counterpart in *VDI*.

$$
\begin{aligned}
VD_c(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.h_\mathrm{D} = VDI^t.h_\mathrm{D} \\
VD_{difi}(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.difi = VDI^t.difi \\
VD_{eifis}(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.eifis = VDI^t.eifis \\
VD_{difo}(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.difo = VDI^t.difo \\
VD_{eifos}(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.eifos = VDI^t.eifos \\
VD_{eev}(\mathrm{T}) &\triangleq \forall t \leq T.\ DI^t.eev = VDI^t.eev
\end{aligned}
$$

Lemma 5.1
The stand-alone device model DI and the device part of VDI have the same behaviour, if they have started from equal states and have received the same inputs.

$$
\begin{aligned}
&VD_c(0) \ \wedge VD_{difi}(T-1) \ \wedge VD_{eifis}(T-1) \Longrightarrow \\
&VD_c(T) \ \wedge VD_{difo}(T) \ \wedge VD_{eifos}(T) \ \wedge VD_{eev}(T)
\end{aligned}
$$

Proof. *The proof is carried out by induction on hardware cycles. The induction base is trivial. To finish the proof, we use (i) the fact that the same step function δ_D is used in both models (Definition 4.1, Definition 5.1) and (ii) the induction hypothesis which guarantees that both step functions receive the same inputs.* □

We similarly extract the *VAMP* from the combined model.

Definition 5.5
We define relations on states, inputs, and outputs of the *VAMP* and its counterpart in *VDI*.

$$
\begin{aligned}
VV_c(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.h_\mathrm{P} = VDI^t.h_\mathrm{P} \\
VV_{difo}(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.difo = VDI^t.difo \\
VV_{eev}(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.eev = VDI^t.eev \\
VV_{difi}(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.difi = VDI^t.difi \\
VV_{mifo}(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.mifo = VDI^t.mifo \\
VV_{mifi}(\mathrm{T}) &\triangleq \forall t \leq T.\ VAMP^t.mifi = VDI^t.mifi
\end{aligned}
$$

Lemma 5.2
The VAMP and the processor of VDI have the same behaviour, if they have started from equal states and have received the same inputs.

$$VV_c(0) \wedge VV_{difo}(T-1) \wedge VV_{eev}(T-1) \wedge VV_{mifo}(T-1) \Longrightarrow$$
$$VV_c(T) \wedge VV_{difi}(T) \wedge VV_{mifi}(T) \wedge M(VDI,t) = M(VAMP,t)$$

5.2 VD-System Specification

The specification of the VAMP-Devices system (*VDS*) consists of the *ISA* model of the processor and the sequential device model *DS* (Figure 5.2). In contrast to the implementation, we do not model the bus between the processor and the device models. This is because the processor-device communication is completed with zero delay, i.e. the processor accesses a device and the device immediately produces the answer to the processor. The configuration of *VDS*, $c_{\mathrm{PD}} \in C_{\mathrm{PD}}$, has the following type:

$$C_{\mathrm{PD}} \triangleq C_{\mathrm{P}} \times C_{\mathrm{D}}$$

The components of the *VDS* system can progress in an interleaved way, e.g. a network adapter can communicate with the network without interacting with the processor. A step of the *VDS* model can either be a processor step with device interaction, a processor step without any device interaction, or a device step. We employ processor-device identifiers (see Section 4.2) to determine which component makes a step. Now we define the next-state function Δ_{PD} which takes an identifier $idx \in PD$ of the updated component, an input from the environment $eifi \in Eifi$, and a system state $c_{\mathrm{PD}} \in C_{\mathrm{PD}}$. It returns a new configuration $c'_{\mathrm{PD}} \in C_{\mathrm{PD}}$.

Definition 5.6

$$\begin{array}{l}
\Delta_{\mathrm{PD}}(idx, eifi, c_{\mathrm{PD}}) \triangleq \\
\quad \textbf{let} \\
\qquad difi \quad := \quad \Omega_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}}) \\
\qquad (c_{\mathrm{D}}', difo_s, eifo, eev) \quad := \quad \Delta_{\mathrm{D}}(idx, c_{\mathrm{PD}}.c_{\mathrm{D}}, eifi, difi) \\
\qquad c'_{\mathrm{P}} \quad := \quad \begin{cases} \Delta_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}}, eev, difo_s) & : \ idx = P \\ c_{\mathrm{PD}}.c_{\mathrm{P}} & : \ otherwise \end{cases} \\
\quad \textbf{in} \ (c'_{\mathrm{P}}, c_{\mathrm{D}}')
\end{array}$$

Recall, Δ_{P} ignores the given $difo_s$ in case there is no device access (Lemma 3.1). Remember also that Δ_{D} allows at most one device make progress (Section 4.2). Thus, the next-state function Δ_{PD} has the following semantics:

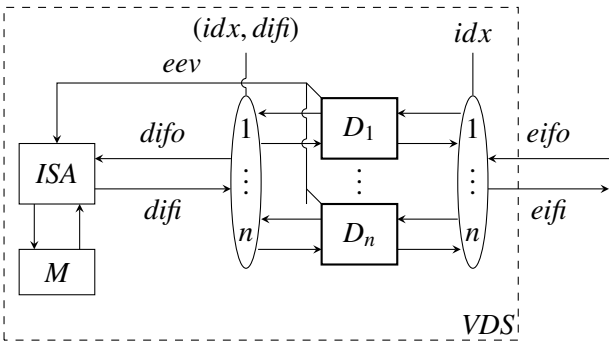


Figure 5.2: Architecture of the specification of the combined model.

- If $idx \neq P$, the device with the identifier *idx* makes a step. This device only considers the given *eifi* and its previous state. The processor state is not changed.

- If $idx = P \wedge difi.req$, the processor communicates with a device (load/store instruction). The accessed device is encoded in *difi.a*. The processor state is updated correspondingly to the executed device access, e.g. storing $difo_s$. In this case, Δ_D ignores the given *eifi* and produces some idle $eifo^\epsilon$.

- Otherwise, only the processor makes a step. It executes an instruction without device access.

The output function Ω_{PD} produces the output to the external environment $eifo \in Eifo$.

Definition 5.7

$$
\begin{array}{l}
\Omega_{PD}(idx, c_{PD}, eifi) \triangleq \\
\quad \textbf{let} \\
\qquad \begin{array}{lcl}
difi & := & \Omega_P(c_{PD}.c_P) \\
(c_D', mifo, eifo, eev) & := & \Delta_D(idx, c_{PD}.c_D, eifi, difi)
\end{array} \\
\quad \textbf{in } eifo
\end{array}
$$

Note that if the input identifier is the one of the processor (i.e., $idx = P$), Δ_D (and, hence, Ω_{PD}) produces an idle value $eifo^\epsilon$.

We define runs of the *VDS* model over computational sequences (Section 4.2). Running *VDS* from an initial state c_{PD}^{init} with computational sequence σ for $j \leq len(\sigma)$ steps results in a new state $VDS^{(j,\sigma)}.c_{PD}$. During a run the *VDS* model communicates with external environment. We denote input data from the external environment for

the specification step j as $VDS^{j-1}.eifi$.

$$VDS^{(j,\sigma)}.c_{\mathrm{PD}} \triangleq \begin{cases} c_{\mathrm{PD}}^{init} & : j = 0 \\ \Delta_{\mathrm{PD}}(\sigma(j-1), VDS^{j-1}.eifi, VDS^{(j-1,\sigma)}.c_{\mathrm{PD}}) & : otherwise \end{cases}$$

The output sequence is produced by the application of the output function:

$$VDS^{(j,\sigma)}.eifo \triangleq \begin{cases} [] & : j = 0 \\ VDS^{(j-1,\sigma)}.eifo \circ & \\ \quad \Omega_{\mathrm{PD}}(\sigma(j-1), VDS^{(j-1,\sigma)}.c_{\mathrm{PD}}, VDS^{j-1}.eifi) & : otherwise \end{cases}$$

Since the internal interfaces $difi$, eev, and $difo_s$ are not visible anymore, we introduce extra notions to access their values.

Definition 5.8

$$VDS^{(j,\sigma)}.difi \triangleq \begin{cases} difi^{\epsilon} & : j = 0 \vee \sigma(j-1) \neq P \\ \Omega_{\mathrm{P}}(VDS^{(j-1,\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{P}}) & : otherwise \end{cases}$$

$$VDS^{(j,\sigma)}.eev \triangleq \begin{cases} eev^{\epsilon} & : j = 0 \\ (\Delta_{\mathrm{D}}(\sigma(j-1), VDS^{(j-1,\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{D}}, & \\ \quad VDS^{j-1}.eifi, VDS^{(j,\sigma)}.difi)).eev & : otherwise \end{cases}$$

$$VDS^{(j,\sigma)}.difo_s \triangleq \begin{cases} difo_s{}^{\epsilon} & : j = 0 \\ (\Delta_{\mathrm{D}}(\sigma(j-1), VDS^{(j-1,\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{D}}, & \\ \quad VDS^{j-1}.eifi, VDS^{(j,\sigma)}.difi)).difo_s & : otherwise \end{cases}$$

We employ the shorthand VDS^{σ} to denote $VDS^{(len(\sigma),\sigma)}$. For example, $VDS^{\sigma}.c_{\mathrm{PD}}.c_{\mathrm{P}}$ denotes the state of the processor after processing the complete computational sequence σ.

The following lemma obviously holds.

Lemma 5.3

$$\forall \sigma_1 \prec \sigma.\ VDS^{(len(\sigma_1),\sigma)} = VDS^{\sigma_1}$$

5.3 The Correctness Criterion

Our goal is to state and to prove that every run of the implementation (*VDI* model) can be simulated by a run of the specification (*VDS* model). We use the scheduling function sI_{PD}, introduced in the previous chapter, to relate the time notion of these two models.

The models *VDI* and *VDS* communicate with the external environment, hence, the correctness criterion can only be stated if both models receive equivalent inputs. Fortunately, this communication in the *VDI* and *VDS* models is literally the same as in the *DI* and *DS* models respectively. Therefore, we can reuse the results from the previous chapter, i.e. we employ the predicates *sync_eifis* and *sync_eifos* to test whether the inputs and outputs of both models are synchronized. In general, our correctness criterion combines the correctness of the VAMP processor and the device model.

We use the following notation to keep the formulation of the correctness criterion and its proof compact and readable.

Definition 5.9
Let σ^t be a computational sequence up to a given cycle t, i.e. $\sigma^t = sI_{\mathrm{PD}}(t)$.

$$
\begin{aligned}
CC_{cp}(T) &\triangleq \forall t \leq T.\ (t = 0 \vee JISR^{t-1}) \longrightarrow \\
&\qquad Rconf(VDI^t.h_{\mathrm{P}}, VDS^{\sigma^t}.c_{\mathrm{PD}}.c_{\mathrm{P}}) \wedge\ M(VDI, t) = VDS^{\sigma^t}.c_{\mathrm{PD}}.c_{\mathrm{P}}.M \\
CC_{cd}(T) &\triangleq \forall t \leq T.\ sim_D(VDI^t.h_{\mathrm{D}}, VDS^{\sigma^t}.c_{\mathrm{PD}}.c_{\mathrm{D}}) \\
CC_{eifi}(T) &\triangleq sync\_eifis(0, T, VDI.eifis, VDS.eifi) \\
CC_{eifo}(T) &\triangleq sync\_eifos(0, T, VDI.eifos, VDS^{\sigma^T}.eifo)
\end{aligned}
$$

In Section 3.3.4 we introduced several software conditions. They restrict the assembly programs which can be executed on the VAMP processor. These conditions are defined for the *ISA* model and we need equivalent conditions for the *VDS* model. We omit the formal definitions of these conditions for *VDS* model because they are straightforward reformulations of the ones presented in Section 3.3.4.

Theorem 5.4 (The simulation theorem)
Let the inputs for the VDS model and the VDI model be synchronised and let the initial states be equivalent. Let the assembly code satisfy the software conditions (Section 3.3.4). Let $\sigma^T = sI_{PD}(T)$. Let the reset signal be inactive for all hardware cycles.

We show that the state of the VDI model after T cycles and the state of the VDS after executing σ^T are equivalent. Moreover, the VDI and the VDS models produce equal outputs.

$$
\begin{aligned}
&CC_{cp}(0)\ \wedge CC_{cd}(0)\ \wedge CC_{eifi}(T) \Longrightarrow \\
&CC_{cp}(T)\ \wedge CC_{cd}(T)\ \wedge CC_{eifo}(T)
\end{aligned}
$$

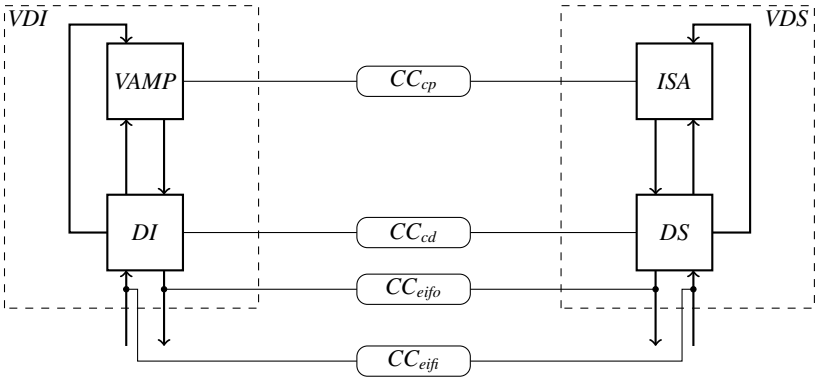


Figure 5.3: The correctness criterion for the combined system.

We graphically represent this theorem in Figure 5.3. On the left-hand side, we schematically depict the VAMP processor combined with the devices, i.e. the *VDI* model.[2] On the right-hand side, we depict the specification model *VDS*.

The assumption of the theorem, which is the relation between inputs from the external environment, is shown in rounded-corner boxes. In the same style we depict relations between the states of the systems and their outputs. In the following sections, we extend this picture to illustrate the proof of Theorem 5.4.

5.4 Overview of Models

In this section we present an overview of all introduced models and relations between them. We have defined *six* different models with *three* different notions of time:

- Gate-level implementations (time notion: hardware cycles)
  - the VAMP processor
  - the external devices
  - the combined VAMP-Devices model
- Instruction-oriented processor specification ISA (time notion: instructions)
- Sequential specification of the device model (time notion: steps of computational sequences)
- Sequential specification of the combined system (time notion: steps of computational sequences)

[2]We don't depict memory interfaces *mifi* and *mifo* to avoid clutter in the figure.

The models and the relations between them can be best depicted graphically (see Figure 5.4). In this figure we show models and relations that we have introduced so far and several new relations. We also blend in Figure 5.4 the relations of Theorem 5.4.

- The VAMP processor is related to the instruction-oriented *ISA* model via the scheduling function sI. The corresponding relations specify how the states, inputs, and outputs of these two models are related with each other in the scope of the combined models. We depict these relations via diamonds and name them P_c, P_{eev}, P_{difo}, and P_{difi}. We formally define these relations in the next section.

- The device implementation *DI* is related to the device specification *DS* via D_c, D_{eev}, D_{difo}, D_{difi}, D_{eifi}, and D_{eifo}. These are depicted via hexagons. Again, we formally define these relations in the next section.

In the proof of Theorem 5.4 we will use results of the previous two chapters. This proof requires several additional relations. These are depicted on the right half of Figure 5.4:

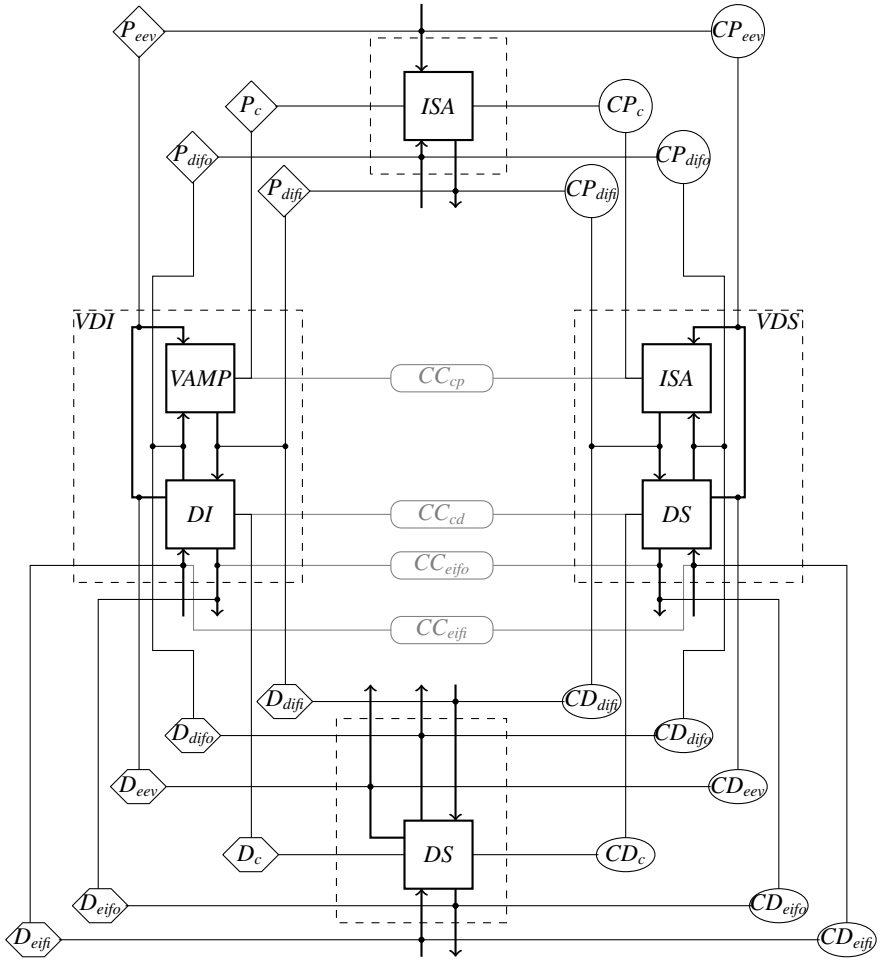
- A run of the processor of the *VDS* has to be simulated by a run of the *ISA*. We denote the needed relations via circles, and we name them CP_c, CP_{eev}, CP_{difo}, and CP_{difi}.

- A run of the devices of the *VDS* has to be simulated by a run of the *DS*. We denote the needed relations via ellipses, and we name them CD_c, CD_{eev}, CD_{difo}, CD_{difi}, CD_{eifi}, and CD_{eifo}.

In Section 5.5 we develop these new relations.

5.5 Model Relations

In Section 3.3 we established the correctness of the VAMP processor against its instruction-oriented specification, the *ISA*. We briefly summarise it here with respect to the combined system, and we define the shorthands for the relations of the processor models:



– relates cycle-based model with sequence-based model
– relates cycle-based model with instruction-based model
– relates instruction-based model with sequence-based model
– relates sequence-based models

Figure 5.4: The overview of the models and the relations.

Definition 5.10

$$P_c(T) \triangleq \forall t \leq T.\ (t = 0 \vee JISR^{t-1}) \longrightarrow Rconf(VDI^t.h_{\mathrm{P}}, ISA^{sI(t)}.c_{\mathrm{P}}) \wedge M(VDI, t) = ISA^{sI(t)}.c_{\mathrm{P}}.M$$

$$P_{eev}(T) \triangleq Reev(T, VDI.eev, ISA.eev)$$

$$P_{difo}(T) \triangleq Rdifo(T, VDI.difo, ISA.difo_s)$$

$$P_{difi}(T) \triangleq Rdifi(T, VDI.difi, ISA.difi)$$

The rewriting rule which expresses the VAMP processor in terms of the *VDI* model (Lemma 5.2) and the VAMP correctness theorem (Theorem 3.10) imply the following corollary.

Corollary 5.5

$$P_c(0) \ \wedge P_{eev}(T) \ \wedge P_{difo}(T) \Longrightarrow P_c(T) \ \wedge P_{difi}(T)$$

In Section 4.3 we introduced the relations between the implementation of the device model *DI* and its specification *DS*. Now we introduce the corresponding shorthands:

Definition 5.11

$$\begin{aligned}
D_c(T) &\triangleq \forall t \leq T.\ sim_D(VDI^t.h_{\mathrm{D}}, DS^{\sigma^t}.c_{\mathrm{D}}) \\
D_{eev}(T) &\triangleq \forall t \leq T.\ VDI^t.eev = DS^{\sigma^t}.eev \\
D_{difo}(T) &\triangleq \forall t \leq T. VDI^t.difo.dout = DS^{\sigma^t}.difo_s \\
D_{difi}(T) &\triangleq sync_difi(0, T, VDI.difi, DS.difi) \\
D_{eifi}(T) &\triangleq sync_eifis(0, T, VDI.eifis, DS.eifi) \\
D_{eifo}(T) &\triangleq sync_eifos(0, T, VDI.eifos, DS^{\sigma^T}.eifo)
\end{aligned}$$

We derive the following corollary from the correctness of the device model (Theorem 4.20) and the rewriting rule for the devices in the scope of the *VDI* model (Lemma 5.1).

Corollary 5.6

$$D_c(0) \ \wedge D_{difi}(T) \wedge D_{eifi}(T) \Longrightarrow D_c(T) \wedge D_{eev}(T) \wedge D_{difo}(T) \wedge D_{eifo}(T)$$

We introduce two additional functions, *Seq2I* and *Seq2P*, to relate the *ISA* model and the processor of the *VDS* model.

The function *Seq2I* counts the number of the processor steps in a given computational sequence up to a given step j. It is defined recursively as follows:

Definition 5.12

$$Seq2I(j,\sigma) \triangleq \begin{cases} 0 & : \ j = 0 \vee \sigma = [] \\ Seq2I(j-1,\sigma)+1 & : \ \sigma(j-1) = P \\ Seq2I(j-1,\sigma) & : \ otherwise \end{cases}$$

We use the shorthand $Seq2I(\sigma)$ to denote $Seq2I(len(\sigma),\sigma)$.

For the rest of this thesis we say that *instruction i is in the computation sequence* if there exists at least $i+1$ numbers of processor identifiers in the sequence. Recall that the first instruction has index 0. Formally,

$$\text{instruction } i \text{ is in } \sigma \iff Seq2I(len(\sigma)) > i$$

The function *Seq2I* establishes a connection between the scheduling functions sI_{PD} and sI.

Lemma 5.7

Let $\sigma^t = sI_{PD}(t)$. If there is an instruction in the processor pipeline with a finished device access, then the number of processor steps counted by Seq2I is greater by one than the number of processor steps counted by sI(t); otherwise the numbers of processor steps counted by Seq2I and sI match.

$$Seq2I(\sigma^t) = \begin{cases} sI(t)+1 & : \ \exists t' < t.\ da^t \ \wedge \forall t'' \in]t : t'[.\ \neg wb^{t''} \\ sI(t) & : \ otherwise \end{cases}$$

The function *Seq2I* is monotonic with respect to a run of the gate-level model:

Lemma 5.8

$$t < t' \implies Seq2I(\sigma^t) \leq Seq2I(\sigma^{t'})$$

The function *Seq2P* returns, for a given instruction index i, the length of the minimal prefix of a computational sequence such that instruction i is in it. For example, for the instruction with index 0, *Seq2P* returns the length of the prefix of a computational–sequence where the last element is the first processor identifier in the sequence.

Definition 5.13

Let j be a given number of specification steps. Let σ be a computational sequence and let i be an instruction index.

$$Seq2P(j,\sigma,i) \triangleq \begin{cases} 0 & : \ j = 0 \vee \sigma = [] \\ j & : \ \sigma(j-1) = P \wedge i = Seq2I(j-1,\sigma) \\ Seq2P(j-1,\sigma,i) & : \ otherwise \end{cases}$$

We use the shorthand $Seq2P(\sigma, i)$ to denote $Seq2P(len(\sigma), \sigma, i)$. The following lemmata describe the main properties of *Seq2P*.

Lemma 5.9
Let instruction i be in the sequence σ. Function Seq2P returns the length of the minimal prefix of a computational sequence with instruction i.

$$Seq2I(len(\sigma)) > i \Longrightarrow$$
$$Seq2P(\sigma, i) = min\{k \mid k < len(\sigma) \ \wedge \sigma(k-1) = P \ \wedge Seq2I(k, \sigma) > i\}$$

Lemma 5.10
If instruction i is not in the sequence, the result of $Seq2P(\sigma, i)$ is zero.

$$Seq2P(\sigma, i) = 0 \Longleftrightarrow Seq2I(len(\sigma)) \leq i$$

Lemma 5.11
$Seq2P(\sigma, i) - 1$ points only to processor identifiers.

$$\forall i < Seq2I(\sigma).\ \sigma(Seq2P(\sigma, i) - 1) = P$$

We can express the position of the last processor identifier in the sequence before j via the *Seq2I* and *Seq2P* functions. We first introduce an auxiliary function *lastP*. It returns the index of the last processor identifier in a given computational sequence:

Definition 5.14
$$lastP(j, \sigma) \triangleq last(j, \sigma, (\lambda x.\ x = P))$$

We use the the shorthand $lastP(\sigma)$ to denote $lastP(len(\sigma), \sigma)$.

Lemma 5.12
Let j be the number of specification steps and let σ be a computational sequence.

$$j \leq len(\sigma) \wedge \ \exists k < j.\ \sigma(k) = P$$
$$\Longrightarrow$$
$$Seq2P(\sigma, Seq2I(j, \sigma) - 1) = lastP(j, \sigma) + 1$$

Lemma 5.13
Let σ be a computational sequence. The prefix with the length $Seq2P(\sigma, i-1)$ contains i processor identifiers.

$$\forall i \leq Seq2I(\sigma).\ \ Seq2I(Seq2P(\sigma, i-1), \sigma) = i$$

We use the function *Seq2P* to locate processor steps in runs of the *VDS* model. For example, $VDS^{(Seq2P(\sigma,i),\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{P}}$ is the processor configuration after execution of the instruction with index i. Similarly, $VDS^{(Seq2P(\sigma,i),\sigma)}.\{eev, difo\}$ are inputs for instruction i and $VDS^{(Seq2P(\sigma,i),\sigma)}.difi$ is the output of execution of instruction i.

Now we define relations between the *ISA* and the processor of the *VDS* system.

Definition 5.15

$$CP_c(\sigma) \quad \triangleq \quad \forall i \leq Seq2I(\sigma).\ ISA^i.c_{\mathrm{P}} = VDS^{(Seq2P(\sigma,i-1),\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{P}}$$

$$CP_{eev}(\sigma) \quad \triangleq \quad \forall i < Seq2I(\sigma).\ ISA^i.eev = VDS^{(Seq2P(\sigma,i),\sigma)}.eev$$

$$CP_{difo}(\sigma) \quad \triangleq \quad \forall i < Seq2I(\sigma).\ ISA^i.difo_s = VDS^{(Seq2P(\sigma,i),\sigma)}.difo_s$$

$$CP_{difi}(\sigma) \quad \triangleq \quad \forall i < Seq2I(\sigma).\ ISA^{i+1}.difi = VDS^{(Seq2P(\sigma,i),\sigma)}.difi$$

We state that every run of the processor of the *VDS* model can be simulated by a run of the *ISA* model.

Theorem 5.14

$$CP_c([]) \wedge CP_{eev}(\sigma) \wedge CP_{difo}(\sigma) \Longrightarrow CP_c(\sigma) \wedge CP_{difi}(\sigma)$$

Proof. *We prove this statement by induction on the length of σ. The base case is trivial because we assume the initial states to be equal, and the outputs have the same idle value. In the induction step the sequence is extended by some element x:*

$$\begin{array}{l} CP_c(\sigma) \wedge CP_{difi}(\sigma) \wedge CP_{eev}(\sigma \circ [x]) \wedge CP_{difo}(\sigma \circ [x]) \\ \Longrightarrow \\ CP_c(\sigma \circ [x]) \wedge CP_{difi}(\sigma \circ [x]) \end{array}$$

We test whether the element x is the processor identifier.

Case 1: *If $x \neq P$, we know that*

- $Seq2I(\sigma \circ [x]) = Seq2I(\sigma)$,
- $\forall i.\ Seq2P(\sigma \circ [x], i) = Seq2P(\sigma, i)$, *and*
- *the processor part of the VDS does not make a step (Definition 5.6)*

Therefore, we can use the induction hypothesis to finish the proof of this case.

Case 2: *If $x = P$, the processors in both models make a step by applying the same step function Δ_P and produce outputs by applying the same output function Ω_P. The induction hypothesis guarantees that the step is done from the same state and the assumptions (CP_{eev} and CP_{difo}) guarantee the equivalence of the inputs for these steps. Thus, the updated processor states and the produced outputs are the same in both models.* □

Lemma 5.15
Let $J \leq len(\sigma)$. Let $I \in \mathbb{N}$ be less or equal J, i.e. $I \leq J$. Steps of the VDS model with device identifiers don't affect the processor configuration:

$$\forall j \in [I : J[.\ \sigma(j) \neq P \implies VDS^{(I,\sigma)}.c_{PD}.c_P = VDS^{(J,\sigma)}.c_{PD}.c_P$$

We can express this lemma in terms of the function *lastP*.

Corollary 5.16
At any given step, the processor configuration of the VDS model equals the one after processing the last step with a processor identifier.

$$\begin{aligned} & j \leq len(\sigma)\ \wedge\ \exists k < j.\ \sigma(k) = P \\ & \implies \\ & VDS^{(j,\sigma)}.c_{PD}.c_P = VDS^{(lastP(j,\sigma)+1,\sigma)}.c_{PD}.c_P \end{aligned}$$

The *difi*-outputs of the *VDS* model depend only on processor configuration. Therefore, *difi*-outputs are not affected by device steps.

Lemma 5.17
The difi-output $VDS^{(Seq2P(\sigma,i),\sigma)}.difi$ is the output of the instruction with index i. $VDS^{(Seq2P(\sigma,i-1),\sigma)}.c_{PD}.c_P$ is the processor configuration after execution of the instruction with index $i-1$. Recall that Ω_P is the processor's output-function. The difi-output equals the result of Ω_P applied on the processor configuration after execution of the instruction with index $i-1$.

$$\forall i < Seq2I(\sigma).\ VDS^{(Seq2P(\sigma,i),\sigma)}.difi = \Omega_P(VDS^{(Seq2P(\sigma,i-1),\sigma)}.c_{PD}.c_P)$$

Proof of this lemma is based on the Definition 5.8 and Corollary 5.16.

Since the device specification *DS* and the devices of the *VDS* are defined over computational sequences, we can easily specify the relations between them.

Definition 5.16

$$
\begin{aligned}
CD_c(\sigma) &\triangleq \forall j \le len(\sigma).\ DS^{(j,\sigma)}.c_{\mathrm{D}} = VDS^{(j,\sigma)}.c_{\mathrm{PD}}.c_{\mathrm{D}} \\
CD_{eev}(\sigma) &\triangleq \forall j \le len(\sigma).\ DS^{(j,\sigma)}.eev = VDS^{(j,\sigma)}.eev \\
CD_{difo}(\sigma) &\triangleq \forall j \le len(\sigma).\ DS^{(j,\sigma)}.difo_s = VDS^{(j,\sigma)}.difo_s \\
CD_{difi}(\sigma) &\triangleq \forall j < len(\sigma).\ DS^{j}.difi = VDS^{(j+1,\sigma)}.difi \\
CD_{eifi}(\sigma) &\triangleq \forall j < len(\sigma).\ DS^{j}.eifi = VDS^{j}.eifi \\
CD_{eifo}(\sigma) &\triangleq \forall j \le len(\sigma).\ DS^{(j,\sigma)}.eifo = VDS^{(j,\sigma)}.eifo
\end{aligned}
$$

We can prove that these two models have the same behavior, if the inputs from the processor and the external environment are the same.

Theorem 5.18

$$
\begin{aligned}
&CD_c([]) \wedge CD_{difi}(\sigma) \wedge CD_{eifi}(\sigma) \Longrightarrow \\
&CD_c(\sigma) \wedge CD_{eev}(\sigma) \wedge CD_{eifo}(\sigma) \wedge CD_{eifo}(\sigma)
\end{aligned}
$$

The proof is carried out by induction on the length of the computational sequence, and we omit it here due to its simplicity.

5.6 The Proof

We make Theorem 5.4 inductive by extending it with relations on the internal interfaces.

Definition 5.17 (Internal Interface Relations)

$$
\begin{aligned}
IIR_{eev}(T) &\triangleq \forall t \le T.\ VDI^t.eev = VDS^{\sigma^t}.eev \\
IIR_{difo}(T) &\triangleq \forall t \le T.\ VDI^t.difo.dout = VDS^{\sigma^t}.difo_s \\
IIR_{difi}(T) &\triangleq sync\_difi(0, T, VDI.difi, (\lambda x.\ VDS^{(x+1,\sigma^T)}.difi)) \wedge \\
&\quad\ Rdifi(T, VDI.difi, (\lambda x.\ \Omega_{\mathrm{P}}(VDS^{(Seq2P(\sigma^T,x-2),\sigma^T)}.c_{\mathrm{PD}}.c_{\mathrm{P}})))
\end{aligned}
$$

We define the relations on the outputs from the devices, the relations on external events (IIR_{eev}) and the answers to the processor (IIR_{difo}), as it is provided by the device theory (Section 4.3.4). The relation on the processor requests (IIR_{difi}) is defined by the conjunction of the relation on inputs for the devices (*sync_difi*, Section 4.3.2) and the outputs of the VAMP (*Rdifi*, Section 3.3.3). Note that $x - 2$ in the definition of IIR_{difi} has exactly two reasons: (i) due to Lemma 5.17, and (ii) the definition of *Rdifi* (Definition 3.23).

Now, we extend the main theorem (Theorem 5.4) as follows:

Theorem 5.19

$$\begin{aligned}
&CC_{cp}(0) \ \wedge CC_{cd}(0) \ \wedge CC_{eifi}(T) \\
&\Longrightarrow \\
&CC_{cp}(T) \ \wedge CC_{cd}(T) \ \wedge CC_{eifo}(T) \ \wedge \\
&IIR_{eev}(T) \ \wedge IIR_{difo}(T) \ \wedge IIR_{difi}(T)
\end{aligned}$$

We prove this theorem by induction on hardware cycles. The induction base, $T = 0$, trivially holds, because we assume the initial states to be the same, and the internal buses are initialised with the idle values.

In the induction step, we assume that for all cycles below T the theorem holds, and we prove that it also holds for cycle $T + 1$. We split the induction step into two lemmata: the correctness of the device part and the processor part of the combined system.

Lemma 5.20 (Devices)

$$\begin{aligned}
&IIR_{difi}(T) \ \wedge CC_{eifi}(T+1) \ \wedge CC_{cd}(0) \\
&\Longrightarrow \\
&CC_{cd}(T+1) \ \wedge CC_{eifo}(T+1) \ \wedge IIR_{eev}(T+1) \ \wedge IIR_{difo}(T+1)
\end{aligned}$$

The reader should not be confused by the presence of $CC_{eifi}(T + 1)$ in the assumptions, because it specifies that the inputs for the gate-level transitions *before* cycle $T + 1$ match the ones for the specification, e.g. the inputs for the transition $T \mapsto T + 1$ match the ones for the specification run with σ_T^{T+1}.

Lemma 5.21 (Processor)

$$\begin{aligned}
&IIR_{eev}(T) \ \wedge IIR_{difo}(T) \ \wedge CC_{cp}(0) \\
&\Longrightarrow \\
&CC_{cp}(T+1) \ \wedge IIR_{difi}(T+1)
\end{aligned}$$

Obviously, these two lemmata imply the induction step. In the following sections, we prove these two lemmata.

5.6.1 Proof for the Device Part: Step 1

In this section, we prove the correctness of the device part of the combined system (Lemma 5.20).

Let DS_v be a device specification model (see Section 4.2) with the following properties:

- initial state of DS_v and initial state of the device part of the *VDI* model are in the relation, i.e. the predicate $D_c(0)$ holds,
- inputs for DS_v and inputs for the device part of the *VDI* model are in relation, i.e. the predicates $D_{difi}(T+1)$ and $D_{eifi}(T+1)$ hold.

Corollary 5.6 guarantees that states and outputs of DS_v and the device part of the *VDI* model are in relation, i.e. predicates $D_c(T+1)$, $D_{eev}(T+1)$, $D_{difo}(T+1)$, and $D_{eifo}(T+1)$ hold.

Thus, DS_v is the device specification model as it seen from the *VDI* point of view (see Figure 5.4).

Let σ^{T+1} be computational sequence up to cycle $T+1$. Let DS_s be a device specification model with the following properties:

- initial state of DS_s and initial state of the device part of the *VDS* model are in relation, i.e. the predicate $CD_c([])$ holds,
- inputs for DS_s and inputs for the device part of the *VDS* model are in relation, i.e. the predicates $CD_{difi}(\sigma^{T+1})$ and $CD_{eifi}(\sigma^{T+1})$ hold.

Theorem 5.18 guarantees that states and outputs of DS_v and the device part of the *VDS* model are in relation, i.e. the predicates $CD_c(\sigma^{T+1})$, $CD_{eev}(\sigma^{T+1})$, $CD_{difo}(\sigma^{T+1})$, and $CD_{eifo}(\sigma^{T+1})$ hold.

Thus, DS_s is the device specification model as it seen from the *VDS* point of view (see Figure 5.4).

Let us for only this section assume the following lemma. The premises of this lemma are properties of inputs and initial states of DS_v and DS_s models as well premises of Lemma 5.20. The conclusions of the lemma states that models DS_v and DS_s have equal initial states and their inputs match.

Lemma 5.22

$$
\begin{aligned}
&D_{difi}(T+1) \wedge D_{eifi}(T+1) \wedge D_c(0) \wedge \\
&CD_{difi}(\sigma^{T+1}) \wedge CD_{eifi}(\sigma^{T+1}) \wedge CD_c([]) \wedge \\
&IIR_{difi}(T) \wedge CC_{eifi}(T+1) \wedge CC_{cd}(0) \\
&\Longrightarrow \\
&DS_v{}^{[]}.c_D = DS_s{}^{[]}.c_D \wedge \\
&\forall j < len(\sigma^{T+1}).\ DS_v{}^{j}.eifi = DS_s{}^{j}.eifi \wedge \\
&\forall j < len(\sigma^{T+1}).\ \sigma^{T+1}(j) = P \longrightarrow \\
&\quad (DS_v{}^{j}.difi.req = DS_s{}^{j}.difi.req) \wedge \\
&\quad (DS_v{}^{j}.difi.req \longrightarrow DS_v{}^{j}.difi = DS_s{}^{j}.difi)
\end{aligned}
$$

Having conclusions of Lemma 5.22, we apply Lemma 4.4 to derive that DS_v and DS_s produce the same outputs and their states match.

Now to prove our goal Lemma 5.20, it is enough to prove the the following lemma. In the premises we collect all known information about states and outputs of DS_v and DS_s models: (i) states and outputs of DS_v and DS_s match, (ii) states and outputs of DS_v and device part of *VDI* are in relation (see above), and (iii) states and outputs of DS_s and device part of *VDS* are in relation (see above). The conclusions are the conclusions of Lemma 5.20.

Lemma 5.23

$$
\begin{aligned}
&\forall j \leq len(\sigma^{T+1}).\ DS_v{}^{(j,\sigma^{T+1})}.c_D = DS_s{}^{(j,\sigma^{T+1})}.c_D \wedge \\
&\forall j \leq \sigma^{T+1}.\ DS_v{}^{(j,\sigma^{T+1})}.eev = DS_s{}^{(j,\sigma^{T+1})}.eev \wedge \\
&\forall j \leq \sigma^{T+1}.\ DS_v{}^{(j,\sigma^{T+1})}.difo_s = DS_s{}^{(j,\sigma^{T+1})}.difo_s \wedge \\
&\forall j \leq \sigma^{T+1}.\ DS_v{}^{(j,\sigma^{T+1})}.eifo = DS_s{}^{(j,\sigma^{T+1})}.eifo \wedge \\
&D_c(T+1) \wedge D_{eev}(T+1) \wedge D_{difo}(T+1) \wedge D_{eifo}(T+1) \wedge \\
&CD_c(\sigma^{T+1}) \wedge CD_{eev}(\sigma^{T+1}) \wedge CD_{difo}(\sigma^{T+1}) \wedge CD_{eifo}(\sigma^{T+1}) \wedge \\
&\Longrightarrow \\
&CC_{cd}(T+1) \wedge CC_{eifo}(T+1) \wedge IIR_{eev}(T+1) \wedge IIR_{difo}(T+1)
\end{aligned}
$$

Thus, the proof of Lemma 5.20 requires two auxiliary lemmata: Lemma 5.22 and Lemma 5.23. We prove these lemmata in the following section.

5.6.2 Proof for the Device Part: Step 2

In this section we prove Lemma 5.22 and Lemma 5.23. We split up these lemmata into a number of smaller lemmata according to the input, output, and state components of the device model.

We split up Lemma 5.22 into a lemma for initial states, a lemma for the input channel *eifis*, and a lemma for the input channel *difi*.

Lemma 5.24 (Initial states)

$$D_c(0) \ \wedge CD_c(\sigma^0) \wedge CC_{cd}(0) \Longrightarrow {DS_v}^{[]}.c_D = {DS_s}^{[]}.c_D$$

We omit the proof for this lemma because it is carried out by straightforward unfolding of definitions.

Lemma 5.25 (Input channel *eifis*)

$$\begin{aligned} &CC_{eifi}(T+1) \wedge D_{eifi}(T+1) \wedge CD_{eifi}(\sigma^{T+1}) \Longrightarrow \\ &\forall j < len(\sigma^{T+1}).\ {DS_v}^j.eifi = {DS_s}^j.eifi \end{aligned}$$

Proof. *Let us first unfold all shorthands and consider some step $j < len(\sigma^{T+1})$. We also apply the definition of CD_{eifi} (Definition 5.16) to rewrite ${DS_s}^j.eifi$ to ${VDS}^j.eifi$.*

$$\begin{aligned} &sync\_eifis(0, T+1, VDI.eifis, VDS.eifi) \wedge \\ &sync\_eifis(0, T+1, VDI.eifis, DS_v.eifi) \wedge \\ &j < len(\sigma^{T+1}) \\ &\Longrightarrow \\ &{DS_v}^j.eifi = {VDS}^j.eifi \end{aligned}$$

According to the definition of sync_eifis (Definition 4.6) we make a case distinction on the following condition: j points to a device identifier and there exists a hardware cycle when it was added into σ^{T+1}. Because Lemma 4.9 guarantees the existence of such a hardware cycle, it suffices to do a case split on $\sigma^{T+1}(j) = P$.

Case 1: *$\sigma^{T+1}(j) \neq P$ holds. Applying the definition of sync_eifis we rewrite the assumptions as follows:*

$$\begin{aligned} &{VDI}^{AddedAt(j)}.eifis(\sigma^{T+1}(j)) = {VDS}^j.eifi \wedge \\ &{VDI}^{AddedAt(j)}.eifis(\sigma^{T+1}(j)) = {DS_v}^j.eifi \wedge \\ &j < len(\sigma^{T+1}) \\ &\Longrightarrow \\ &{DS_v}^j.eifi = {VDS}^j.eifi \end{aligned}$$

Obviously this holds.

Case 2: *If $\sigma^{T+1}(j) = P$ holds, $sync\_eifis(0, T+1, VDI.eifis, VDS.eifi)$ guarantees that ${VDS}^j.eifi = eifi^{\epsilon}$. Similarly, $sync\_eifis(0, T+1, VDI.eifis, DS_v.eifi)$ guarantees that ${DS_v}^j.eifi = eifi^{\epsilon}$. Thus, the goal holds.* □

Lemma 5.26 (Input channel *difi*)

$$
\begin{aligned}
&IIR_{difi}(T) \wedge D_{difi}(T+1) \wedge CD_{difi}(\sigma^{T+1}) \Longrightarrow \\
&\forall\, j < len(\sigma^{T+1}).\ \sigma^{T+1}(j) = P \longrightarrow \\
&\quad (DS_v{}^j.difi.req = DS_s{}^j.difi.req) \wedge \\
&\quad (DS_v{}^j.difi.req \longrightarrow DS_v{}^j.difi = DS_s{}^j.difi)
\end{aligned}
$$

Proof. *We unfold the shorthands and consider some step $j < len(\sigma^{T+1})$ such that $\sigma^{T+1}(j) = P$. We also apply the definition of the predicate CD_{difi} to rewrite $DS_s{}^j.difi$ to $VDS^{(j+1,\sigma^{T+1})}.difi$.*

$$
\begin{aligned}
&sync\_difi(0, T, VDI.difi, (\lambda x.\ VDS^{(x+1,\sigma^T)}.difi)) \wedge \\
&Rdifi(T, VDI.difi, (\lambda x.\ \Omega_P(VDS^{(Seq2P(\sigma^T,x-2),\sigma^T)}.c_{PD}.c_P))) \wedge \\
&sync\_difi(0, T+1, VDI.difi, DS_v.difi) \wedge \\
&j < len(\sigma^{T+1}) \ \wedge \sigma^{T+1}(j) = P \\
&\Longrightarrow \\
&(DS_v{}^j.difi.req = VDS^{(j+1,\sigma^{T+1})}.difi.req) \wedge \\
&(DS_v{}^j.difi.req \longrightarrow DS_v{}^j.difi = VDS^{(j+1,\sigma^{T+1})}.difi)
\end{aligned}
$$

Case 1: *Let $j < len(\sigma^T)$. In this case we use the definition of sync_difi (Definition 4.7) to finish the proof.*

Case 2: *Now we know that $j < len(\sigma^{T+1})$ and $j \geq len(\sigma^T)$, thus, j is added during the transition $T \mapsto T+1$. Since the position j in the computational sequence σ^{T+1} corresponds to a processor step, we consider two cases: the step is with and without a device access. The case splitting is carried out according to the definition of sync_difi (Definition 4.7).*

Case 2.1: *Here we consider j as a processor step with a device access, i.e. $\exists t \in [0 : T+1[.\ da^t \wedge t = AddedAt(j)$. We know that j is added during the transition $T \mapsto T+1$, and hence, we have da^T. In this case $sync\_difi(0, T+1, VDI.difi, DS_v.difi)$ guarantees that the specification receives the same data as the gate-level implementation, and the request flag is turned on. Let us unfold the definition of sync_difi for this case (we also drop first premise):*

$$
\begin{aligned}
&Rdifi(T, VDI.difi, (\lambda x.\ \Omega_P(VDS^{(Seq2P(\sigma^T,x-2),\sigma^T)}.c_{PD}.c_P))) \wedge \\
&VDI^T.difi.\{w, a, din\} = DS_v{}^j.difi.\{w, a, din\} \ \wedge DS_v{}^j.difi.req \wedge \\
&j < len(\sigma^{T+1}) \ \wedge \sigma^{T+1}(j) = P \ \wedge j \geq len(\sigma^T) \ \wedge da^T \\
&\Longrightarrow \\
&(DS_v{}^j.difi.req = VDS^{(j+1,\sigma^{T+1})}.difi.req) \wedge \\
&(DS_v{}^j.difi.req \longrightarrow DS_v{}^j.difi = VDS^{(j+1,\sigma^{T+1})}.difi)
\end{aligned}
$$

The semantics of da guarantees that there is a hardware cycle $t_{st} < T$ when the considered device access starts (Assumption 4.5). Now we apply definition of Rdifi (Definition 3.23, first conjunct) to derive the fact that $VDI^{t_{st}}.difi$ is correct. We also simplify the goal by applying premises.

$$
\begin{aligned}
&VDI^{t_{st}}.difi.req \ \wedge \ VDI^{t_{st}}.difi = \Omega_P(VDS^{(Seq2P(\sigma^T, sI(t_{st})-1), \sigma^T)}.c_{PD}.c_P) \wedge \\
&VDI^T.difi.\{w, a, din\} = {DS_v}^j.difi.\{w, a, din\} \ \wedge {DS_v}^j.difi.req \wedge \\
&j < len(\sigma^{T+1}) \ \wedge \sigma^{T+1}(j) = P \ \wedge j \geq len(\sigma^T) \ \wedge da^T \\
&\Longrightarrow \\
&{DS_v}^j.difi = VDS^{(j+1, \sigma^{T+1})}.difi
\end{aligned}
$$

The stability of the processor outputs (Corollary 4.11) guarantees that difi-outputs at cycles t^{st} and T match, i.e. $VDI^{t_{st}}.difi.\{w, a, din\} = VDI^T.difi.\{w, a, din\}$. Moreover, we know that any instruction with a device access must be the oldest in the VAMP pipeline, and hence, there can not be any write backs during the execution of the device access. Therefore, $sI(t_{st}) = sI(T)$ (Lemma 3.5). We apply this knowledge to our goal and unfold one step of the VDS recursive definition:

$$
\begin{aligned}
&VDI^{t_{st}}.difi.req \ \wedge \ VDI^{t_{st}}.difi = \Omega_P(VDS^{(Seq2P(\sigma^T, sI(T)-1), \sigma^T)}.c_{PD}.c_P) \wedge \\
&VDI^T.difi.\{w, a, din\} = {DS_v}^j.difi.\{w, a, din\} \ \wedge {DS_v}^j.difi.req \wedge \\
&j < len(\sigma^{T+1}) \ \wedge \sigma^{T+1}(j) = P \ \wedge j \geq len(\sigma^T) \ \wedge da^T \\
&\Longrightarrow \\
&\Omega_P(VDS^{(Seq2P(\sigma^T, sI(T)-1), \sigma^T)}.c_{PD}.c_P) = \Omega_P(VDS^{(j, \sigma^{T+1})}.c_{PD}.c_P)
\end{aligned}
$$

Thus, to prove current goal it is enough to show equivalence of processor states, i.e. $VDS^{(Seq2P(\sigma^T, sI(T)-1), \sigma^T)}.c_{PD}.c_P = VDS^{(j, \sigma^{T+1})}.c_{PD}.c_P$.

Now, we do case splitting on existence of the processor identifier in σ^T, i.e. is there any written back instruction before T. We do this because we are going to use the function lastP and its result is undefined if there is no processor identifier in σ^T.

Case 2.1.1: *In this case we assume that in σ^T there are no processor identifiers, i.e. σ^T consists of only device identifiers. By Lemma 5.15 we know that device steps don't affect processor state. Thus, $VDS^{(Seq2P(\sigma^T, sI(T)-1), \sigma^T)}.c_{PD}.c_P$ equals the initial state and $VDS^{(j, \sigma^{T+1})}.c_{PD}.c_P$ is initial state as well. Hence, the proof of this subgoal is finished.*

Case 2.1.2: *In this case we assume that there exists a processor identifier in σ^T.*

Lemma 5.7 guarantees $sI(T) = Seq2I(\sigma^T)$. This is because (i) device access can only be started if the instruction is the oldest in the pipeline, (ii) we have da^T, and hence, there cannot be another instruction in the VAMP pipeline with a finished device access, i.e. $\neg(\exists t' < T.\ da^t \ \wedge \forall t'' \in]t : t'[.\ \neg wb^{t''})$.

Lemma 5.11 provides us with $Seq2P(\sigma^T, Seq2I(\sigma^T) - 1) = lastP(\sigma^T) + 1$. Thus,

to prove our goal is the same as to prove the following equality in the conclusions:

$$
\begin{aligned}
&VDI^{t_{st}}.difi.req \ \wedge \ VDI^{t_{st}}.difi = \Omega_P(VDS^{(Seq2P(\sigma^T, sI(T)-1), \sigma^T)}.c_{PD}.c_P) \wedge \\
&VDI^T.difi.\{w, a, din\} = {DS_v}^j.difi.\{w, a, din\} \ \wedge {DS_v}^j.difi.req \wedge \\
&j < len(\sigma^{T+1}) \ \wedge \sigma^{T+1}(j) = P \ \wedge j \geq len(\sigma^T) \ \wedge da^T \\
&\Longrightarrow \\
&VDS^{(lastP(\sigma^T)+1, \sigma^T)}.c_{PD}.c_P = VDS^{(j, \sigma^{T+1})}.c_{PD}.c_P
\end{aligned}
$$

To finish the proof we have to show the equivalence of the processor configurations. By Definition 5.14 (lastP) we have: $\forall k \in [lastP(\sigma^T) + 1, len(\sigma^T)[. \ \sigma^T(k) \neq P$. *Since* $\sigma^{T+1}(j) = P$, *Definition 4.4 provides us with* $\forall l \in [len(\sigma^T) : j[. \ \sigma^{T+1}(l) \neq P$. *Thus, we have* $\forall k \in [lastP(\sigma^T) + 1, j[. \ \sigma^{T+1}(k) \neq P$.

In other words, every index k from the region $[lastP(\sigma^T) + 1, j[$ *points to a device identifier. Lemma 5.16 states that processor configuration is not affected by the device steps. We apply this lemma to prove our goal:* $VDS^{(lastP(\sigma^T), \sigma^T)}.c_{PD}.c_P = VDS^{(j, \sigma^{T+1})}.c_{PD}.c_P$.

Case 2.2: *Let j point to a processor step without a device access, i.e.* $\neg(\exists t \in [0 : T+1[. \ t = AddedAt(j) \wedge da^t)$. *We know that j is added during the transition* $T \mapsto T+1$, *and hence, we have* $\neg da^T$. *Since j points to processor identifier, we must have write back at T, i.e.* wb^T *(Definition 4.4). In this case* $sync\_difi(0, T+1, VDI.difi, DS_v.difi)$ *only guarantees that the request bit is turned off:* $\neg {DS_v}^j.difi.req$. *Thus, to prove this goal we have to show that* $\neg VDS^{(j+1, \sigma^{T+1})}.difi.req$.

We prove this by contradiction: let us assume $VDS^{(j+1, \sigma^{T+1})}.difi.req$. *Now we apply definition of Rdifi (Definition 3.23, third conjunct) to derive the fact that there must be a hardware cycle below T, when the VAMP started device access. Let us note this cycle as* t_{st}. *Recall that we have* wb^T, *therefore, there must be a hardware cycle* $t_{da} \in]t_{st} : T[$, *when devices provide answer to the processor, i.e.* $da^{t_{da}}$. *By Lemma 4.9 we know that j is added at the unique hardware cycle, hence,* $t_{da} = T$. *Now we have the contradiction in the premises* $\neg da^T$ *and* $da^{t_{da}}$. □

Similarly to the proof of Lemma 5.22, we split up Lemma 5.23 into a lemma for states, a lemma for output channel to the processor *difo*, and a lemma for the output channel to external environment *eifos*.

Lemma 5.27 (States c_D)

$$
\begin{aligned}
&D_c(T+1) \ \wedge CD_c(\sigma^{T+1}) \wedge \\
&\forall j \leq len(\sigma^{T+1}).\ {DS_v}^{(j, \sigma^{T+1})}.c_D = {DS_s}^{(j, \sigma^{T+1})}.c_D \\
&\Longrightarrow \\
&CC_{cd}(T+1)
\end{aligned}
$$

Proof. *We start the proof by unfolding the shorthands:*

$$\begin{aligned}
&\forall t \leq T+1.\ sim_D(VDI^t.h_D, {DS_v}^{\sigma^t}.c_D) \wedge \\
&\forall j \leq len(\sigma^{T+1}).\ {VDS}^{\sigma^{(j,\sigma^{T+1})}}.c_{PD}.c_D = {DS_s}^{(j,\sigma^{T+1})}.c_D \wedge \\
&\forall j \leq len(\sigma^{T+1}).\ {DS_v}^{(j,\sigma^{T+1})}.c_D = {DS_s}^{(j,\sigma^{T+1})}.c_D \\
&\Longrightarrow \\
&\forall t \leq T+1.\ sim_D(VDI^t.h_D, {VDS}^{\sigma^t}.c_{PD}.c_D)
\end{aligned}$$

Let us consider an arbitrary cycle $t \leq T+1$. We instantiate the for all quantifier of second and third premises with the length of the computational sequence up to t, i.e. with $len(\sigma^t)$. This is possible because $t \leq T+1 \longrightarrow len(\sigma^t) \leq len(\sigma^{T+1})$ by Lemma 4.8. Thus, we rewrite the goal as follows:

$$\begin{aligned}
&sim_D(VDI^t.h_D, {DS_v}^{\sigma^t}.c_D) \wedge \\
&{VDS}^{(len(\sigma^t),\sigma^{T+1})}.c_{PD}.c_D = {DS_s}^{(len(\sigma^t),\sigma^{T+1})}.c_D \wedge \\
&{DS_v}^{(len(\sigma^t),\sigma^{T+1})}.c_D = {DS_s}^{(len(\sigma^t),\sigma^{T+1})}.c_D \wedge \\
&t \leq T+1 \\
&\Longrightarrow \\
&sim_D(VDI^t.h_D, {VDS}^{\sigma^t}.c_{PD}.c_D)
\end{aligned}$$

Since ${DS_v}^{(len(\sigma^t),\sigma^{T+1})}$ is equivalent to ${DS_v}^{take(len(\sigma^t),\sigma^{T+1})}$ and $take(len(\sigma^t),\sigma^{T+1}) = \sigma^t$, we can conclude ${DS_v}^{(len(\sigma^t),\sigma^{T+1})} = {DS_v}^{\sigma^t}$. Similarly we conclude ${VDS}^{(len(\sigma^t),\sigma^{T+1})} = {VDS}^{\sigma^t}$. Hence, the conclusion holds. □

The proofs of lemmata for the *eev* and *difo* output channels are employed the same argumentations as in the previous proof. Therefore, we omit these proofs.

Lemma 5.28 (Output channel *eev*)

$$\begin{aligned}
&D_{eev}(T+1) \wedge CD_{eev}(\sigma^{T+1}) \wedge \\
&\forall j \leq \sigma^{T+1}.\ {DS_v}^{(j,\sigma^T)}.eev = {DS_s}^{(j,\sigma^T)}.eev \\
&\Longrightarrow \\
&IIR_{eev}(T+1)
\end{aligned}$$

Lemma 5.29 (Output channel *difo*)

$$\begin{aligned}
&D_{difo}(T+1) \wedge CD_{difo}(\sigma^{T+1}) \wedge \\
&\forall j \leq \sigma^{T+1}.\ {DS_v}^{(j,\sigma^{T+1})}.difo_s = {DS_s}^{(j,\sigma^{T+1})}.difo_s \\
&\Longrightarrow \\
&IIR_{difo}(T+1)
\end{aligned}$$

Now, we are left with the a lemma for the outputs to the external environment.

Lemma 5.30 (Output channel *eifos*)

$$
\begin{array}{l}
D_{eifo}(T+1) \ \wedge CD_{eifo}(\sigma^{T+1}) \wedge \\
\forall\, j \leq \sigma^{T+1}.\ DS_v{}^{(j,\sigma^{T+1})}.eifo = DS_s{}^{(j,\sigma^{T+1})}.eifo \\
\Longrightarrow \\
CC_{eifo}(T+1)
\end{array}
$$

This proof is only slightly different from the previous ones, therefore, we omit it.

5.6.3 Proof for the Processor Part: Step 1

In this section, we prove the correctness of the processor part of the combined system (Lemma 5.21). We apply the same strategy which we have used in Section 5.6.1.

Let ISA_v be an *ISA* model (see Section 3.1) with the following properties:

- initial state of ISA_v and initial state of the processor of the *VDI* model are in the relation, i.e. the predicate $P_c(0)$ holds.
- inputs for ISA_v and inputs for the processor of the *VDI* model are in relation, i.e. the predicates $P_{eev}(T+1)$ and $P_{difo}(T+1)$ hold.

Corollary 5.5 guarantees that states and outputs of ISA_v and the processor of the *VDI* model are in relation, i.e. predicates $P_{difi}(T+1)$ and $P_c(T+1)$ hold.

Thus, ISA_v is the processor specification as it seen from the *VDI* point of view (see Figure 5.4).

Let σ^{T+1} be computational sequence up to cycle $T+1$. Let ISA_s be an *ISA* model with the following properties:

- initial state of ISA_s and initial state of the processor of the *VDS* model are in relation, i.e. the predicate $CP_c([])$ holds.
- inputs for ISA_s and inputs for the processor of the *VDS* model are in relation, i.e. the predicates $CP_{eev}(\sigma^{T+1})$ and $CP_{difo}(\sigma^{T})$ hold.

Theorem 5.14 guarantees that states and outputs of ISA_s and the processor of the *VDS* model are in relation, i.e. the predicates $CP_{difi}(\sigma^{T+1})$ and $CP_c(\sigma^{T+1})$ hold.

Thus, ISA_s is the processor specification as it seen from the *VDS* point of view (see Figure 5.4).

Let us for only this section assume the following lemma. The premises of this lemma are properties of inputs and initial states of ISA_v and ISA_s models as well premises of Lemma 5.21. The conclusions of the lemma states that models ISA_v and ISA_s have equal initial states and their inputs match.

Lemma 5.31

$$
\begin{aligned}
&P_{eev}(T+1) \ \wedge P_{difo}(T+1) \ \wedge \ P_c(0) \wedge \\
&CP_{eev}(\sigma^{T+1}) \ \wedge CP_{difo}(\sigma^{T+1}) \ \wedge CP_c([]) \wedge \\
&IIR_{eev}(T) \ \wedge IIR_{difo}(T) \ \wedge CC_{cp}(0) \\
&\Longrightarrow \\
&{ISA_v}^0.c_P = {ISA_s}^0.c_P \wedge \forall i < sI(T+1).\ {ISA_v}^i.eev = {ISA_s}^i.eev \\
&\forall i < sI(T+1).\ {ISA_v}^i.difi.req \longrightarrow {ISA_v}^i.difo_s = {ISA_s}^i.difo_s
\end{aligned}
$$

Having conclusions of Lemma 5.31, we apply Lemma 3.2 to derive that ISA_v and ISA_s produce the same outputs and their states match.

Now to prove our goal Lemma 5.21, it is enough to prove the the following lemma. In the premises we collect all known information about states and outputs of ISA_v and ISA_s models: (i) states and outputs of ISA_v and ISA_s match, (ii) states and outputs of ISA_v and the processor of *VDI* are in relation (see above), and (iii) states and outputs of ISA_s and the processor of *VDS* are in relation (see above). The conclusions are the conclusions of Lemma 5.21.

Lemma 5.32

$$
\begin{aligned}
&\forall i \leq sI(T+1).\ {ISA_v}^i.c_P = {ISA_s}^i.c_P \wedge \\
&\forall i \leq sI(T+1)+1.\ {ISA_v}^i.difi = {ISA_s}^i.difi \wedge \\
&P_c(T+1) \wedge P_{difi}(T+1) \wedge \\
&CP_c(\sigma^{T+1}) \ \wedge CP_{difi}(\sigma^{T+1}) \\
&\Longrightarrow \\
&CC_{cp}(T+1) \ \wedge IIR_{difi}(T+1)
\end{aligned}
$$

Thus, the proof of Lemma 5.21 requires two auxiliary lemmata: Lemma 5.31 and Lemma 5.32. We prove these lemmata in the following section.

5.6.4 Proof for the Processor Part: Step 2

In this section we prove Lemma 5.31 and Lemma 5.32. We split up these lemmata into a number of smaller lemmata according to the input, output, and state components of the processor model.

We split up Lemma 5.31 into a lemma for initial states, a lemma for the input channel *eev*, and a lemma for the input channel $difo_s$.

Lemma 5.33 (Initial states)

$$P_c(0) \ \wedge CP_c(\sigma^0) \ \wedge CC_{cp}(0) \Longrightarrow {ISA_v}^0.c_P = {ISA_s}^0.c_P$$

We omit the proof for this lemma because it is carried out by straightforward unfolding of definitions.

Lemma 5.34 (Input channel *eev*)

$$\begin{array}{l} IIR_{eev}(T) \ \wedge P_{eev}(T+1) \ \wedge CP_{eev}(\sigma^{T+1}) \\ \Longrightarrow \\ \forall i < sI(T+1).\ {ISA_v}^i.eev = {ISA_s}^i.eev \end{array}$$

Proof. *First we unfold shorthands and consider some instruction $i < sI(T+1)$:*

$$\begin{array}{l} (\forall t \leq T.\ VDI^t.eev = VDS^{\sigma^t}.eev) \wedge \\ (Reev(T+1, VDI.eev, ISA_v.eev)) \wedge \\ (\forall i < Seq2I(\sigma^{T+1}).\ {ISA_s}^i.eev = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.eev) \wedge \\ i < sI(T+1) \\ \Longrightarrow \\ {ISA_v}^i.eev = {ISA_s}^i.eev \end{array}$$

Lemma 3.8 guarantees that there is a hardware cycle $t < T+1$ at which instruction i is written back, i.e. $sI(t) = i \wedge wb^t$. Lemma 5.7 guarantees $sI(T+1) \leq Seq2I(\sigma^{T+1})$. The latter implies that $i < Seq2I(\sigma^{T+1})$, and hence, we can instantiate the for all quantifier of the third premise with i.

Case 1: *Let us consider the case where instruction i does not access any device. For such an instruction $Reev(T+1, VDI.eev, ISA_v.eev)$ (Definition 3.21) guarantees that the interrupts sampled on the bus at cycle t match the ones for the ISA_v, i.e. $VDI^t.eev = {ISA_v}^i.eev$. We instantiate the for all quantifier of the first premise with t:*

$$\begin{array}{l} VDI^t.eev = VDS^{\sigma^t}.eev \wedge \\ VDI^t.eev = {ISA_v}^i.eev \wedge \\ {ISA_s}^i.eev = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.eev \wedge \\ i < sI(T+1) \ \wedge i = sI(t) \ \wedge wb^t \\ \Longrightarrow \\ {ISA_v}^i.eev = {ISA_s}^i.eev \end{array}$$

After a sequence of simple rewrites we have the goal as follows (we drop unnecessary premises):

$$VDS^{\sigma^t}.eev = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.eev$$

We know that $Seq2P(\sigma^{T+1}, i)$ *points right behind the position in the sequence where the processor executes an instruction with the index* $i = sI(t)$*. Let us find out where it exactly points to. Since we have the write back at t,* $sI(t+1) - 1 = sI(t)$ *holds (Definition 3.20). The active value of* wb^t *also implies* $Seq2I(\sigma^{t+1}) = sI(t+1)$ *(Lemma 5.7). Thus,* $Seq2P(\sigma^{T+1}, i) = Seq2P(\sigma^{T+1}, Seq2I(\sigma^{t+1}) - 1)$*. By Lemma 5.12* $Seq2P(\sigma^{T+1}, Seq2I(\sigma^{t+1}) - 1) = lastP(\sigma^{t+1})$*.*

Since at cycle t we write back an instruction without a device access, sI_{PD} *adds the processor identifier in* σ_t^{t+1}*. Moreover, it adds this processor identifier at the beginning of this subsequence (Definition 4.4), i.e. the first element right after* σ^t*. We conclude that* $Seq2P(\sigma^{T+1}, i)$ *points right behind the end of the sequence* $\sigma^t \circ [P]$*, i.e.* $Seq2P(\sigma^{T+1}, sI(t)) = len(\sigma^t \circ [P])$*. Thus, we rewrite the goal as follows:*

$$\begin{aligned} VDS^{\sigma^t}.eev &= VDS^{(len(\sigma^t \circ [P]),\sigma^{T+1})}.eev \\ &\quad \text{by Lemma 5.3 and } (\sigma^t \circ [P]) \prec \sigma^{T+1} \\ &= VDS^{(\sigma^t \circ [P])}.eev \end{aligned}$$

Recall that we are proving the case where the considered instruction does not access devices. Remember also that a step of the device specification with the processor identifier and without a processor request does not have any effect (Assumption 4.3). Therefore, $VDS^{\sigma^t}.eev = VDS^{(\sigma^t \circ [P])}.eev$*, and it concludes the proof of this case.*

Case 2: *Now we consider the case where the processor writes back an instruction with a device access. For such an instruction, the VAMP uses the external interrupts which have been sampled on the eev-bus at the end of the device access and have been saved in the eev_da register. The bus between VAMP and gate-level devices introduces one cycle delay (Definition 5.1). Therefore, this sampled value is computed by the gate-level devices in one cycle before the VAMP registers the end of the device access. Moreover, at this previous cycle the predicate da holds, because the device access is finished. Let us summarize: the gate-level devices place the interrupts at cycle* $last_{hw}(t, da)$ *and VAMP samples them at cycle* $last_{hw}(t, da) + 1$*.*

Recall that the register eev_da keeps the sampled value until the instruction with the device access is written back (Lemma 3.4). $Reev(T+1, VDI.eev, ISA_v.eev)$ *guarantees that the sampled interrupts match the ones for the* ISA_v*, i.e.* $VDI^{last_{hw}(t,da)+1}.eev = {ISA_v}^i.eev$*.*

$$\begin{aligned} & VDI^{last_{hw}(t,da)+1}.eev = VDS^{(\sigma^{last_{hw}(t,da)+1})}.eev \wedge \\ & VDI^{last_{hw}(t,da)+1}.eev = {ISA_v}^i.eev \wedge \\ & {ISA_s}^i.eev = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.eev \wedge \\ & i < sI(T+1) \ \wedge sI(t) = i \wedge wb^t \\ & \Longrightarrow \\ & {ISA_v}^i.eev = {ISA_s}^i.eev \end{aligned}$$

After several rewrites we have (we drop unnecessary premises):

$$
\begin{array}{l}
i < Seq2I(\sigma^{T+1}) \ \wedge i = sI(t) \ \wedge wb^{t-1} \\
\Longrightarrow \\
VDS^{(\sigma^{last_{hw}(t,da)+1})}.eev = VDS^{(Seq2P(\sigma^{T},i),\sigma^{T+1})}.eev
\end{array}
$$

We know that $Seq2P(\sigma^{T+1}, i)$ points to the position in the sequence where the processor executes an instruction i. Since this instruction is written back at cycle t and this is an instruction with a device access, by Lemma 5.7 we have $sI(t) = Seq2I(\sigma^t) - 1$. Since $\sigma^t \prec \sigma^{T+1}$, Lemma 5.12 guarantees $Seq2P(\sigma^{T+1}, Seq2I(\sigma^t) - 1) = lastP(\sigma^t) + 1$.

Thus, the considered processor step must be somewhere in σ^t. We know that a device access can only be started if the corresponding instruction is the oldest one in the pipeline. Thus, between cycles $last_{hw}(t, da)+1$ and $t-1$ there can not be active value of the predicate da (Definition 5.14) nor write back signals (Lemma 3.3). Therefore, the last processor identifier in σ^t is added during the transition $last_{hw}(t, da) \mapsto last_{hw}(t, da) + 1$. During this transition, the computational sequence is extended with the processor identifier at the end of $\sigma^{last_{hw}(t,da)+1}$ (Definition 4.4). Thus, $lastP(\sigma^t)+1 = len(\sigma^{last_{hw}(t,da)+1})$. Now we apply this knowledge to our goal:

$$
\begin{array}{l}
i < Seq2I(\sigma^{T+1}) \ \wedge i = sI(t) \ \wedge wb^{t} \\
\Longrightarrow \\
VDS^{\sigma^{last_{hw}(t,da)+1}}.eev = VDS^{(len(\sigma^{last_{hw}(t,da)+1}),\sigma^{T+1})}.eev
\end{array}
$$

Finally, we apply Lemma 5.3 to finish the proof. □

This proof illustrates the problem of sampling external interrupts which we discussed in Chapter 3. If we sampled external interrupts at the write back for all instructions, we would have to prove the following case: $VDS^{\sigma^t}.eev = VDS^{(Seq2P(\sigma^{T},i),\sigma^{T+1})}.eev$. This can be rewritten as follows: $VDS^{\sigma^t}.eev = VDS^{(\sigma^{last_{hw}(t,da)})^{t-last_{hw}(t,da)}}.eev$. The difference $t - last_{hw}(t, da)$ specifies the number of hardware cycles between the end of the device access and the sampling of the *eev*-bus in the write back stage. During these hardware cycles devices, and especially the accessed device, can change their states. Thus, we will sample the interrupts which are produced by the new device states. As a result, these interrupts would not match the ones we have in the specification.

Lemma 5.35 (Input channel *difo*)

$$
\begin{array}{l}
IIR_{difo}(T) \ \wedge P_{difo}(T+1) \ \wedge CP_{difo}(\sigma^{T+1}) \\
\Longrightarrow \\
\forall i < sI(T+1).ISA_v{}^i.difi.req \longrightarrow ISA_v{}^i.difo_s = ISA_s{}^i.difo_s
\end{array}
$$

The proof of this lemma is similar to the second case of the previous proof.

Similarly to the proof of Lemma 5.31, we split up Lemma 5.32 into a lemma for states and a lemma for output channel to devices *difi*.

Lemma 5.36 (States c_P)

$$
\begin{aligned}
&P_c(T+1) \ \wedge CP_c(\sigma^{T+1}) \wedge \\
&\forall i \leq sI(T+1).\ {ISA_v}^i.c_P = {ISA_s}^i.c_P \\
&\Longrightarrow \\
&CC_{cp}(T+1)
\end{aligned}
$$

Proof. *As usual, we unfold the shorthands:*

$$
\begin{aligned}
&(\forall t \leq T+1.\ JISR^{t-1} \longrightarrow Rconf(VDI^t.h_P, {ISA_v}^{sI(t)}.c_P) \wedge \\
&\qquad\qquad\qquad\qquad M(VDI,t) = {ISA_v}^{sI(t)}.c_P.M) \wedge \\
&(\forall i \leq Seq2I(\sigma^{T+1}).\ {ISA_s}^i.c_P = VDS^{(Seq2P(\sigma,i-1),\sigma)}.c_{PD}.c_P) \wedge \\
&(\forall i \leq sI(T+1).\ {ISA_v}^i.c_P = {ISA_s}^i.c_P) \\
&\Longrightarrow \\
&\forall t \leq T+1. JISR^{t-1} \longrightarrow Rconf(VDI^t.h_P, VDS^{\sigma^t}.c_{PD}.c_P) \wedge \\
&\qquad\qquad\qquad\qquad M(VDI,t) = VDS^{\sigma^t}.c_{PD}.c_P.M
\end{aligned}
$$

Let us consider an arbitrary cycle $t \leq T+1$. A closer look at the first assumption and the goal reveals the fact that this lemma holds if ${ISA_v}^{sI(t)}.c_P = VDS^{\sigma^t}.c_{PD}.c_P$.

$$
\begin{aligned}
&t \leq T+1 \ \wedge JISR^{t-1} \wedge \\
&(\forall i \leq Seq2I(\sigma^{T+1}).\ {ISA_s}^i.c_P = VDS^{(Seq2P(\sigma^{T+1},i-1),\sigma^{T+1})}.c_{PD}.c_P) \wedge \\
&(\forall i \leq sI(T+1).\ {ISA_v}^i.c_P = {ISA_s}^i.c_P) \\
&\Longrightarrow \\
&{ISA_v}^{sI(t)}.c_P = VDS^{\sigma^t}.c_{PD}.c_P
\end{aligned}
$$

We instantiate the for all quantifier of the second premise with $sI(t)$. We can do this because $sI(T+1) \leq Seq2I(\sigma^{T+1})$ by Lemma 5.7, and hence, $sI(t) \leq Seq2I(\sigma^{T+1})$. By Lemma 3.6 $sI(t) \leq sI(T+1)$, and hence, we can instantiate the for all quantifier of the third premise with $sI(t)$ as well.

$$
\begin{aligned}
&t \leq T+1 \ \wedge JISR^{t-1} \wedge \\
&{ISA_s}^{sI(t)}.c_P = VDS^{(Seq2P(\sigma^{T+1},sI(t)-1),\sigma^{T+1})}.c_{PD}.c_P) \wedge \\
&{ISA_v}^{sI(t)}.c_P = {ISA_s}^{sI(t)}.c_P \\
&\Longrightarrow \\
&{ISA_v}^{sI(t)}.c_P = VDS^{\sigma^t}.c_{PD}.c_P
\end{aligned}
$$

Since we have $JISR^{t-1}$ (which implies wb^{t-1}) there is no pending write-back of an instruction with a device access at cycle t. Thus, Lemma 5.7 guarantees that $sI(t) = Seq2I(len(\sigma^t), \sigma^{T+1})$. Together with an application of Lemma 5.12 we rewrite the goal as follows (we drop the premises):

$$VDS^{(lastP(len(\sigma^t),\sigma^{T+1})+1,\sigma^{T+1})}.c_{PD}.c_P = VDS^{\sigma^t}.c_{PD}.c_P$$

Since σ^t is a subsequence of σ^{T+1}, $lastP(len(\sigma^t), \sigma^{T+1}) = lastP(\sigma^t)$:

$$VDS^{(lastP(\sigma^t)+1,\sigma^{T+1})}.c_{PD}.c_P = VDS^{\sigma^t}.c_{PD}.c_P$$

By Definition 5.14 (lastP) we have: $\forall j \in [lastP(\sigma^t) + 1, len(\sigma^t)[.\ \sigma^t(j) \neq P$. Thus, all these indices point to device identifiers. Lemma 5.16 states that processor configuration is not affected by the device steps. We apply this lemma to finish the proof. □

Lemma 5.37 (Output channel *difi*)

$$\begin{array}{l} P_{difi}(T+1)\ \wedge CP_{difi}(\sigma^{T+1}) \wedge \\ \forall i \leq sI(T+1)+1.\ {ISA_v}^i.difi = {ISA_s}^i.difi \\ \Longrightarrow \\ IIR_{difi}(T+1) \end{array}$$

Proof. *We first unfold the shorthands:*

$$\begin{array}{l} Rdifi(T+1, VDI.difi, ISA_v.difi) \wedge \\ (\forall i < Seq2I(\sigma^{T+1}).\ {ISA_s}^{i+1}.difi = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.difi) \wedge \\ (\forall i \leq sI(T+1)+1.\ {ISA_v}^i.difi = {ISA_s}^i.difi) \\ \Longrightarrow \\ Rdifi(T+1, VDI.difi, (\lambda x.\ \Omega_P(VDS^{(Seq2P(\sigma^{T+1},x-2),\sigma^{T+1})}.c_{PD}.c_P)) \wedge \\ sync\_difi(0, T+1, VDI.difi, (\lambda x.\ VDS^{(x+1,\sigma^{T+1})}.difi)) \end{array}$$

We split the conclusion into two subgoals.

Case 1: *In this case we prove that Rdifi predicate holds. This predicate consists of three conjuncts (Definition 3.23). We prove every conjunct as a separate subgoal.*

Case 1.1: *In this case we consider the first conjunct from Rdifi:*

$$\begin{array}{l} Rdifi(T+1, VDI.difi, ISA_v.difi) \wedge \\ (\forall i < Seq2I(\sigma^{T+1}).\ {ISA_s}^{i+1}.difi = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.difi) \wedge \\ (\forall i \leq sI(T+1)+1.\ {ISA_v}^i.difi = {ISA_s}^i.difi) \\ \Longrightarrow \\ (\forall t \leq T+1.\ VDI^t.difi.req \longrightarrow VDI^t.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},sI(t)-1),\sigma^{T+1})}.c_{PD}.c_P)) \end{array}$$

Let us consider an arbitrary cycle $t \leq T+1$ with a request, i.e. $VDI^t.difi.req$. Applying this information we can rewrite the goal, using $Rdifi(T+1, VDI.difi, ISA_v.difi)$ and Lemma 5.2, to:

$$ISA_v{}^{sI(t)+1}.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},sI(t)-1),\sigma^{T+1})}.c_{PD}.c_P)$$

Lemma 5.7 and Lemma 3.6 guarantee $sI(t) - 1 < Seq2I(\sigma^{T+1})$. Therefore, we can instantiate the for all quantifier of the second premise with $sI(t) - 1$. By Lemma 3.6 $sI(t) \leq sI(T+1)$, and thus, we rewrite the goal as follows:

$$VDS^{(Seq2P(\sigma^{T+1},sI(t)),\sigma^{T+1})}.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},sI(t)-1),\sigma^{T+1})}.c_{PD}.c_P)$$

We know that the difi output of an instruction $sI(t)$ depends only on the processor configuration after execution instruction $sI(t) - 1$. Therefore, we apply Lemma 5.17 to finish the proof of this case.

Case 1.2: *In this case we consider the second conjunct from the conclusions:*

$$\begin{array}{l}
Rdifi(T+1, VDI.difi, ISA_v.difi) \wedge \\
(\forall i < Seq2I(\sigma^{T+1}).\ ISA_s{}^{i+1}.difi = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.difi) \wedge \\
(\forall i \leq sI(T+1)+1.\ ISA_v{}^{i}.difi = ISA_s{}^{i}.difi) \\
\Longrightarrow \\
(\forall i < sI(T+1).\ \Omega_P(VDS^{(Seq2P(\sigma^{T+1},i-1),\sigma^{T+1})}.c_{PD}.c_P).req \longrightarrow \\
\quad \exists t \leq T+1.\ i = sI(t)\ \wedge VDI^t.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},i-1),\sigma^{T+1})}.c_{PD}.c_P))
\end{array}$$

Let us consider some instruction i such that $\Omega_P(VDS^{(Seq2P(\sigma^{T+1},i-1),\sigma^{T+1})}.c_{PD}.c_P).req$ and $i < sI(T+1)$ hold. From the premises and Lemma 5.17 we can derive that $ISA_v{}^{i+1}.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},i-1),\sigma^{T+1})}.c_{PD}.c_P)$. This is because $i-1 < Seq2I(\sigma^{T+1})$ (Lemma 5.7). Thus, we can rewrite the goal as follows:

$$\exists t \leq T+1.\ i = sI(t)\ \wedge VDI^t.difi = ISA_v{}^{i+1}.difi$$

Now we unfold the definition of Rdifi in the premises to finish the proof.

Case 1.3: *In this case we consider third conjunct from the conclusions.*

$$\begin{array}{l}
Rdifi(T+1, VDI.difi, ISA_v.difi) \wedge \\
(\forall i < Seq2I(\sigma^{T+1}).\ ISA_s{}^{i+1}.difi = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.difi) \wedge \\
(\forall i \leq sI(T+1)+1.\ ISA_v{}^{i}.difi = ISA_s{}^{i}.difi) \\
\Longrightarrow \\
wb^{T+1} \wedge \Omega_P(VDS^{(Seq2P(\sigma^{T+1},sI(T+1)-1),\sigma^{T+1})}.c_{PD}.c_P).req \longrightarrow \\
\quad \exists t \leq T+1.\ VDI^t.difi = \Omega_P(VDS^{(Seq2P(\sigma^{T+1},sI(T+1)-1),\sigma^{T+1})}.c_{PD}.c_P) \wedge \\
\qquad\qquad sI(T+1) = sI(t)
\end{array}$$

The proof for this case is similar to the previous proof with i replaced by $sI(T+1)$.

Case 2: *In this case we prove the second conjunct of* $IIR_{difi}(T+1)$*:*

$$
\begin{aligned}
& Rdifi(T+1, VDI.difi, ISA_v.difi) \wedge \\
& (\forall i < Seq2I(\sigma^{T+1}).\ {ISA_s}^{i+1}.difi = VDS^{(Seq2P(\sigma^{T+1},i),\sigma^{T+1})}.difi) \wedge \\
& (\forall i \leq sI(T+1)+1.\ {ISA_v}^{i}.difi = {ISA_s}^{i}.difi) \\
& \Longrightarrow \\
& sync\_difi(0, T+1, VDI.difi, (\lambda x.\ VDS^{(x+1,\sigma^{T+1})}.difi))
\end{aligned}
$$

The proof for this case is similar to the proof of Lemma 5.26 and we omit it. □

5.7 Summary

In this section we constructed and proved a computer system which consists of the VAMP processor and the generic device model. This system inherits the genericity of the device model, and hence, it can be instantiated with arbitrary concrete devices. In the next chapter we demonstrate how this can be done, and we build a control unit for a distributed automotive system.

Chapter 6

An Electronic Control Unit

In Verisoft subproject [The03] an automotive system is built. The system consists of typical components which are placed and run on a vehicle. The system components are a time-triggered bus system (inspired by FlexRay [Con06]) and a time-triggered operating system (inspired by OSEKTime [Con93]). On top of these system components, an emergency-call application is realized.

FlexRay is a standard for a high-end real-time bus for automotive applications. It is developed and propagated by the FlexRay Consortium consisting of leading automotive manufacturers and suppliers. The FlexRay standard specifies a time-triggered communication bus, and its main part is a communication protocol. This protocol serves two purposes: (i) deterministic, periodic message exchanges and (ii) a clock synchronization for all nodes connected to the bus. Thus, the FlexRay bus connects a fixed number of nodes, where every node is an electronic control unit (ECU). A typical ECU consists of a processor and a hardware which implements the FlexRay protocol.

The goal of TP6 subproject is the development and the verification of an automotive system, which consists of several ECUs connected via FlexRay-like bus. The applications in this system run with a fixed scheduler under OSEKTime-like operating system [DDS08]. A paper&pencil correctness proof of the whole system is presented in [Kna08, KP07]. In this chapter, we concentrate on the correctness of a single ECU. The latter is used to prove the correctness of the complete automotive system.

We build an ECU which consists of the VAMP processor and a bus interface. The bus interface is called the automotive bus controller or the ABC device.

In this chapter, we instantiate the VAMP-Device model with the ABC device. We describe this device and build on its base a device model. We also justify that the built device model fits to the developed generic device theory. Finally, we present a model of the ECU and its verified correctness criterion.

6.1 Bus Controller

The hardware for the ABC device is developed and verified as a part of the TP6 subproject [BBG+05]. The device is placed between a processor and a time-triggered bus, and it can be architecturally split into two parts. One part is only visible from the bus side and another from the processor side. The main components of every part are a send buffer and a receive buffer. Since the buffers on the bus and the processor side are independent, the device can service both sides simultaneously.

The buffers on the bus side are used to hold the outgoing messages to the bus (the send buffer) and to store the incoming messages from the bus (the receive buffer). The buffers on the processor side are memory mapped into the VAMP address space (see Section 3.1). The programmer can send data to other ECUs by writing the data into the send buffer. She/he can also receive data by reading them from the receive buffer. Note that the hardware on the bus side takes care about the sending and the receiving data to/from the bus.

There are special hardware cycles when the roles of the buffers of the processor and the bus sides are swapped, i.e. the buffers of the processor side become the buffers of the bus side and vise versa. This implements the data transfer between the processor and the bus sides of the ABC device.

The communication between the ABC device and the processor is based on interrupts. If the ABC device is ready for communication, it raises an interrupt (a bit on *eev*–bus). The processor executes an interrupt service routine. This routine may read the receive buffer and store the read data in the processor GPR file or main memory, set up the device operation mode, and write data into the send buffer.

The implementation of the ABC device is defined on the gate level. It is designed in such a way that it serves processor requests with no delay [ABK08]. The function δ_{ABC} implements the transition function of the gate level model. Let H_{ABC} denote the type of the ABC device configuration, $Eifi_{\mathrm{ABC}}$ and $Eifo_{\mathrm{ABC}}$ denote the input and the output from/to the bus. Then, δ_{ABC} has the following signature:

$$\delta_{\mathrm{ABC}} : Difi \times Eifi_{\mathrm{ABC}} \times H_{\mathrm{ABC}} \mapsto H_{\mathrm{ABC}} \times Eifo_{\mathrm{ABC}} \times Difo$$

We employ the function $\omega_{\mathrm{ABC}} : H_{\mathrm{ABC}} \mapsto \{0, 1\}$ which tests whether in a given state the interrupt is set. We can determine the swapping of the buffers based on the current state of the ABC device. This test is implemented by the function $SwapBuf : H_{\mathrm{ABC}} \mapsto \mathbb{B}$.

Correctness of a single ECU can not be treated fully decoupled from the rest of the automotive system. During a run of the automotive system, the ECUs exchange data via a time-triggered bus. Such a run is split into so called time slots. One of the main slot characteristics is the absence of buffer swappings. The buffers are swapped at the border of two slots. For example, the processor of an ECU can only read/write data during a slot. Therefore, the correctness of an ECU is split into two parts: a local and a distributed one. The local correctness captures the communication of the processor and the ABC device during one slot. The distributed correctness states that during a run the exchange of the data between ECUs is correct, and that this run can

be decomposed into slots. Obviously the distributed correctness requires the local one. In this thesis, we are only interested in local correctness. Our result is reused for the distributed correctness which is presented in [Pau08, KP07].

The correctness of the ABC device is stated against its specification, which is a model as seen by an assembly programmer. With in a slot, this model solely consists of a read buffer and a write buffer. The read buffer holds the last received message and the write buffer is used to store a message to be sent. This model completely abstracts the communication bus, and hence, it abstracts the buffers for the communication with the bus. The transfer of messages at this level is managed by an abstract distributed system, which consists of several ECUs. For more details on this system, we recommend the reader [Pau08, Kna08].

Let C_{ABC} be the state of the specification of the ABC device. We use the function Δ_{ABC} as the step function of specification of the ABC device:

$$\Delta_{\mathrm{ABC}} : \mathit{Difi} \times C_{\mathrm{ABC}} \mapsto C_{\mathrm{ABC}} \times \mathbb{B}^{32}$$

Since the communication bus is abstracted, Δ_{ABC} has neither *eifi* nor *eifo*. We use function $\Omega_{\mathrm{ABC}} : C_{\mathrm{ABC}} \mapsto \mathbb{B}$ which tests whether in a given specification state the interrupt is set up.

We say that a specification state and an implementation state are equivalent, if the send and receive buffers contain the same data. This fact is captured by the predicate $\mathit{Rabc}(h_{\mathrm{ABC}}, c_{\mathrm{ABC}})$.

The local correctness criterion requires that Δ_{ABC} and δ_{ABC} have an equivalent behavior during a slot. The main invariant during a slot states (i) the send and the receive buffers are equivalent, (ii) the data for the processor match, and (iii) the interrupts for the processor match. Thus, the correctness has the following form:

Proposition 6.1

let

$(c_{ABC}', \mathit{difo}_s) := \Delta_{ABC}(\mathit{difi}, c_{ABC})$

$(h_{ABC}', \mathit{eifo}_{ABC}, \mathit{difo}) := \delta_{ABC}(\mathit{difi}, \mathit{eifi}_{ABC}, h_{ABC})$

in

$\neg \mathit{SwapBuf}(h_{ABC}) \ \wedge \mathit{Rabc}(h_{ABC}, c_{ABC}) \longrightarrow$

$\mathit{Rabc}(h_{ABC}', c_{ABC}') \wedge \mathit{difo.data} = \mathit{difo}_s \ \wedge \omega_{ABC}(h_{ABC}) = \Omega_{ABC}(c_{ABC})$

This proposition has to be proved by the designers of the ABC device.

6.2 Instantiation Plan

In oder to construct a verified ECU we have to instantiate the *VDI* and the *VDS* models with the ABC device. The whole instantiation consists of several simple steps:

1. Instantiate the step function of the implementation of the device model, i.e. define δ_{D} from Section 4.1
2. Instantiate the step function of the specification of the device model, i.e. define Δ_{D} from Section 4.2
3. Instantiate the trigger functions *da* and *DevIds* (Section 4.3)
4. Instantiate the simulation relation sim_D (Section 4.3.3)
5. Prove that the definition of *da* and *DevIds* satisfies their intended semantics (Section 4.3)
6. Prove the assumptions which guarantee that the implementation can be simulated by the specification of the device model (Section 4.3.4)

The first four steps instantiate the generic device theory. The last two steps guarantee that the instantiated theory preserves its properties.

6.3 The Device Model

We construct a device model consisting of a single ABC device. Recall that we consider the memory mapped devices, where each device has an associated range of memory addresses. Let DA_{ABC} denote the address range of the bus controller, and let id_{ABC} be an identifier for the ABC device. Thus, the processor-device identifier (Section 4.2) is a set with two elements: $PD = \{P, id_{\mathrm{ABC}}\}$.

Implementation

First we instantiate the generic implementation types H_{D}, *Eifis*, and *Eifos* as follows:

$$
\begin{aligned}
H_{\mathrm{D}} &\triangleq \{id_{\mathrm{ABC}}\} \mapsto H_{\mathrm{ABC}} \\
\mathit{Eifis} &\triangleq \{id_{\mathrm{ABC}}\} \mapsto \mathit{Eifi}_{\mathrm{ABC}} \\
\mathit{Eifos} &\triangleq \{id_{\mathrm{ABC}}\} \mapsto \mathit{Eifo}_{\mathrm{ABC}}
\end{aligned}
$$

We use δ_{ABC} as the base for the implementation of the device model.

Definition 6.1

$$
\begin{array}{l}
\delta_{\mathrm{D}}(\mathit{eifis}, \mathit{difi}, h_{\mathrm{D}}, \mathit{reset}) \triangleq \\
\quad \textbf{let} \\
\qquad (h'_{\mathrm{D}}, \mathit{eifo}, \mathit{difo}) := \delta_{\mathrm{ABC}}(\mathit{eifis}, \mathit{difi}, h_{\mathrm{D}}, \mathit{reset}) \\
\qquad \mathit{eev} := \omega_{\mathrm{ABC}}(h'_{\mathrm{D}}) \circ 0^{18} \\
\quad \textbf{in } (h'_{\mathrm{D}}, \mathit{difo}, \mathit{eifos}, \mathit{eev})
\end{array}
$$

Specification

We instantiate the generic specification types C_{D}, *Eifi*, and *Eifo* as follows:

$$C_{\mathrm{D}} \triangleq \{id_{\mathrm{ABC}}\} \mapsto C_{\mathrm{ABC}}$$
$$Eifi \triangleq \{eifo^{\epsilon}\}$$
$$Eifo \triangleq \{eifi^{\epsilon}\}$$

Since the specification of the ABC device does not consider the external environment, types *Eifi* and *Eifo* contain only idle values $eifo^{\epsilon}$ and $eifi^{\epsilon}$ respectively.

The specification of the device model is based on Δ_{ABC}.

Definition 6.2

$$\begin{array}{l}
\Delta_{\mathrm{D}}(idx, eifi, difi, c_{\mathrm{D}}) \triangleq \\
\quad \textbf{let} \\
\qquad (c_{\mathrm{D}}', difo) \;:=\; \begin{cases} \Delta_{\mathrm{ABC}}(difi^{\epsilon}, c_{\mathrm{D}}) & : \; idx = id_{\mathrm{ABC}} \\ \Delta_{\mathrm{ABC}}(difi, c_{\mathrm{D}}) & : \; idx = P \wedge difi.req \wedge difi.a \in DA_{\mathrm{ABC}} \\ (c_{\mathrm{D}}, difo^{\epsilon}) & : \; else \end{cases} \\
\qquad eev \;:=\; \Omega_{\mathrm{ABC}}(c_{\mathrm{D}}') \circ 0^{18} \\
\quad \textbf{in}\ (c_{\mathrm{D}}', difo, eifo^{\epsilon}, eev)
\end{array}$$

Now we instantiate the trigger functions: *da* signals the end of a device request and *DevIds* which signals which devices make a step due to the *eifi*.

The ABC device answers the processor requests at once, i.e. for a given request it directly produces an output. Thus, the *da* only depends on a given *difi*:

Definition 6.3

$$da(h_{\mathrm{D}}, difi, eifis) \triangleq difi.req$$

The constructed ECU is used in the scope of a distributed framework, and the progress of the device due to the external environment is modelled in this framework. Therefore, at our level, the ECU ignores the input from the bus.

Definition 6.4

$$DevIds(h_{\mathrm{D}}, eifis) \triangleq []$$

Finally, we use as *Rabc* as the simulation relation $sim_{Did_{\mathrm{ABC}}}$.

6.4 Justification of the Device Model

To finish the instantiation of the generic device model, we have to show that the device model consisting of the ABC device fits to the generic device theory developed in Section 4.2. Thus, we have to justify the seven assumptions of the device theory.

There are three assumptions which guarantee a sequential semantics of Δ_D. Two of them (Assumption 4.1 and Assumption 4.2) hold automatically, because there exists only one device in the constructed device model. Definition 6.2 guarantees that if the processor does not access a device, the device won't be changed, and, hence, Assumption 4.3 holds.

Assumption 4.5 requires that *da* can only be activated if there was a request from the processor. It trivially holds, because *da* only depends on the request bit of a given *difi*. This definition also guarantees the liveness property of the device model. Thus, Assumption 4.6 holds.

Assumption 4.7 requires that *DevIds* does not produce sequences with duplicated elements. It holds because our instance of *DevIds* always produces empty sequence.

Finally, we have the assumption which requires that a step of the implementation can be represented by a subsequence of the specification steps (Assumption 4.12). This holds, because there is only one device, and the developers of the ABC device guarantee its correctness.

6.5 Summary: Correctness of the ECU

By instantiating the generic device theory, we created two models of the ECU. One is the ECU model on the gate level, this is the *VDI* model with the ABC device. Let us call this model ECU_I. Another is the ECU model at the assembly level, this is *VDS* model with the specification of the ABC device. We call this model ECU_S. The correctness criterion of the gate-level ECU_I is formulated according to Theorem 5.4:

Theorem 6.2
Let the initial state of the ECU_S model and the ECU_I model be equivalent. Let us assume that up to cycle T there are no buffer swappings, i.e. we consider one slot. Let the executed assembly code satisfy the software conditions. The ECU_I model is correct after T cycles, if after running ECU_S with $\sigma^T = sI_{PD}(T)$, both models are in equivalent states:

$$
\begin{aligned}
&(\forall t < T.\ \neg SwapBuf({ECU_I}^t.h_{PD}.h_D(id_{ABC}))) \wedge \\
&Rconf({ECU_I}^0.h_{PD}.h_P, {ECU_S}^{\sigma^0}.c_{PD}.c_P)\ \wedge M(ECU_I, 0) = {ECU_S}^{\sigma^0}.c_{PD}.c_P.M \\
&Rabc({ECU_I}^0.h_{PD}.h_D(id_{ABC}),\ {ECU_S}^{\sigma^0}.c_{PD}.c_D(id_{ABC})) \\
&\Longrightarrow \\
&ECU_I.JISR^{T-1} \longrightarrow \quad (Rconf({ECU_I}^T.h_{PD}.h_P, {ECU_S}^{\sigma^T}.c_{PD}.c_P) \wedge \\
&\qquad\qquad\qquad\qquad M(ECU_I, T) = {ECU_S}^{\sigma^T}.c_{PD}.c_P.M) \\
&Rabc({ECU_I}^T.h_{PD}.h_D(id_{ABC}),\ {ECU_S}^{\sigma^T}.c_{PD}.c_D(id_{ABC}))
\end{aligned}
$$

Proof. *The proof of this theorem is a single application of Theorem 5.4.* □

As the reader can see the instantiation and the justification are quite simple.

Chapter 7

Summary and Future Work

In this thesis, we presented a formally verified gate-level computer system, which consists of a processor and external devices. To the best of our knowledge, we are the first to present a formally verified computer system with devices at the gate level.

The base of the computer system is a pipelined processor, called VAMP. The VAMP is a 32-bit RISC processor featuring out-of-order execution with five functional units[1], precise interrupts, and address translation. External devices can be easily integrated in the computer system. As an example, we integrated a device which implements an interface for a time-triggered bus. Thus, we built an electronic control unit (ECU) for a distributed automotive system which consist of the VAMP processor and the device. This unit is used as a basic element for building and verifying a distributed automotive system in the Verisoft subproject TP6. These results are formalized and mechanically proved in the interactive theorem prover Isabelle/HOL. Moreover, we synthesised and ran the verified ECU on an FPGA.[2]

As part of this thesis, we developed an environment called IHaVeIt for design and verification of hardware in Isabelle/HOL. The main components of IHaVeIt are two reduction algorithms for Kripke structures. These algorithms combine and extend earlier techniques, which were applicable solely to combinational properties, to temporal properties of Kripke structures. IHaVeIt allows the automatic verification of hardware in Isabelle/HOL and can synthesize the verified hardware. The IHaVeIt environment has also been successfully used in other projects, for example:

- Müller [Mül07] reported on the semi–automated verification of cache systems
- Schmaltz [Sch06b] and Bueker [Bue05] applied IHaVeIt for the verification of an automotive bus controller
- Böhm [Böh07] verified the implementation of a scheduling algorithm of an automotive bus controller

[1] For details on the verification of functional units see Section 3.4.

[2] This is joint work with Andrey Shadrin.

Part	Person years	Theorems	Proof steps
VAMP (no FPU, MU) in Isabelle	1.5	1206	20455
Devices	0.5	52	967
Combining Systems	0.7	118	2714
Total	2.7	1376	24316

Table 7.1: Verification efforts in Isabelle/HOL.

The work in this thesis is partially based on the previous work of the VAMP project [Krö01, BJK+05, DHP05], which is carried out in the interactive theorem prover PVS. In contrast to the previous work, we target the verification of the VAMP in the context of pervasive system verification, i.e. the consideration of the VAMP with external devices. This point of view allowed us to establish clean semantics of the external interrupts, which was not considered in the previous work. Moreover, we decreased the user's involvement in the proving process by usage of the IHaVeIt. This reduction is ca. 30% with respect to the size of the PVS proofs (Section 3.4). Table 7 presents our verification efforts in Isabelle/HOL.

In the rest of this chapter, we discuss the possible direction for further work.

Hardware Optimisations and Extensions

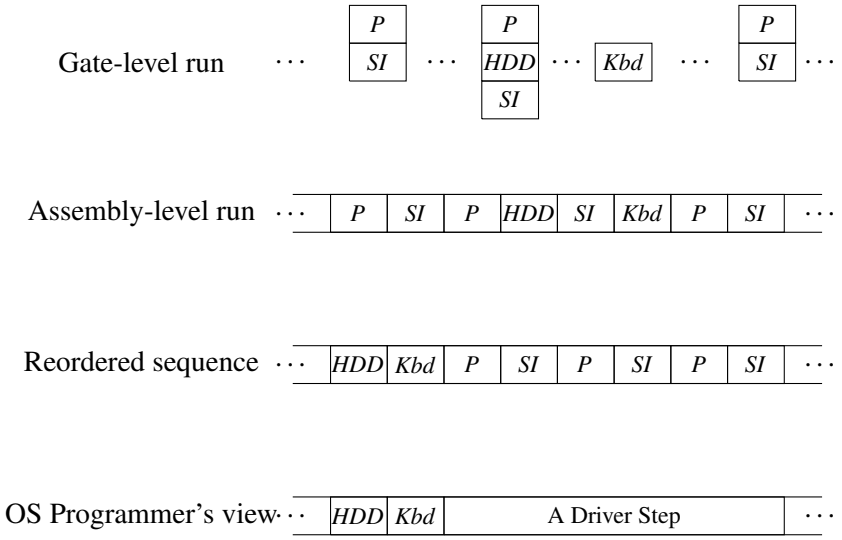
A typical industrial processor with memory management units has table look-aside buffers (TLB). A TLB is used to cache page table entries and, hence, to speed-up the following address translations. Dalinger is currently working on the verification of the memory unit of the VAMP extended by a TLB. This new memory unit also contains faster circuits for computation of effective addresses for memory accesses.

Another possible extension is a multi-level address translation, e.g. one could use the model proposed by Hillebrand [Hil05].

The memory unit of the VAMP can only process one instruction at a time. We can imagine adding a pipeline to this unit, e.g. the address translation is executed in two steps, and, thus, at least two instructions can be processed in the pipeline. One could use the construction proposed by Preiß [Pre05], where the memory unit without address translations is a three-stage pipeline.

Pervasive Verification

Some of the most interesting further work is the usage of the verified computer system in the scope of pervasive verification. Alkassar et al. [AHK+07] instantiated an assembly-level model with a formal model of the serial interface controller UART 16550A. Thus, they constructed an assembler-level programming model for a serial interface, and showed how it can be used for the verification of an assembly-level driver for the controller. They also noticed that the computational sequences provide



where *SI* is the index of the serial interface controller,
Kbd is a keyboard, and *HDD* is a hard disk drive.

Figure 7.1: Reordering and abstracting of computational sequences.

a comfortable way to represent the complete driver execution as an atomic step, i.e. the driver execution as seen by an operating-system programmer. For example, let us consider a subsequence of computational sequence describing a run of an operating system (Figure 7) on a computer system with a processor *P*, a serial interface *SI*, a hard disk drive *HDD*, and a keyboard *Kbd*. At the gate level, this run can last over thousands of cycles, and system components can progress in parallel. Of course, we do not want to prove any properties at this level. The result of this thesis provides us with an assembly-level abstraction of the run. However, if we want to prove and to export properties of a software driver, we do not want to consider the processor- and device steps which are not relevant to the driver execution. Therefore, we can select all processor- and device steps which correspond to a single execution of the driver. Then, we can reorder the processor and the device steps in such a way that the steps belonging to the driver execution are grouped into one continuous subsequence. Now, we can treat this subsequence as an atomic driver step. Therefore, we can provide the OS programmer with the correctness/properties of the whole driver execution.

Alkassar, Starostin, and Schirmer [ASS08] presented how this approach can be applied for a paging mechanism, which is implemented in a page fault handler.

There are a number of interesting opportunities in this area, such as the verification of a driver for a hard disk drive [ASS08, AH08] (or network adapter) and the verification of a file system (a network socket) on top of this driver.

Appendix A

IHaVeIT: Library of Predicate Sets

There is no built-in subtyping mechanism in Isabelle/HOL. Therefore, we restrict infinite types by means of *predicate sets*. A predicate set defines the set of all elements satisfying a given predicate. We defined in Isabelle/HOL a library of the predicate sets for the supported types, which can be described as follows:

$$
\begin{aligned}
sub\_type \quad ::= \quad & boolT \mid bitT \mid bv\_n(\mathbb{N}) \mid nat\_range(\mathbb{N}, \mathbb{N}) \mid \\
& int\_range(\mathbb{Z},\ \mathbb{Z}) \mid arr\_of(\mathbb{N},\ sub\_type) \mid \\
& PAIR(sub\_type,\ sub\_type) \mid RAM(\mathbb{N},\ \mathbb{N}) \mid \\
& ROM([sub\_type_1 \ldots sub\_type_n],\ sub\_type) \mid \\
& G\_MEM([sub\_type_1 \ldots sub\_type_n],\ sub\_type)
\end{aligned}
$$

where

- $boolT$ – $\{True, False\}$
- $bitT$ – $\{1, 0\}$
- $bv\_n(n)$ – Defines a set of all bit vectors of length n. n must be a natural constant.
- $arr\_of(n, pset)$ – Defines a set of all lists of the length n with elements from $pset$. n must be a natural constant.
- $nat\_range(n, m)$ – Defines a set of natural numbers. n and m must be natural constants. If $m < n$, this set is empty.
- $int\_range(n, m)$ – Defines a set of integers. n and m must be integer constants. If $m < n$, this set is empty.
- $PAIR(pset, qset)$ – Defines a set of all pairs where the first element is from $pset$, and the second element is from $qset$.

- $RAM(n, m)$ – Defines a set of functions with domain *bv_nn* and range *bv_nm*. n and m must be natural constants. The terms of this subtype can be updated.
- $ROM([dset_1 \ldots dset_n], rset)$ – Defines a set of functions with range $rset$ and domain $[dset_1 \ldots dset_n]$. The terms of this subtype can not be updated.
- $G\_MEM([dset_1 \ldots dset_n], rset)$ – Defines a set of functions with range $rset$ and domain $[dset_1 \ldots dset_n]$. The terms of this subtype can be updated.

Records are user-defined types, and, therefore, the user has to define the appropriate subtypes for the used records. A subtype for a record is a predicate set those definition captures the subtype information for every record field. Note that the subtyping information for fields of *bool*, *bit*, and *enumerations* types is not required, because it is deduced automatically.

Thus, defining a record which is suitable for the automatic verification and synthesis consists of two steps:

1. define a regular Isabelle/HOL record
2. define a constant representing the required subset of the record type.

We illustrate the defining of such a record by an example. Let us define record *test_record*:

record *test_record* =
field_1 :: *int*
field_2 :: "*bit list*"
field_3 :: *bool*
field_4 :: *some_record*

A subset of record type *test_SubT*, and so the subtype, is to be defined as follows:

constdefs *test_SubT* :: "*test_record set*"
"*test_t* ≡ {*z*.
field_1(*z*) ∈ *int_range*(5, 32) ∧
field_2(*z*) ∈ *bv_n*(32) ∧
field_4(*z*) ∈ *some_SubT*}"

The predicate sets for records can be nested, but they can not be parametrized.

Appendix B

IHaVeIT: Temporal Logic

In this section we employ Isabelle/HOL notation which is introduced in [NPW02].

B.1 LTL formulae specification

In this section we present the syntax for LTL which is supported by IHaVeIt.

The following abstract datatype defines the syntax of LTL.

```
datatype 'a LTL_formula =

    APl "'a ⇒ bool"
  | APl' "'a ⇒ 'a ⇒ bool"
  | Negl "'a LTL_formula"                    ("¬L _" [40] 40)
  | Andl "'a LTL_formula" "'a LTL_formula"   (infixr "∧L" 35)
  | Orl "'a LTL_formula" "'a LTL_formula"    (infixr "∨L" 30)
  | Impl "'a LTL_formula" "'a LTL_formula"   (infixr "⟶L" 30)


  | X "'a LTL_formula"
  | G "'a LTL_formula"
  | F "'a LTL_formula"
  | U "'a LTL_formula" "'a LTL_formula"      (infixr "U" 35)
  | R "'a LTL_formula" "'a LTL_formula"      (infixr "R" 35)


  | Y "'a LTL_formula"
  | Z "'a LTL_formula"
  | H "'a LTL_formula"
  | Once "'a LTL_formula"
  | S "'a LTL_formula" "'a LTL_formula"      (infixr "S" 35)
```

```
  | Trg "'a LTL_formula" "'a LTL_formula"  (infixr "Trg" 35)

  | ExVar "'a" "'a ⇒ 'a ⇒ bool" "'a LTL_formula"
  | AllVar "'a" "'a ⇒ 'a ⇒ bool" "'a LTL_formula"
```

We define the semantics of the LTL formulae via evaluation function `valid1`*.This function evaluates a given LTL formula for a given position in the path.*

```
consts valid1 :: "nat ⇒ (nat ⇒ 'a) ⇒ 'a LTL_formula ⇒ bool"
       ("(_ _ ⊨l _)" [80,80] 80)
primrec
  "s pt ⊨l (APl a)    = (a (pt s))"
  "s pt ⊨l (APl' a)   = a (pt s) (pt (Suc s))"
  "s pt ⊨l (¬L f)   = (¬(s pt ⊨l f))"
  "s pt ⊨l (f ∧L g) = ((s pt ⊨l f) ∧ (s pt ⊨l g))"
  "s pt ⊨l (f ∨L g)  = ((s pt ⊨l f) ∨ (s pt ⊨l g))"
  "s pt ⊨l (f ⟶L g) = ((s pt ⊨l f) ⟶ (s pt ⊨l g))"


  "s pt ⊨l (X f)    = ((Suc s) pt  ⊨l f)"
  "s pt ⊨l (G f)    = (∀ ns. s ≤ ns ⟶  ns pt ⊨l f)"
  "s pt ⊨l (F f)    = (∃ ns. s ≤ ns ∧  ns pt ⊨l f)"
  "s pt ⊨l (f U g)  = (∃ tg. s ≤ tg ∧  (∀ tf < tg. s ≤ tf ⟶
                        tf pt ⊨l f) ∧ (tg pt ⊨l g))"
  "s pt ⊨l (f R g)  = (∀ tg. s ≤ tg ⟶ (∀ tf < tg. s ≤ tf ⟶
                         ¬(tf pt ⊨l f)) ⟶ (tg pt ⊨l g))"



  "s pt ⊨l (Y f)    =  (s ~= 0 ∧ (s - 1) pt  ⊨l f)"
  "s pt ⊨l (Z f)    =  (s = 0 ∨ (s - 1) pt  ⊨l f)"
  "s pt ⊨l (H f)    =  (∀ tf ≤ s. tf pt ⊨l f)"
  "s pt ⊨l (Once f) =  (∃ tf ≤ s. tf pt ⊨l f)"
  "s pt ⊨l (f S g)  =  (∃ tg ≤ s. tg pt ⊨l g ∧
                        (∀ tf ≤ s. tg < tf ⟶  tf pt ⊨l f))"
  "s pt ⊨l (f Trg g)  = (∀ tg ≤ s. tg pt ⊨l g ∨
                        (∃ tf ≤ s. tg < tf ∧  tf pt ⊨l f))"

  "s pt ⊨l AllVar x P f = (∀ x'.  x=x'  ∧
                           P x' (pt s) ⟶ s pt ⊨l f)"
  "s pt ⊨l ExVar x P f  = (∃ x'.  x=x'  ∧
                           P x' (pt s) ∧ s pt ⊨l f)"
```

Function `Paths` *for a given initial state and a state relation computes the set of all paths from the initial state.*

```
constdefs Paths :: "'a ⇒ ('a × 'a) set ⇒ (nat ⇒ 'a) set"
  "Paths s M ≡ {p. s = p 0 ∧ (∀ i. (p i, p (i+1)) ∈ M)}"

constdefs is_Path :: "'a ⇒ ('a × 'a) set ⇒ (nat ⇒ 'a) ⇒ bool"
  "is_Path s M p ≡ s = p 0 ∧ (∀ i. (p i, p (i+1)) ∈ M)"
```

We define functions `LTL_valid` *which check whether a given Kripke structure satisfies a given LTL formula.*

```
constdefs LTL_valid :: "['a ⇒ bool] ⇒ ['a ⇒ 'a] ⇒ 'a set ⇒
                        ' a LTL_formula ⇒ bool"
        ("(_ _ _ ⊨_ _)" [80,80,80,80] 80)
"LTL_valid  Init  Trans  SubT  LTLf ≡
  (let  M = {(s1, s2). Trans s1 = s2  ∧ s1 ∈ SubT ∧ s2 ∈ SubT};
        traces = {trc. (trc 0)  ∈ ({st. Init st} ∩ SubT) ∧
                       is_Path (trc 0) M trc}
   in ∀ trc ∈ traces. 0 trc ⊨_l LTLf)"

constdefs LTL_validG :: "['a ⇒ bool] ⇒ ['a ⇒ 'a ⇒ bool] ⇒
                         'a set ⇒ 'a LTL_formula ⇒ bool"
"LTL_validG  Init  Trans  SubT  LTLf ≡
  (let  M = {(s1, s2). Trans s1 s2  ∧ s1 ∈ SubT ∧ s2 ∈ SubT};
        traces = {trc. (trc 0)  ∈ ({st. Init st} ∩ SubT) ∧
                       is_Path (trc 0) M trc}
   in ∀ trc ∈ traces. 0 trc ⊨_l LTLf)"

consts NX :: "nat ⇒ 'a LTL_formula ⇒  'a LTL_formula"
primrec
"NX 0 f = f"
"NX (Suc n) f = X (NX n f)"
```

B.2 CTL formulae specification

In this section we present the syntax for CTL which is supported by IHaVeIt. This section is based on the case study "Verified Model Checking" presented in [NPW02].

The following abstract datatype defines the syntax of CTL.

```
datatype 'a CTL_formula =
    Atom "'a ⇒ bool"
  | Neg "'a CTL_formula"                          ("¬c _" [40] 40)
  | And "'a CTL_formula" "'a CTL_formula"         (infixr "∧c" 35)
  | Or "'a CTL_formula" "'a CTL_formula"          (infixr "∨c" 35)
  | Imp "'a CTL_formula" "'a CTL_formula"         (infixr "⟶c" 35)
  | ITE bool "'a CTL_formula" "'a CTL_formula"

  | EN "'a CTL_formula"
  | EF "'a CTL_formula"
  | EG "'a CTL_formula"
  | EU "'a CTL_formula" "'a CTL_formula"

  | AX "'a CTL_formula"
  | AF "'a CTL_formula"
  | AG "'a CTL_formula"
  | AU "'a CTL_formula" "'a CTL_formula"
```

Function `Paths` *computes all paths starting from a given initial states.*

```
constdefs Paths :: "'a ⇒ ('a × 'a) set ⇒ (nat ⇒ 'a) set"
  "Paths s M ≡ {p. s = (p 0) ∧ (∀ i. (p i, p(i+1)) ∈ M)}"
```

We define an auxiliary predicate `until` *which checks whether there is a path consisting of states from* `A` *and ending in a state from* `B`.

```
consts until :: "('a × 'a) set ⇒ 'a set ⇒ 'a set ⇒ 'a ⇒
                 'a list ⇒ bool"
primrec
until_Nil:  "until M A B s []    = (s ∈ B)"
until_Cons:  "until M A B s (t#p)  = (s ∈ A ∧ (s, t) ∈ M ∧
                                      until M A B t p)"
```

Function `valid1` *defines the semantics of the CTL.*

```
consts valid1 :: "'a ⇒ ('a × 'a) set ⇒ 'a CTL_formula ⇒ bool"
          ("(_ ⊨₁ _ _)" [80,80,80] 80)
primrec
"s ⊨₁ M (Atom a)  = a s"
"s ⊨₁ M (Neg f)   = (¬(s ⊨₁ M f))"
"s ⊨₁ M (And f g) = ((s ⊨₁ M f) ∧ (s ⊨₁ M g))"
"s ⊨₁ M (Or f g) = ((s ⊨₁ M f) ∨ (s ⊨₁ M g))"
"s ⊨₁ M (Imp f g) = ((s ⊨₁ M f) ⟶ (s ⊨₁ M g))"
"s ⊨₁ M (ITE a f g) = (if a then (s ⊨₁ M f) else (s ⊨₁ M g))"


"s ⊨₁ M (EN f) = (∃ t. (s, t) ∈ M ∧ t ⊨₁ M f)"
"s ⊨₁ M (EF f)   = (∃ t. (s, t) ∈ M* ∧ t ⊨₁ M f)"
"s ⊨₁ M (EG f) =  (∃ p ∈ Paths s M. (∀ i. p i ⊨₁ M f))"
"s ⊨₁ M (EU f g)  = (∃ p. until M {t. t ⊨₁ M f} {t. t ⊨₁ M g} s p)"

"s ⊨₁ M (AX f) = (∀ t. (s, t) ∈ M ⟶ t ⊨₁ M f)"
"s ⊨₁ M (AF f)   = (∀ p ∈ Paths s M. ∃i. p i ⊨₁ M f)"
"s ⊨₁ M (AG f)   = (∀ t. (s, t) ∈ M* ⟶ t ⊨₁ M f)"
"s ⊨₁ M (AU f g) =  (∀ p. until M {t. t ⊨₁ M f} {t. t ⊨₁ M g} s p)"
```

Finally, function `ctl_valid` *checks whether a given Kripke structure satisfies a given CTL formula.*

```
constdefs ctl_valid :: "['a ⇒ bool] ⇒ ['a ⇒ 'a] ⇒ 'a set ⇒
                  'a CTL_formula ⇒ bool"
          ("(_ _ _ ⊨  _)" [80,80,80,80] 80)
"Init M SubTyp ⊨ F ≡
  ∀ s ∈ ({t. Init t} ∩ SubTyp).
    s ⊨₁  {(s1,s2). M s1 = s2 ∧ s1∈ SubTyp ∧ s2 ∈ SubTyp}   F"

constdefs ctl_validG :: "['a ⇒ bool] ⇒ ['a ⇒ 'a ⇒ bool] ⇒
                  'a set ⇒ 'a CTL_formula ⇒ bool"
"ctl_validG Init M SubTyp F ≡
  ∀ s ∈ ({t. Init t} ∩ SubTyp).
    s ⊨₁  {(s1,s2). M s1 s2 ∧ s1∈ SubTyp ∧ s2 ∈ SubTyp}   F"
```

Appendix C

VAMP: Instruction Set

The VAMP instruction set is taken from [Dal06] with minimal modifications.

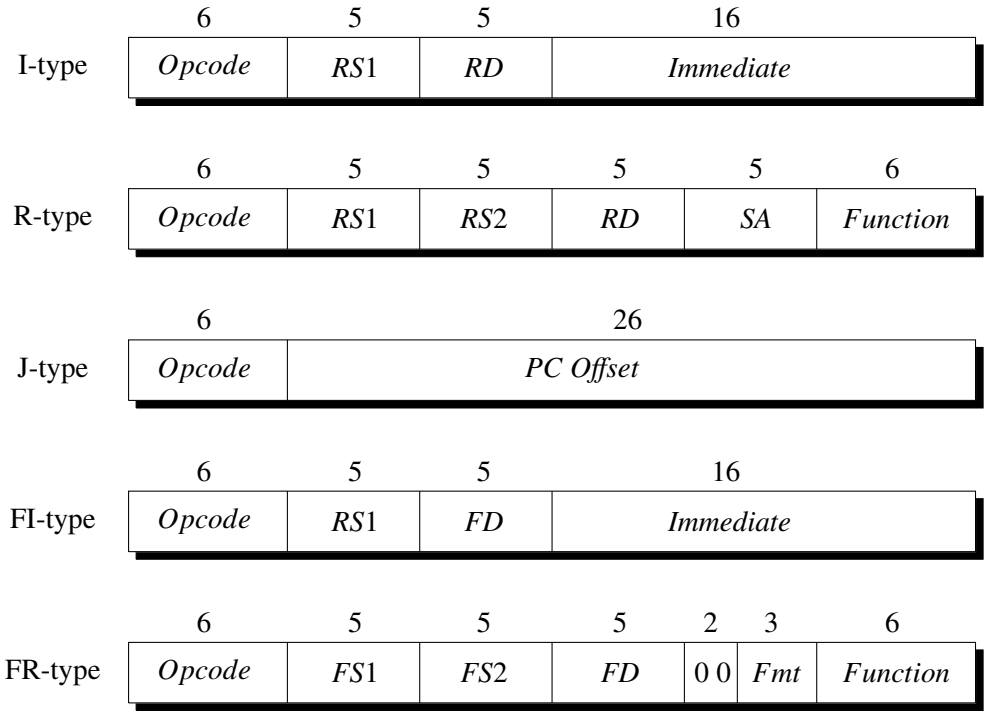


Figure C.1: Instruction formats of the VAMP.

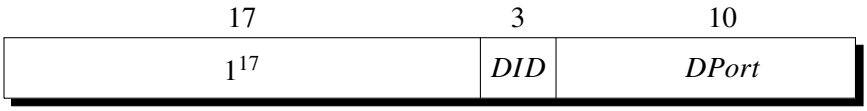


Figure C.2: Format of memory address for device access.

DID specifies the device index and hence we can have up to eight devices in the computer system. *DPort* defines the accesses register (it is also called device port).

$IR[31:26]$	Mnem.	d	Effect
Memory operations, *pa* is a 29-bit effective address or a translated effective address			
100000	lb	1	$RD = sext(M(pa)[29]\ ?\ M(pa)[39:32] : M(pa)[7:0])$
100001	lh	2	$RD = sext(M(pa)[29]\ ?\ M(pa)[47:32] : M(pa)[15:0])$
100011	lw	4	$RD = sext(M(pa)[29]\ ?\ M(pa)[63:32] : M(pa)[31:0])$
100100	lbu	1	$RD = 0^{24}(M(pa)[29]\ ?\ M(pa)[39:32] : M(pa)[7:0])$
100101	lhu	2	$RD = 0^{16}(M(pa)[29]\ ?\ M(pa)[47:32] : M(pa)[15:0])$
101000	sb	1	$M(pa)[7:0] = RD[7:0]$ if $\neg M(pa)[29]$
101001	sh	2	$M(pa)[15:0] = RD[15:0]$ if $\neg M(pa)[29]$
101011	sw	4	$M(pa)[31:0] = RD$ if $\neg M(pa)[29]$
101000	sb	1	$M(pa)[39:32] = RD[7:0]$ if $M(pa)[29]$
101001	sh	2	$M(pa)[47:32] = RD[15:0]$ if $M(pa)[29]$
101011	sw	4	$M(pa)[63:32] = RD$ if $M(pa)[29]$
Arithmetic, logical operation			
001000	addi		$RD = RS1 + imm$
001001	addiu		$RD = RS1 + imm$ (no overflow)
001010	subi		$RD = RS1 - imm$
001011	subiu		$RD = RS1 - imm$ (no overflow)
001100	andi		$RD = RS1 \wedge imm$
001101	ori		$RD = RS1 \vee imm$
001110	xori		$RD = RS1 \oplus imm$
001111	lhgi		$RD = imm \circ 0^{16}$
Test and set operations			
011000	clri		$RD = 0^{32}$
011001	sgri		$RD = 0^{31}(RS1 > imm)$
011010	seqi		$RD = 0^{31}(RS1 = imm)$
011011	sgei		$RD = 0^{31}(RS1 \geq imm)$
011100	slsi		$RD = 0^{31}(RS1 < imm)$
011101	snei		$RD = 0^{31}(RS1 \neq imm)$
011110	slei		$RD = 0^{31}(RS1 \leq imm)$
011111	seti		$RD = 0^{31}1$
Control operation			
000100	beqz		$PCp = PCp + 4 + (RS1 = 0?imm : 0)$
000101	bnez		$PCp = PCp + 4 + (RS1 \neq 0?imm : 0)$
000110	jr		$PCp = RS1$
000111	jalr		$R31 = PCp + 4; PCp = RS1$

Table C.1: I-type instruction layout.

$IR[5:0]$	Mnem.	Effect
Shift operations		
000000	`slli`	$RD = RS1 \ll SA$
000001	`slai`	$RD = RS1 \ll SA$ (arith.)
000010	`srli`	$RD = RS1 \gg SA$
000011	`srai`	$RD = RS1 \gg SA$ (arith.)
000100	`sll`	$RD = RS1 \ll RS2[4:0]$
000101	`sla`	$RD = RS1 \ll RS2[4:0]$ (arith.)
000110	`srl`	$RD = RS1 \gg RS2[4:0]$
000111	`sra`	$RD = RS1 \gg RS2[4:0]$ (arith.)
Data transfer		
010000	`movs2i`	$GPR[RD] = SPR[SA]$
010001	`movi2s`	$SPR[SA] = GPR[RS1]$
Arithmetic and logical operations		
100000	`add`	$RD = RS1 + RS2$
100001	`addu`	$RD = RS1 + RS2$ (no overflow)
100010	`sub`	$RD = RS1 - RS2$
100011	`subu`	$RD = RS1 - RS2$ (no overflow)
100100	`and`	$RD = RS1 \wedge RS2$
100101	`or`	$RD = RS1 \vee RS2$
100110	`xor`	$RD = RS1 \oplus RS2$
100111	`lhg`	$RD = RS2[15:0] \circ 0^{16}$
Test and set operations		
101000	`clr`	$RD = 0^{32}$
101001	`sgr`	$RD = 0^{31}(RS1 > RS2)$
101010	`seq`	$RD = 0^{31}(RS1 = RS2)$
101011	`sge`	$RD = 0^{31}(RS1 \geq RS2)$
101100	`sls`	$RD = 0^{31}(RS1 < RS2)$
101101	`sne`	$RD = 0^{31}(RS1 \neq RS2)$
101110	`sle`	$RD = 0^{31}(RS1 \leq RS2)$
101111	`set`	$RD = 0^{31}1$

Table C.2: R-type instruction layout.

Note that $IR[31:26] = 0^6$ holds for all instructions in this table and that we identify a boolean value of *true* with 1 and $false$ with 0.

$IR[31:26]$	Mnem.	Effect
000010	`j`	$PCp = PCp + 4 + imm$
000011	`jal`	$GPR[31] = PCp + 4$; $PCp = PCp + 4 + imm$
111110	`trap`	$trap = 1$; $EData = imm$
111111	`rfe`	$SR = ESR$; $PCp = EPC$; $DPC = EDPC$

Table C.3: J-type instruction layout.

$IR[31:26]$	Mnem.	d	Effect
Memory operations, *pa* is a 29-bit effective address or a translated effective address			
110001	`load.s`	4	$FD[31:0] = M(pa)[31:0]$
110101	`load.d`	8	$FD[63:0] = M(pa)[63:0]$
111001	`store.s`	4	$M(pa)[31:0] = FD[31:0]$
111101	`store.d`	8	$M(pa)[63:0] = FD[63:0]$
Control operations			
000110	`fbeqz`		$PCp = PCp + 4 + (FCC = 0?imm : 0)$
000111	`fbnez`		$PCp = PCp + 4 + (FCC \neq 0?imm : 0)$

Table C.4: FI-type instruction layout.

$IR[5:0]$	$IR[8:6]$	Mnem.	Effect
Arithmetic and compare operations			
000000		`fadd`	$FD = FS1 + FS2$
000001		`fsub`	$FD = FS1 - FS2$
000010		`fmul`	$FD = FS1 * FS2$
000011		`fdiv`	$FD = FS1 \div FS2$
000100		`fneg`	$FD = -FS1$
000101		`fabs`	$FD = abs(FS1)$
000110		`fsqt`	$FD = sqrt(FS1)$
000111		`frem`	$FD = rem(FS1, FS2)$
$11c[3:0]$		`fc.cond`	$FCC = (FS1\ c\ FS2)$
Data transfer			
001000	000	`fmov.s`	$FD[31:0] = FS1[31:0]$
001000	001	`fmov.d`	$FD[63:0] = FS1[63:0]$
001001		`mf2i`	$GPR[FD] = FPR[FS1][31:0]$
001010		`mi2f`	$FPR[FD][31:0] = GPR[FS2]$
Conversion			
100000	001	`cvt.s.d`	$FD = cvt(FS1, s, d)$
100000	100	`cvt.s.i`	$FD = cvt(FS1, s, i)$
100001	000	`cvt.d.s`	$FD = cvt(FS1, d, s)$
100001	100	`cvt.d.i`	$FD = cvt(FS1, d, i)$
100100	000	`cvt.i.s`	$FD = cvt(FS1, i, s)$
100100	001	`cvt.i.d`	$FD = cvt(FS1, i, d)$

Table C.5: FR-type instruction layout.

Note that $IR[31:26] = 010001$ holds for all instructions in this table.

Condition			Relations				Invalid
Code	Mnemonic		Greater	Less	Equal	Unordered	if
c	True	False	>	<	=	?	unordered
0000	F	T	0	0	0	0	
0001	UN	OR	0	0	0	1	
0010	EQ	NEQ	0	0	1	0	
0011	UEQ	OGL	0	0	1	1	
0100	OLT	UGE	0	1	0	0	No
0101	ULT	OGE	0	1	0	1	
0110	OLE	UGT	0	1	1	0	
0111	ULE	OGT	0	1	1	1	
1000	SF	ST	0	0	0	0	
1001	NGLE	GLE	0	0	0	1	
1010	SEQ	SNE	0	0	1	0	
1011	NGL	GL	0	0	1	1	
1100	LT	NLT	0	1	0	0	Yes
1101	NGE	GE	0	1	0	1	
1110	LE	NLE	0	1	1	0	
1111	NGT	GT	0	1	1	1	

Table C.6: Floating-point relational operators for the `fc` instruction.

Appendix D

Mapping to Lemmata in Isabelle/HOL

In this section we give a mapping from lemmata, corollaries, and theorems in this thesis to the corresponding places in the Isabelle/HOL theories.

Name	Page	Name in Isabelle
Lemma 3.1	55	`step_no_DA_mifo_ignored_spec`
Lemma 3.2	56	`no_DA_mifo_ignored_spec`
Lemma 3.3	68	`da_pend_impl_was_da_and_no_wb`
Lemma 3.4	68	`EevDaR_Correct`
Lemma 3.6	70	`in_order_sI_writeback2`
Lemma 3.7	70	`sI_writeback_free_interval`
Lemma 3.8	71	`all_instr_are_added`
Theorem 3.10	75	`vamp_correct`
Lemma 4.4	86	`DS_DT_ignores_upG`
Lemma 4.8	91	`sIPD_mono`
Lemma 4.9	92	`IsAddedAt_UniqueT`
Lemma 4.10	92	`IsAddedAt_UniqueIdx`
Lemma 4.14	96	`DI_decompose`
Lemma 4.15	96	`DS_decompose`
Lemma 4.16	96	`sIPD_decompose`
Lemma 4.17	96	`sync_eifis_decompose`
Lemma 4.18	96	`sync_difis_decompose`
Lemma 4.19	96	`sync_eifos_decompose`
Theorem 4.20	97	`devs_correct`
Lemma 5.1	102	`VDI_DI_decompose`

Name	Page	Name in Isabelle
Lemma 5.2	103	VDI_VAMP_decompose
Lemma 5.3	105	VDS_PDT_ignores_up
Theorem 5.4	106	part of VDI_vs_VDS_correct
Corollary 5.5	110	VDIP_vs_ISA_correct
Corollary 5.6	110	VDID_vs_DS_correct
Lemma 5.7	111	sIPD_vs_sIwb_rw
Lemma 5.8	111	Seq2I_mono
Lemma 5.10	112	Seq2P_noP_step_rw
Lemma 5.11	112	Seq2P_correct
Lemma 5.12	112	Seq2P_Seq2I_last
Lemma 5.13	112	Seq2I_Seq2P_inv
Theorem 5.14	113	VDSP_vs_ISA_correct
Lemma 5.15	114	VDS_DLX_ignores_devices
Theorem 5.18	115	VDS_DS_decompose
Theorem 5.19	116	VDI_vs_VDS_correct
Lemma 5.24	119	Inst_VDI_vs_DS_SucT
Lemma 5.25	119	Inst_VDI_vs_DS_SucT
Lemma 5.26	120	Inst_VDI_vs_DS_SucT
Lemma 5.27	122	part of VDI_vs_VDS_correct
Lemma 5.28	123	part of VDI_vs_VDS_correct
Lemma 5.29	123	part of VDI_vs_VDS_correct
Lemma 5.30	124	part of VDI_vs_VDS_correct
Lemma 5.33	126	Inst_VDI_vs_ISA_SucT
Lemma 5.34	126	Inst_VDI_vs_ISA_SucT
Lemma 5.35	128	Inst_VDI_vs_ISA_SucT
Lemma 5.36	129	part of VDI_vs_VDS_correct
Lemma 5.37	130	part of VDI_vs_VDS_correct
Theorem 6.2	138	ECU_correct

Bibliography

[ABK08] E. Alkassar, P. Böhm, and S. Knapp. Formal correctness of a gate-level automotive bus controller implementation. In *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES08), to appear*. Springer Science and Business Media, 2008.

[ACHK04] Mark Aagaard, Vlad C. Ciubotariu, Jason T. Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In *FMCAD*, pages 98–112, 2004.

[Ack54] W. Ackermann. Solvable cases of the decision problem. In Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[ADJ04] Mark Aagaard, Nancy A. Day, and Robert B. Jones. Synchronization-at-retirement for pipeline verification. In *FMCAD*, pages 113–127, 2004.

[AH08] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 225–239. Springer, 2008.

[AHK+07] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. Formal device and programming model for a serial interface. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 4–20. CEUR-WS Workshop Proceedings, 2007.

[ALD06] ALDEC – The Design Verification Company. UART nVS. `http://www.aldec.com/products/ipcores/_datasheets/nSys/UART_nVS.pdf`, 2006.

[Ale08] Artem Alekhin. The VAMP memory unit: Hardware design and formal verification effort (draft). Master's thesis, Saarland University, Saarbrücken, Germany, 2008.

[ASS08] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *14th intl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), to appear*, LNCS. Springer, 2008.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[BB05] Bernhard Beckert and Gerd Beuster. Email client verification goals. Interner Technischer Bericht #46, Verisoft Project, 2005.

[BBG+05] Sven Beyer, Peter Bohm, Michael Gerke, Mark Hillebrand, Tom In der Rieden, Steffen Knapp, Dirk Leinenbach, and Wolfgang J. Paul. Towards the formal verification of lower system layers in automotive systems. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 317–326, Washington, DC, USA, 2005. IEEE Computer Society.

[BBM+07] Jörg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg, Tim Blackmore, and Fabio Bruno. Complete formal verification of TriCore2 and other processors. In *DVCon 2007*, February 2007.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV '94*, pages 68–80, London, UK, 1994. Springer.

[Bev94] W. Bevier. Kit: A study in operating system verification. Technical report, Technical Report 28, Computational Logic Inc. (CLI), 1994.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, 2005.

[BHK94] A. Brock, W. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical report, Technical Report 86, Computational Logic Inc. (CLI), 1994.

[BHMY89] William R. Bevier, Warren A. Hunt, Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[BJK+03] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP. In D. Geist and E. Tronci, editors, *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK+05] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Putting it all together – formal verification of the VAMP. *STTT Journal, Special Issue on Recent Advances in Hardware Verification*, 2005.

[BKS03] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System level design and verification using a synchronous language. In *ICCAD*, pages 433–440, 2003.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Saarbrücken, 2008.

[Böh07] Peter Böhm. Formal Verification of a Clock Synchronization Method in a Distributed Automotive System, Master Thesis, Saarland University, Saarbrücken, 2007.

[Brü91] Rolf-Jürgen Brüss. *RISC Die MIPS-R3000-Familie*. Architektur, Systembausteine, Compiler, Tools, Anwendungen. Siemens, 1991.

[Bue05] Matthias Bueker. Automated verification of FlexRay hardware. Interner Technischer Bericht #79, Verisoft Project, 2005.

[BV00] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. In *Computer Aided Verification*, pages 85–98, 2000.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open source tool for symbolic model checking. In *CAV '02*, pages 359–364. Springer, 2002.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[Coh00] Ben Cohen. Component design by example: A step-by-step process using VHDL with UART as vehicle. VhdlCohen, November 2000.

[Con93] OSEK Consortium. OSEK VDX portal. `http://portal.osek-vdx.org/`, 1993.

[Con06] FlexRay Consortium. FlexRay – the communication system for advanced automotive control applications. `http://www.flexray.com/`, 2006.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Saarbrücken, 2006.

[DdM06] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[DDS08] Matthias Daum, Jan Dörrenbächer, and Mareike Schmidt. Modelling user programs on top of a microkernel. In *TPHOLs*, 2008. submitted.

[DDSW08] Matthias Daum, Jan Dörrenbächer, Mareike Schmidt, and Burkhart Wolff. A verification approach for system-level concurrent programs. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, and Experiments*, volume 5295/2008 of *LNCS*, pages 161–176. Springer, 2008.

[DHP05] I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *CHARME 2005*, LNCS, pages 301–316. Springer, 2005.

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[EMS07] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up SAT. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.

[GGA05] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Verification of embedded memory systems using efficient memory modeling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1096–1101, Washington, DC, USA, 2005. IEEE Computer Society.

[Hal95] Tom R. Halfhill. The truth behind the Pentium bug. `http://www.byte.com/art/9503/sec13/art1.htm`, 1995.

[HGS03] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam K. Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, 2003.

[HIdRP05] M. Hillebrand, T. In der Rieden, and W.J. Paul. Dealing with I/O devices in the context of pervasive system verification. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings*, pages 309–316. IEEE, 2005.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Saarbrücken, 2005.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[IdRT08] T. In der Rieden and A. Tsyban. CVM – a verified framework for microkernel programmers. In *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B.V., 2008.

[IEE85] ANSI/IEEE standard 754-1985. IEEE standard for binary floating-point arithmetic. Technical report, 1985.

[Jac02] Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Computer Aided Verification (CAV 02)*, volume 2404 of *LNCS*, pages 309–323. Springer, 2002.

[JED+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 396–410, London, UK, 2001. Springer.

[KMM00] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Kna08] Steffen Knapp. *Distributed Automotive Systems (draft)*. PhD thesis, Saarland University, Saarbrücken, 2008.

[KP07] Steffen Knapp and Wolfgang Paul. Pervasive verification of distributed real-time systems. In M. Broy, J. Grünbauer, and T. Hoare, editors, *Software System Reliability and Security*, volume 9 of *IOS Press, NATO Security Through Science Series.*, 2007.

[Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Saarbrücken, 2001.

[Lef04] Jörg Lefèvre. Ausgebremst: Kein Auto-Rückruf trotz Defekt (german). `http://www.daserste.de/plusminus/beitrag_archiv.asp?aid=254`, 2004.

[LNR05] Ranko Lazic, Thomas Christopher Newcomb, and Bill Roscoe. Polymorphic systems with arrays, 2-counter machines and multiset rewriting. *Electr. Notes Theor. Comput. Sci.*, 138(3):61–86, 2005.

[LNRS07] B. Langenstein, A. Nonnengart, G. Rock, and W. Stephan. A history-based verification of distributed applications. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 70–84. CEUR-WS Workshop Proceedings, 2007.

[LP08] D. Leinenbach and E. Petrova. Pervasive compiler verification – From verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[LPP05] Dirk Leinenbach, Wolfgang J. Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *SEFM*, pages 2–12, 2005.

[LSB02] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD '02*, pages 142–159, London, UK, 2002. Springer.

[LT93] Nancy G. Leveson and Clark Savage Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[McM99] K. L. McMillan. Getting started with SMV. Technical report, Berkeley Labs, 1999.

[MLH07] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 385–390. Springer, 2007.

[Moo94] J Moore. A mechanically verified language implementation. Technical report, Technical Report 30, Computational Logic Inc. (CLI), 1994.

[Moo02] J Strother 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 *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.

[MP00] Silvia Müller and Wolfgang Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[MS06] Panagiotis Manolios and Sudarshan K. Srinivasan. A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures. *Journal of Automated Reasoning*, 37(1–2):93–116, 2006.

[MSV06] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. Automatic memory reductions for RTL model verification. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 786–793, New York, NY, USA, 2006. ACM Press.

[Mül98] Oliver Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Techn. Univ. Munich, 1998.

[Mül07] Christian Müller. Verification of a simple cache system,, Master Thesis, Saarland University, Saarbrücken, 2007.

[Neu89] Peter Neumann. Mariner I – no holds BARred. `http://catless.ncl.ac.uk/Risks/8.75.html#subj1`, 1989.

[NK00] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, pages 435–449, 2000.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Nus97] Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Softw.*, 14(3):15–16, 1997.

[OS06] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In *IJCAR*, pages 298–302, 2006.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In *CADE'92*, pages 748–752, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, New York, NY, USA, 1994.

[Pau08] Wolfgang Paul. Computer architecture 2. lecture notes. `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur2/ws0809/`, 2008.

[Pfe07] Frank Pfenning, editor. *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*. Springer, 2007.

[PMV98] V. Paruthi, N. Mansouri, and R. Vemuri. Automatic data path abstraction for verification of large scale designs. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 192, Washington, DC, USA, 1998. IEEE Computer Society.

[Pre05] Jochen Preiß. *Complexity and Correctness of a Super-Pipelined Processor*. PhD thesis, Saarland University, Saarbrücken, 2005.

[PRSS02] Amir Pnueli, Yoav Rodeh, Ofer Strichmann, and Michael Siegel. The small model property: how small can it be? *Inf. Comput.*, 178(1):279–293, 2002.

[Rob02] Nina Amla Robert. Autoabs: Syntax-directed program abstraction. citeseer.ist.psu.edu/608703.html, 2002.

[RPS01] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-Chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[RSS95] S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. An integration of model checking with automated proof checking. In *CAV '05*, pages 84–97. Springer, 1995.

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In *FTRTFT'02*, volume 2469 of *LNCS*, pages 83–105, Oldenburg, Germany, September 2002. Springer-Verlag.

[RV01] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 376–380, London, UK, 2001. Springer.

[Sch06a] J. Schmaltz. A formal model of lower system layer. In *FMCAD'06*, pages 191–192. IEEE/ACM Press, 2006.

[Sch06b] J. Schmaltz. A formalization of clock domain crossing and semi-automatic verification of low level clock synchronization hardware. Technical report, Saarland University, 2006.

[Sch07] Wolfram Schulte. Experiments in verifying low level concurrent C code. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, page 299, Washington, DC, USA, 2007. IEEE Computer Society.

[SH98] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. 10th International Computer Aided Verification Conference*, pages 135–146, 1998.

[SJ02] Jun Sawada and Warren A. Hunt Jr. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, 2002.

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide. Technical report, SRI International, 2001.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.*, 37(5):562–573, 1988.

[Str02] O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical report, Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.

[TA08] Sergey Tverdyshev and Eyad Alkassar. Efficient bit-level model reductions for automated hardware verification. In *15th International Symposium on Temporal Representation and Reasoning: TIME 2008, to appear*. IEEE, 2008.

[The03] The Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`, 2003.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, pages 25–33, 1967.

[TS07] Sergey Tverdyshev and Sebastian Skalberg. Private email conversations, 2003-2007.

[Tve05a] Sergey Tverdyshev. Combination of Isabelle/HOL with automatic tools. In Bernhard Gramlich, editor, *FroCoS 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2005.

[Tve05b] Tverdyshev, Sergey. The IHaVeIt project. `http://www-wjp.cs.uni-sb.de/ihaveit/`, 2005.

[TYRM04] Daijue Tang, Yinlei Yu, Darsh Ranjan, and Sharad Malik. Analysis of search based algorithms for satisfiability of quantified boolean formulas arising from circuit state space diameter problems. In *SAT*, pages 292–305, 2004.

[VB99] Miroslav N. Velev and Randal E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *CHARME*, pages 37–53, 1999.

[VB05] Miroslav N. Velev and Randal E. Bryant. TLSim and EVC: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories. *Int. J. Embedded Systems*, 1(1/2):134–149, 2005.

[Vel05] Miroslav N. Velev. Automatic formal verification of liveness for pipelined processors with multicycle functional units. In *CHARME*, pages 97–113, 2005.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated deduction, CADE-18*, volume 2392, pages 275–279, Kopenhagen, Denmark, 2002. Springer.

[You94] W. Young. A mechanically verified code generator. Technical report, Technical Report 93, Computational Logic Inc. (CLI), 1994.

Index