Timing Model Derivation

Pipeline Analyzer Generation
from
Hardware Description Languages

Dissertation

zur Erlangung des Grades des

Doktors der Ingenieurwissenschaften

der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Diplom-Informatiker
Markus Pister

Saarbrücken
Mai 2012

Kolloquium

Tag des Kolloquiums	Mittwoch, 19. September 2012
Dekan	Prof. Dr. Mark Groves
Prüfungsausschuss	
Vorsitzender	Prof. Dr. Sebastian Hack
Berichterstatter	Prof. Dr. Dr. h.c. Reinhard Wilhelm
	Prof. Dr.-Ing. Wolfgang Kunz
Akad. Mitarbeiter	Dr.-Ing. Daniel Grund

Impressum

Herstellung und Verlag: Pirrot Verlag, Saarbrücken

ISBN: 978-3-937436-40-1

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.d-nb.de` abrufbar.

Abstract

Safety-critical systems are forced to finish their execution within strict deadlines so that worst-case execution time (WCET) guarantees are a crucial part of their verification. Timing models of the analyzed hardware form the basis for static analysis-based approaches like the aiT WCET analyzer. Currently, timing models are hand-crafted based on frequently incorrect documentation causing the process to be error-prone and time-consuming.

This thesis bridges the gap between automatic hardware synthesis and WCET analysis development by introducing a process for the derivation of timing models from VHDL specifications. We propose a set of transformations and abstractions to reduce the hardware design's complexity enabling the generation of efficient and provably correct WCET analyzers. They employ an abstract interpretation-based simulation of program executions based on a defined abstract simulation semantics. We have defined workflow patterns showing how to gradually apply the derivation process to VHDL models, thereby removing timing-irrelevant constructs. Interval property checking is used to validate the transformations.

A further contribution of this thesis is the implementation of a tool set that realizes the introduced derivation process and shows its applicability to non-trivial industrial designs in experimental evaluations. Influences on design choices to the quality of the derived timing model are presented building an informal predictability notion for VHDL.

Zusammenfassung

Sicherheits-kritische Systeme unterliegen oft der Einhaltung strikter Laufzeitschranken, weshalb zur Verifikation sichere Obergrenzen der Laufzeit im schlimmsten Fall (WCET) bestimmt werden. Zeitmodelle der analysierten Hardware sind hierbei die Grundlage für auf statischen Analysen basierende Verfahren. Aktuell werden solche Modelle händisch aus Handbüchern extrahiert, ein sehr zeitaufwändiger und fehleranfälliger Prozess.
Diese Arbeit schlägt eine Brücke zwischen automatischer Hardware-Synthese und der Entwicklung von WCET-Analysen durch die Einführung eines Ableitungsprozesses von Zeitmodellen aus VHDL-Spezifikationen. Transformationen und Abstraktionen werden zur Komplexitätsreduktion eingesetzt, um die Erzeugung von effizienten und beweisbar korrekten Analysatoren zu ermöglichen. Selbige bedienen sich abstrakter Interpretation von Programmausführungen basierend auf einer Simulations-Semantik. Definierte Arbeitsabläufe zeigen, wie man die Ableitung schrittweise auf VHDL-Modellen umsetzt und dadurch für das Zeitverhalten irrelevante Teile des Modells entfernt. Interval Property Checking gewährleistet hierbei, dass die Transformationen semantik-erhaltend sind.
Eine Tool-Implementierung realisiert den vorgestellen Ableitungsprozess und unterstreicht seine Anwendbarkeit auf komplexe industrielle Designs durch experimentelle Untersuchungen. Außerdem werden VHDL-Designentscheidungen hinsicht ihres Einflusses auf die Qualität des abgeleiteten Zeitmodells betrachtet.

Acknowledgments

I want to express my gratitude to many people that have influenced this work in a variety of ways. First, many thanks go to *Prof. Reinhard Wilhelm* not only for allowing me to scientifically address this challenging topic. But also because I have learned a lot due to his ability to steer people into the right direction while still fostering their personal development by very free working conditions.

I am also very grateful to *Prof. Wolfgang Kunz* for his willingness to judge my thesis and his motivating interest in my work, in general. *Prof. Sebastian Hack* and *Dr. Daniel Grund* are the remaining members of my thesis jury, so I thank both for their commitment, as well.

Reinhold Heckmann, *Daniel Kästner*, *Philipp Lucas*, *Stephan Thesing* and *Markus Wedler* all have proof-red the thesis at hand and provided many excellent comments. I am indebted to them for this.

Thanks go to various (both current and former) colleagues at *AbsInt GmbH* and the *Compiler Design Group* at Saarland University. I enjoyed many hilarious (coffee) breaks with them over the last couple of years.

Although he has already been covered by the credits above, I want to point out my special thanks to my long-lasting colleague *Marc Schlickling*. We are cooperating since the early phases of our studies and have jointly developed large parts of the VHDL Derivation Tool Set. His ability to quickly find conceptual corner cases was a great help to me and I appreciate him a lot as my office roommate and friend.

During my research, I was (at least partly) funded by different European research projects *AVACS*, *ES_PASS* and *Verisoft XT*, so I want to thank for these investments in my work.

Last but not least I want to thank my parents, *Christel* and *Gerhard*, and my love *Katharina* for their support in all phases of my life.

Contents

1

Introduction

"There is no reason for any individual to have a computer in his home."

(Ken Olson)

1.1 Embedded Systems

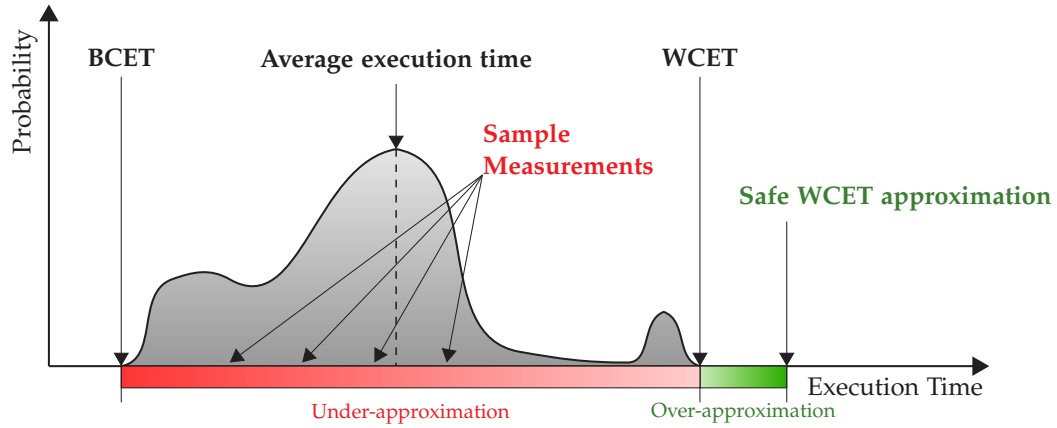
Since the industrialization, our daily life is more and more influenced by technological innovations. This trend emerges, e.g., the evolution of mobile phones from their original purpose — being able to phone a person more or less independently of the own location — towards small but integrated smartphones allowing their users to not only do phone calls but also to organize their life and thereby integrating business with spare time. But embedded systems support human beings even in less visible areas: namely embedded into larger products like cars, air planes, trains, medical devices and more.

Among those applications, there are the *safety-critical* applications/systems, as e.g., a flight controller in fly-by-wire steered air planes which surely is one of the most prominent and critical examples. Failures of such systems are simply not acceptable and the probability of their occurrence must be at least minimized. Otherwise, their consequences would create high costs or even endanger human life. Therefore, utmost carefulness and state-of-the-art techniques for verifying software safety requirements have to be applied in order to assure an application's proper mode of operation.

1.2 Timing Analysis of Embedded Systems

Assuring correctness is not limited to the validation of the program logic of the application. Beyond that, the absence of runtime errors (division by zero, array index over-/underflow, ...) has to be proven [KWN+10]. In addition, safety-critical systems are often forced to finish their execution within strict deadlines dictated by the surrounding physical environment. Not to fulfill such deadlines might lead to system errors ending up in hazards that actually compromise the functional correctness of a system. This dependency is also reflected in all current safety standards (like DO-178B [DO92], ISO 26262 [ISO11], IEC 61508 [IEC10], ...) whose goals are to require the system developer to identify both functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Depending on the criticality level, a sophisticated examination of the timing behavior has to be contributed to show the functional correctness of a safety-critical system, i.e., it has to be proven that the system meets its deadlines even in the worst-case scenario [SPH+05]. While the standards do not enforce

Figure 1.1 – Execution time distribution



specific testing and verification methods, the importance of static verification is emphasized.

Code level timing analysis addresses the non-preempted execution of a single process, a task or an interrupt service routine. Its results are then used by the system level timing analysis which checks a system of functions or tasks for their schedulability according to a specified scheduling algorithm. This thesis focuses on the code level timing analysis as described in Chapter 4.

1.2.1 The Timing Problem

> *Why is the determination of the runtime of a program a difficult and complex problem?*

One might come up with this question. But when starting to think about how to exactly determine a program's execution time, it quickly becomes clear that runtime is not a single and precisely computable value. Figure 1.1 tries to exemplarily illustrate that the runtime of a program or task varies among different executions between a so-called *best-* and *worst-case execution time* which are typically abbreviated with BCET and WCET.

Possible reasons for the execution time's variability are

- the input program (size, number of memory accesses, ...),

- the initial system state (cache contents, hardware configuration, ...),
- interferences from the environment (preemptions, interrupts, ...).

In general, the concrete WCET cannot be determined for all programs because this would decide the halting problem (cf. Chapter 4). But a suitably restricted subset of programs (no endless loops, no dynamic memory allocation, ...) would at least theoretically allow this computation. However practically, this task is just too demanding from a computational complexity point of view because of the above mentioned uncertainties in the starting state of the system. Additionally, a "brute-force" simulation of all execution paths in the program is not feasible in most cases, as well, since the number of paths grows exponentially with the size of the program [WEE+08]. Any test case selection for a program potentially under-approximates its WCET as long as the corresponding worst-case input (the one triggering the worst-case execution time) is not known. Unfortunately, this worst-case input is unknown for non-trivial software and its determination would require an exhaustive exploration of all program paths. Therefore, it can only be approximated in a safely manner by computing an upper bound on the execution time which is guaranteed to be greater than or equal to all possible execution times independently from the sources of variability. Moreover, the computed bound needs to be precise for industrial applications because large overestimations do not enable users to prove their timing constraints to be met.

A traditional approach for this timing problem has been to partition the application under analysis into code snippets for which the determination of the worst-case input seems to be possible. The execution time of each snippet based on these inputs is then measured and the results are combined to find the global worst-case path together with its runtime. Based on the above described variability of the runtime, this is an error-prone and expensive approach and does not satisfy modern safety requirements. The goal must be the determination of safe and precise upper bounds on the concrete WCET.

1.2.2 WCET Analysis by Abstract Interpretation

Most interesting program properties like the WCET are undecidable. Abstract interpretation [CC77, CC92a, CC92b] is a semantics-based methodology for static program analyses where the concrete semantics is mapped to a simpler – abstract – model. Static analysis then determines program properties with

respect to that model. Although mapping concrete to abstract semantics is at the expense of analysis precision, i.e., it leads to over-approximations of the concrete WCET, the computation may be faster. Another advantage is the provable soundness of abstract interpretation-based analyzers. Sound means that its results hold for any program execution and can therefore be used as a safety-guarantee. A WCET bound computed by a sound analyzer will never under-approximate the concrete WCET of the analyzed program. This combination of analysis efficiency and soundness matches the above requirements of timing analyses of safety-critical embedded systems.

Over the last years, a standard architecture for timing analysis tools has emerged [TFW00, FHL+01, Erm03] and consists of the following phases:

- control-flow reconstruction,
- micro-architectural analysis and
- path analysis.

The **control-flow reconstruction phase** operates on the fully linked binary executable and generates a combined call and control-flow graph for its input [The03]. All subsequent analyses are based on this intermediate representation.

The computationally more intensive **micro-architectural analysis phase** is realized by *abstract interpretations* and has the following three constituents:

1. *Loop/Value analysis* attempts to compute information about data accesses and control flow. In particular it tries to identify infeasible paths, i.e., syntactically possible paths that will never be taken because of contradictory conditions. The underlying abstract model is manually derived from the concrete instruction-set semantics.

2. *Cache-behavior prediction* determines a safe and concise approximation of the contents of caches in order to classify memory accesses as definite cache hits or misses.

3. *Pipeline-behavior prediction* analyzes how instructions pass through the processor pipeline taking cache-hit or miss information into account. There, basic block timings are determined using an abstract processor model (*timing model*) that defines a cycle-level abstract semantics for each instruction's execution yielding in a certain set of final system states. After the analysis of one instruction has been finished, these

states are used as start states in the analysis of the successor instruction(s). Here, the timing model introduces nondeterminism that leads to multiple possible execution paths in the analyzed program. The pipeline analysis needs to examine all of these paths.

The last phase of the timing analysis architecture, the **path analysis**, combines the timing information computed by its predecessor phase to determine the global worst-case execution time for the input program and the triggering path, the so-called worst-case execution path.

The most challenging part of this architecture is the creation of the underlying timing model for the pipeline-behavior prediction because it needs to precisely model the behavior of the processor with its employed memory hierarchy like memory controller, bus system, etc. Depending on the architecture under analysis, the modeled level of detail varies, i.e., for rather simple processors like an ARM7 [ARM00], the analysis "only" needs to count the instructions while applying specific execution latencies. But the unit-time (executing an instruction always takes exactly one time unit) or constant-time abstraction used for simpler architectures is rendered obsolete by the advent of modern processors. With features like branch prediction, out-of-order execution or speculation, the state space of input data and initial states, in general, is too large to exhaustively explore all possible executions and so determine the exact worst-case execution time. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions inevitably lose information, and yet must guarantee upper bounds for the worst-case execution time.

The aiT WCET analyzer developed by AbsInt GmbH and Saarland University is a prominent timing analysis tool that implements the above described architecture. It has been used in international industrial and research projects, e.g., Verisoft, Verisoft XT [Ver], PREDATOR [PRE], INTERESTED [INT], ..., and has been used in the certification of safety-critical industrial systems, e.g. [SPH+05]. Therefore, the implementations that are contributed by this thesis, are realized on top of the AbsInt tool chain for WCET analysis, which is detailed in Section 4.3.

1.3 Deriving Timing Models

Currently, the timing models described above are *hand-crafted* [SP10] by human experts based on

- processor/system documentation and
- reverse engineering by runtime measurements.

Although the hardware manufacturers are usually cooperative and provide internal and partly confidential documentation, there is the drawback that those documents usually contain errors or leave out some important details. Traces of specific runtime measurements on partly hand-written assembler code should fill this gap. Depending on the hardware architecture, those traces might only show end-to-end timings or timestamps of active bus transaction signals (cf. Section 8.2.2). Sometimes, it is also possible to monitor processor core internals by so-called performance monitoring features within the hardware (cf. Section 8.2.1).

This basis for the development of timing models renders the model engineering as well as the implementation of the corresponding pipeline analyzer to be a time-consuming, complex and error-prone process. Creating such a timing model from scratch for a modern and complex processor like the Freescale PowerPC 7448 [Fre05a] might consume 3–4 man-months even for rather experienced people.

Nowadays, hardware circuits are automatically synthesized from formal hardware specifications like VHDL or Verilog (cf. Section 5.3). Besides a formalization of the functional details, such specifications implicitly contain an execution model that also reflects the timing behavior of the whole system. This enables the derivation of timing models based on their formal hardware specification to simplify the above described error-prone development process. Moreover, this bridges the gap between hardware circuit synthesis and WCET analysis development and constitutes one of the major contributions of this thesis.

1.4 Contributions

In the context of timing analysis for safety-critical embedded control software and the development of the required timing models, the contributions of this thesis are outlined below. Additionally, a differentiation between the present thesis and the work of Schlickling [Sch13] is given where appropriate.

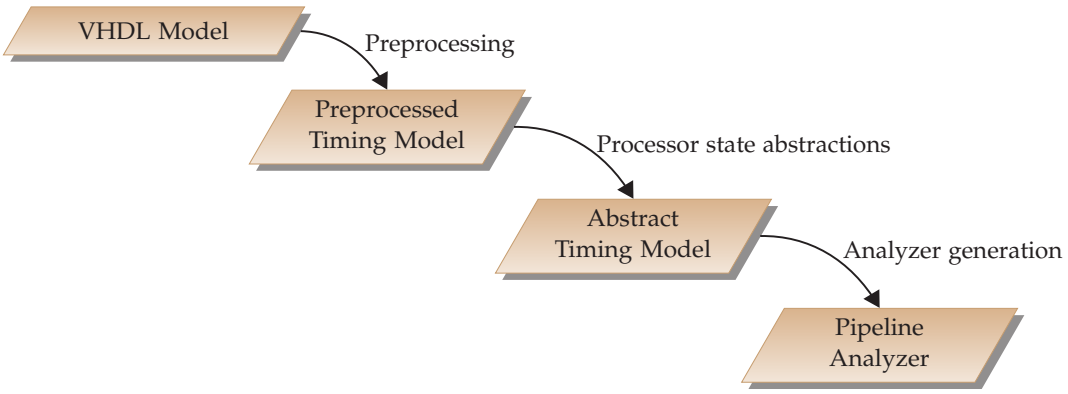
1.4.1 Timing Model Derivation

Static analysis and model transformations are employed to extract the timing information of hardware circuits from its formal specification. Where Schlickling [Sch13] introduces abstract interpretation-based static analysis of formal hardware models, this thesis focuses on the derivation process as a whole. Starting from the hardware model, transformations and abstractions based on the results of static analyzers extract the timing-relevant information, the timing model. In the end, an aiT-compatible pipeline analyzer can be generated from such a model.

Semantic Level Reduction A prerequisite for the application of both static program analysis and transformations is a common intermediate representation of the hardware model. Because the chosen analysis approach, abstract interpretation, has its origins in program analysis, the hardware model needs to be represented as a sequential program. Although hardware descriptions are concurrent models, i.e., their processes are designed to run in parallel, such a conversion is possible because their two-level semantics allows to choose an arbitrary execution order of the processes (cf. Section 5.2.3). Thus, this thesis defines translation rules to convert VHDL language constructs into control-flow entities combined with a framework of generated routines, so that the resulting control-flow representation forms a simulation environment for the original hardware design. On top of that, abstract interpretation-based static analyzers [Sch13] are enabled to examine the model and result communication is possible via a shared intermediate format called *CRL*. This translation from VHDL into a sequential execution model is called *semantic level reduction* (cf. Section 6.3.2) and is generic enough to be applied to related hardware description languages like Verilog, as well.

Derivation Process Definition As these hardware specification languages have been partly designed for integrated simulation of the specified circuits [Ash08], the question might come up here whether it would not be possible to just use these simulation capabilities to determine execution costs of critical tasks. The answer is that VHDL models of real-world processors are usually big and complex making WCET determination a difficult task [SP10]. In the presence of the uncertainties described above like nondeterministic system starting states, a static timing analysis based on the unchanged formal hardware specification would be infeasible in terms of space and analysis

Figure 1.2 – Sketched Timing Model Derivation Process



time consumption. Furthermore, the VHDL code cannot be used "as is" for a static timing analysis because such a simulation cannot cope with "unknown" data, e.g., unknown cache contents.

Therefore, a distinct process for extracting timing-relevant information is necessary which is one of the main contributions of this thesis. Figure 1.2 shows a structural picture of this process.

In a first step, the size of the model is reduced by pruning out all parts that do not contribute to the timing behavior at all. For example, there is no need for information about each step within a multiplier unit. Instead, it suffices to know the number of clock cycles each stage of the multiplier pipeline is occupied by a specific instruction. The pruned model still contains a lot of detailed information about the processor state. But for practical reasons it is impossible to represent all state information in full detail. For example, if the contents of all memory cells or registers should be modeled in all details, the resulting space requirements would be prohibitive for non-trivial architectures. Fortunately, the exact knowledge about such details often is not important as far as timing is concerned: an addition always takes the same amount of time, no matter what the arguments are. In other cases, the timing does depend on such information, but a loss of precision is acceptable in order to make the analysis more efficient, or even possible at all. One example for this situation are multiplications on some architectures which are faster if one argument has leading zero bits. By not keeping track of the arguments exactly, an entire range of execution times has to be assumed for multiplication. The loss in precision is acceptable in this case as the difference

usually is below ten processor cycles and multiplications are in general rare. Therefore, abstractions from the concrete model are introduced, i.e., some details of the processor state are left out or will be approximated. Using the methodology of abstract interpretation, one can trade precision of the analysis against efficiency by choosing different abstractions and concretion relations between the original and abstract model. Finally, an aiT-compatible pipeline-behavior analysis is generated from the timing model in the last step that concludes the derivation process.

Derivation of Workflow Patterns The derivation process described above represents a general methodology. But the particular abstractions and transformations depend on the concrete hardware architecture whose timing behavior has to be modeled so that the invention of these abstractions remains an intellectual challenge. However, typical working patterns came up during the implementation/experiments and are described in Section 6.5.

The model preprocessing phase is realized by assigning fixed values to signals/variables so that a subsequent constant propagation effectively renders parts of the specification unused – *timing dead*. Their transitive closure can then be removed automatically. Inventing assumptions about the model is an iterative procedure and coupled with feedback from an interactive exploration and understanding of the hardware design that can be reached using the level-based slicing tool developed by Schlickling [Sch13].

In principle, a processor's timing behavior is dominated by the (timing-)effect of the instruction flow through the processor pipeline and latencies for memory accesses. Therefore, the resulting timing model needs to represent this flow. Required combinations of program slices and additional model assumptions can be used to extract this information. They are described in Section 6.5.1.

After model preprocessing, abstractions may be applied to further reduce the resulting timing analysis' complexity. Three different kinds of abstractions are proposed:

- *Memory abstraction*
- *Domain abstraction* and
- *Process replacement*

The memory abstraction aims at removing all data paths from the hardware model. Instead, queries to the value analysis results (cf. Section 1.2.2) are inserted appropriately. Domain abstractions perform source-to-source type transformations. A prominent example is the so-called address abstraction where concrete addresses are replaced by abstract values standing for certain sets of concrete values. Process replacements enable the integration of custom simulation routines into the timing model. They substitute VHDL processes by simplified code snippets. Typically, these snippets specify the timing effect of the replaced process and remove purely functional code. For example, the functionality of a simple arithmetic unit can be replaced by a small component with a timer. It is started for an instruction newly entering the unit and simulates their execution time because this is the essential information within the timing analysis.

1.4.2 Simulation Semantics

A pipeline-behavior analysis is an abstract (and thereby computable) simulation of a program's execution. The generated analyzer is based on a corresponding abstract simulation semantics that is constructed throughout Chapter 7: An operational semantics for the simulation of non-abstracted VHDL models is formalized and followed by an abstract variant so that the pipeline analyzer's abstract simulation safely approximates any concrete simulation. Employed abstractions can render the model nondeterministic, i.e., the simulation process might compute multiple successors for a given input system state leading to multiple possible execution paths partially with different costs in terms of execution time. The defined abstract simulation semantics is able to cope with such uncertainties and simulates all potential execution paths.

1.4.3 Timing Model Validation

The derived timing model per construction is a correct representation of the underlying hardware's timing behavior and the defined abstract simulation semantics is a sound approximation to any concrete simulation. Concerning the correctness of the resulting pipeline-behavior analysis, it remains to show that the employed model abstractions and transformations do not introduce unsafe[1] changes to the timing behavior. Interval property checking techniques

[1] under-estimations

(IPC) [Bor09] from the area of formal functional hardware verification fit that purpose because they are able to show the "timing-semantic" preserving translation from the input design to the timing model. Thus, this thesis contributes approaches for the validation of derived timing models based on IPC.

Additionally, complementing validation techniques are presented where confidence on the correctness is achieved by testing. Measurement capabilities can produce runtime observations at different levels that are compared to the corresponding prediction of the timing analyzer. Sample levels are processor core events like cache hits and number of dispatched instructions at a specific execution point or visible bus transaction signals triggered by memory accesses.

1.4.4 VHDL Predictability

Experiments with different VHDL models have revealed that the quality of the derived timing model (in terms of the precision of computed WCET bounds as well as the computational complexity) is influenced by the VHDL coding style. This thesis describes design choices together with their effect on the derivation process and thereby formulates a kind of predictability notion for VHDL language constructs along with advices to the corresponding hardware development. Minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design are the most prominent and important properties which support and simplify the semi-automatic derivation process.

1.4.5 VHDL Derivation Tool Set

The timing model derivation process has been realized by the implementation of a set of tools:

- a VHDL compiler,
- abstract interpretation-based static analyzers,
- model transformation tools,
- a pipeline analyzer generator and
- an abstract VHDL generator.

The VHDL compiler is called Vhdl2Crl2 and transforms an input VHDL design into its equivalent representation as a sequential program. By this, it implements the semantic level reduction that has been mentioned at the beginning of this section. All analysis and transformation tools operate on the generated CRL representation (cf. Section section 1.4.1 on page 8) that serves as an exchange format.

Three static analyzers are employed during the derivation process: VhdlResetAnalyzer, VhdlAssumptionBasedModelRefiner and the VhdlSlicer. The first two tools deal with the computation of initial signal assignments and the identification of unused code snippets based on given signal assumptions, respectively. Model exploration and understanding is supported by the slicing tool. All three have been developed by Schlickling [Sch13] and are used within the defined timing model derivation process.

Three model transformation tools are contained in the tool set: **VhdlTimingDeadCodeEliminator** incorporates the results of the assumption-based model refiner by automatically removing the transitive closure ("timing-dead propagation") of all marked CRL entities. By this, it effectively reduces the size of the model. **VhdlDomainAbstractor** implements automatic type changes to signals or variables of a design as a source-to-source translation where the destination domain has to be specified by the user. For example, an integer data type might be transformed into a custom-defined abstract data type representing an interval of integers. Expressions using variables/signals whose type has been transformed to the specified target domain are potentially invalid as the standard operators might not be defined for the specified target domain. In such cases, alternative operators have to be provided by the user. To support this need, the tool prints signatures of needed operators as a result of the transformation. Additionally, different application scopes are supported, i.e., the transformation effect can be restricted to a specified set of identifiers. By this, the model can be iteratively transformed. **VhdlProcessReplacer** automates the replacement of VHDL processes by custom simulation routines which are provided by the user.

The pipeline analyzer generation tool is called **PipelineAnalyzerGenerator** and realizes the abstract simulation semantics contributed by this thesis and mentioned above. A reconstruction of the timing model into an abstract VHDL can be performed by the tool **AbstractVhdlGenerator**. Its purpose is to establish a link between the derived timing model in its CRL representation and the validation of the employed model transformations because the above

mentioned interval property checking techniques are implemented within a VHDL-reading analyzer.

1.4.6 Experimental Results

Experiments with the tool implementations have been conducted to underline the industrial applicability of the approach in total.

The following VHDL designs have been examined:

- a superscalar DLX variant similar to a PowerPC 603e,
- the SPARC V8 architecture based LEON2 processor (typically used in aeronautics applications),
- a memory controller used within modern avionics systems and
- two representative automotive processors.

Except the first one, all these models represent processors or memory controller specifications that are utilized within real-world safety-critical systems. The superscalar DLX machine is an implementation from the Technical University of Darmstadt [Hor97] that is based on the DLX presented by Hennessy [HPG06]. Although the design is not industrially used, it offers features like out-of-order execution, speculation and branch prediction. Non-disclosure agreements with the particular manufacturers forbid the exposure of the original names for the anonymous avionics and automotive designs.

Runtime and memory consumption experiments show a good performance of the implemented tools along with a linear scaling in the code size on the selected hardware models. Even big processor specifications like a LEON2 (with about 70 000 lines of code) can be translated into their sequential program representation within acceptable time (about 17 min for the LEON2). The memory consumption is high with about 7 GB but this has been expected regarding the transformation's complexity. Resource consumption of the transformation and generator tools is low compared to the VHDL compiler and dominated by the size of the intermediate representation. Execution times are just below 30 s and the maximal observed memory consumption is 802 MB in the experiments.

During the derivation process employed model transformations are shown to reduce the size of input VHDL models (around 50 % for a modern memory

controller of an avionics system) enabling the generation of aiT-compatible pipeline analyzers.

Moreover, there are two different kinds of experiments underlining the competitiveness of derived timing models against hand-crafted ones. For the superscalar DLX, synthetic execution traces retrieved by VHDL simulations have been compared to the predictions of the corresponding generated timing analyzer. Results show an average overestimation of around 10 % for the predictions. Additionally, the semi-automatically derived timing model of the avionics memory controller is qualitatively compared to an existing hand-crafted model where the term hand-crafted in this case means that the model has been developed by a manual examination of the controller's VHDL design. Computed WCET bounds either are equal or over-estimate the legacy bounds only by a small percentage (less then 1 %).

1.5 Thesis Outline

After this introductory chapter, a classification of related work is given in **Chapter 2**.

Chapter 3 develops a terminological basis for "embedded systems" with its characteristic properties including typical application areas. Moreover, it introduces current hardware architectures and processor pipelines for such systems with their performance enhancing features.

The current state-of-the-art in timing analysis of embedded systems is described in **Chapter 4**. First, there is an overview of the different existing approaches and after that, the structure and functionality of the aiT framework is presented in detail.

In **Chapter 5**, formal hardware specification languages are introduced with the example of VHDL. Typical language constructs and the two-level semantics are explained and the automated synthesis of hardware circuits from formal specifications is address, as well.

Chapter 6 contains a detailed description of the workflow for timing model derivation.

Chapter 7 shows how to generate a pipeline analyzer from a derived timing model.

In **Chapter 8**, methods for the validating timing models are presented.

Considerations about timing predictability of hardware components and the impact of certain language constructs on the predictability of a derived timing model is detailed in **Chapter 9**.

After that, **Chapter 10** presents the VHDL derivation tool set implementation with the structure of the contained tools.

Corresponding experimental results on industry hardware models are then shown in **Chapter 11**.

Finally, **Chapter 12** concludes this thesis and gives an outlook to potential future work.

2

Related Work

“Honos reddatur dignis.”

This chapter describes work related to this thesis and points out the particular relation. In parts, this relations are already picked up in other chapters by citations.

The first part addresses the worst-case execution time determination and gives an overview of existing tools and techniques in this area. Because the implementations presented in this thesis base on the aiT WCET analyzer framework, a specific chapter has been dedicated to it. Thus, aiT is not mentioned in this chapter.

In Section 2.2, there is a short introduction to research work about the analysis of formal hardware specifications, in general, but also with the intention of proving safety properties.

The last section in this chapter lists capabilities of hardware simulators and discusses the difference between such a concrete system simulation and the abstract simulation performed by the combined cache and pipeline analysis as presented in Section 4.3.3.

2.1 Worst-Case Execution Time Computation

As detailed by Wilhelm [WEE+08], there are numerous tools around that are concerned with the determination of upper bounds on the execution time of tasks within embedded systems. In the following, a short description of the most prominent and usable tools is given. More details and references can be found in the WCET tool survey [WEE+08] and WCET Tool Challenge report 2011 [HHL+11]. According to the theoretical considerations on timing analysis in Section 4.2, existing methods can be categorized into two major groups, static and dynamic methods. Whereas approaches in the first group are based on static analysis of binary executables and/or source code, the second represents dynamic methods relying on runtime measurements to determine the timing bounds. In the end, there is another group, called the hybrid methods which try to combine both worlds in order to overcome their specific limitations.

2.1.1 Static Methods

Besides the aiT WCET analyzer which is described in detail in Section 4.3, there are other WCET tools that are based on static analysis. The following

sections describe all available tools that have participated in the WCET tool challenge 2011 [HHL+11] with the exception of aiT which is described in Section 4.3 in detail.

Bound-T

Bound-T [HS02] has originally been developed at Space Systems Finland Ltd. for the verification of software running in spacecrafts. Currently, Tidorum Ltd. has taken over the development.

Bound-T is based on the binary executable from which the control-flow graph is reconstructed by a decoding process. After that, static analysis like constant and copy propagation is performed and integer computations are modeled as transfer relations described by Presburger arithmetic formulas [KK67]. These transfer relations are then analyzed in order to identify loop induction variables and bounds as well as the resolution of dynamic branches. The worst-case execution path and its costs are then determined by implicit path enumeration [LM95] which is applied to each subprogram separately.

Neither caches nor dynamic hardware components like caches or out-of-order execution pipelines are considered by Bound-T.

METAMOC

METAMOC [DOT+10] is a research prototype from Uppsala university in Sweden that has been developed on top of the UPPAAL model checker. Its tool chain consists of the following different analyses: control-flow analysis, value analysis and pipeline analysis. During control-flow analysis, the input binary executable is translated into its assembly representation. The value analysis analyzed the assembly representation and determines an approximation to register values and addresses of accessed memory cells for each program point. Flow information like loop bounds that cannot be automatically computed, have to be specified by the user. The control-flow graph of the decoded assembly is reconstructed and represented as an UPPAAL model. This model is combined with a model of the underlying hardware architecture (given by the user as networks of timed automata [DOT+10]) so that the resulting model can be checked using the UPPAAL model checker whose final result is the WCET bound.

Exposing the hardware model to the user demands for a detailed knowledge about the hardware's behavior. The authors of METAMOC state that modeling complex and timing anomaly exhibiting [RWT+06] processors push the underlying model checker to its limits [DOT+10]. Similar consequences might result from imprecise flow information.

OTAWA

OTAWA [BCRS10] is an open-source framework library from IRIT and University of Toulouse intended to support the development of WCET analyzers operating on the binary executable. The supported tool chain is similar to the one of the aiT framework, i.e., a decoding phase to reconstruct the control flow of the input executable is followed by micro-architectural analyses for the determination of flow facts and execution timings based on the previous analysis results.

A generic kernel with defined interfaces allows the integration of different analyses implementations which makes the framework extensible. Functionality for loading binaries of certain architectures like PowerPC, ARM, TriCore, Sparc and HCS12 is already available. Additionally, loaders of user annotations for flow facts (loop bounds, branch targets or custom flow constraints, ...) and custom information on each control-flow graph node exist. The actual development of a timing analysis is supported by an interface that allows to plug in different WCET computation models. From that point of view, OTAWA is not a stand-alone WCET analyzer by itself.

Proof-of-concept analyzers have been implemented and participated in the WCET tool challenge 2011 [HHL+11]. There, they are described to have problems with unsupported instructions and not automatically computed flow facts.

SWEET

SWEET [Erm03, ESG+07] is the abbreviation for *Swedish Execution Time Tool* which has been developed and maintained by the Mälardalen WCET research group. SWEET operates on a custom intermediate program representation format called ALF. Its tool chain is similar to the one of aiT and consists of three phases:

- A *flow analysis* phase which consists of program slicing, pattern matching and abstract execution (symbolic execution based on abstract interpretation). It is used to determine flow facts on the analyzed program like loop bounds. These flow facts are usable only if the input program has been compiled with the integrated SWEET research compiler. Otherwise, all flow information must be given manually.
- *Processor-behavior analysis* that consists of a memory-access analysis to identify the memory accessed by instructions and a pipeline analysis performing a simulation of the analyzed program based on a cycle-accurate model of the architecture with the determined flow analysis results as additional input. This analysis phase only supports instruction caches and in-order processors with bounded long-timing effects and no timing anomalies (cf. Section 6.2.2). Out-of-order execution is explicitly not supported.
- *Estimate calculation phase* that actually calculates the worst-case execution path with its costs using implicit path enumeration [LM95] as well as fast local-path techniques.

TuBound

TuBound is a research prototype from Vienna University of Technology [PSK08]. The idea behind that tool is that annotations and optimizations can be made at the source code level while the actual timing analysis still happens at the object code level. Source code is parsed with the C/C++ frontend from the Edison Design Group into a syntax tree format which is analyzed with an inter-procedural interval analysis generated by the program analyzer generator PAG [Mar99]. By this, value ranges of variables are attached to the syntax tree. Then, loop bounds are determined via equation system solving and constraint logic programming on this annotated syntax tree which results in annotated flow constraints. After that, the input program is optimized by the ROSE source-to-source transformation and optimization framework while the annotated flow information is changed accordingly. The optimized program code is then processed with a modified compiler that preserves the annotated flow information. Worst-case execution times of basic blocks are computed by instruction table lookups which effectively prevents to model complex processor architectures with superscalar pipelines and related performance-enhancing features. The global worst-case execution path is then computed via integer linear programming over the combined equation

system of basic block execution times, control-flow graph connections and annotated flow constraints.

WCA

WCA is research prototype of a static WCET analysis tool for the JOP architecture [SPPH10]. Tool inputs are a processor specification and the java bytecode that is executed on the architecture. Additionally, annotations from the java source code can be loaded. For the execution timing, data-flow analysis is employed first to obtain loop bounds and branch targets (for dynamic method dispatching). After that, a pipeline analysis derives symbolic formulas of the worst-case execution time of bytecode instructions. WCA only supports a subset of C, called *wcetC*, enhanced by extensions for flow annotations. According to [HHL+11], the tool cooperates with a modified version of the GNU C compiler that performs abstract co-interpretation during code translation yielding an integer linear program. That program in the end is solved by an ILP solver to compute the overall timing bound.

2.1.2 Dynamic Methods

The counterpart for the above described static methods is the dynamic method. In this section, some existing tools implementing such approaches are shortly presented. Their common property is that they do not rely on static analysis to obtain the worst-case timing bounds. Instead, they basically perform runtime measurements of the analyzed program and try to infer WCET estimations from them.

Research Prototype from TU Vienna

The research prototype from TU Vienna [PN98] has no explicit tool name. Its approach is to iteratively perform runtime measurements for the computation of a WCET estimate where genetic algorithms are used to provide the input data for that measurements. An evaluation of the computed timing estimates steers the repeated generation of new input data for the next iteration of measurements. To stop the iteration, a termination criterion has to be specified by the user. As described in Section 4.2.2, such an approach cannot assure any safety guarantee on the worst-case execution time of a program, in general.

2.1.3 Hybrid Methods

The third category of WCET computations are the hybrid methods. They combine static analysis with runtime measurements.

RapiTime

RapiTime [BCP02] is the commercial version of a research tool from the University of York and distributed by Rapita System Ltd. It computes the distribution of the execution time for basic blocks of a program by performing runtime measurements where the corresponding program input has to be provided by the user. The needed runtime traces can be obtained by techniques like code instrumentation (optionally with external hardware support), non-intrusive tracing techniques (e.g. Nexus traces) or cycle-accurate simulator traces. Basic block execution times are then combined using an algebra of probability distributions [WEE+08].

FORTAS

As described by Hanxleden [HHL+11], *FORTAS* [CG11] uses a combination of measurements and static analysis where the traditional measurement-based approach is augmented by a feedback loop implemented by iterative refinements of the measurement inputs. The tool still has a prototype status.

Research Prototype from TU Vienna

TU Vienna has developed another research prototype [WRKP05] in addition to the one already mentioned above under the dynamic methods. This one uses static analysis of the source code to partition the analyzed program into segments of reasonable size and therefore yields a set of execution paths and information on infeasible paths. For each path, a model checker generates test data serving as inputs for measurements on the corresponding paths. The global WCET estimate is then computed using integer linear programming. As the flow information is based on a source code analysis, it has to be assured that the compiler does not change the program's structure. Path coverage is ensured by the static analysis part, but in the presence of complex processor pipelines, the execution time strongly varies depending on the

input state. Due to this, the needed state coverage is not provided by the tool.

2.2 Analysis of Formal Hardware Specifications

As the timing model derivation approach presented in this thesis uses formal hardware specifications as the starting point, it is interesting to distinguish it from other research work or tools that are concerned with the analysis of specification models in VHDL, Verilog or similar formal hardware specification languages.

2.2.1 Functional Verification

Formal Functional Hardware Verification

Another kind of analysis on hardware specifications exists in the area of formal functional verification where the correctness of a model according to a given specification, like the instruction set semantics, is proven. The employed methods are variants of (bounded) model checking which is common in this area. Details about the state-of-the-art in functional verification of hardware designs, *Equivalence Checking*, are presented in Section 8.3, so further descriptions are omitted here.

Testing-based Validation

Besides formal verification techniques, "simple" testing is still widely used for showing correctness properties of a circuit design. Here, the key point for the testing scenarios is the coverage of the test cases on the specification code. Therefore, tools examine such metrics for a given set of tests, for example

- the tools distributed by *TRANSEDA* (`http://www.transeda.com`) or
- the open-source tool *covered* (cf. `http://covered.sourceforge.net`).

More tools are available especially in the embedded system community. Often, this functionality is integrated into commercially available development suites for hardware circuits including synthesis tools.

Timed Automata

In [Neh04], VHDL models are transformed into timed automata specifications by a tool called *VAT*. The overall goal is to apply existing analysis techniques for such automata to hardware specifications in VHDL. Exemplarily, the finite state machine generated by VAT can be checked with the UPPAAL model checker [BLL+96]. In principle, the provided functionality seems to be directly related to the VHDL frontend Vhdl2Crl2 from the VHDL derivation tool set introduced in Chapter 10 if one abstracts from the concrete output syntax of the automata and the CRL files. The difference lies in the type of intermediate representation of the analyzed model. Where Vhdl2Crl2 generates a control-flow graph like description of the VHDL design, VAT generates timed automata that can be processed directly by a model checker.

Information Flow Analysis

T. Tolstrup presents an approach [TN06] to detect security leaks in VHDL specifications due to differences in the execution time. The idea is that the timing of cryptographic algorithms often varies depending on the input, e.g., a password or a key. An "observer" then might retrieve such passwords or keys by precisely examining the execution time differences for different inputs. A data-flow analysis is presented that could detect such possibilities by analyzing the timing difference between all paths along which an input signal can flow through the system until it reaches the modification of an output signal. Despite of its nature as a data-flow analysis, this kind of analysis is similar to a WCET analysis as the execution of a program induces signal changes in the hardware model which are integrated in the abstract simulation. It therefore might be possible to implement the proposed analysis in the environment of the VHDL derivation tool set.

Abstract Interpretation of VHDL

Charles Hymans has introduced an abstract interpretation of VHDL models that computes an approximation on the states reachable during the simulation of such a design [Hym02]. A trade-off between precision and computational complexity can be reached by changes to the employed abstract domains. In principle, its method is similar to the pipeline analysis used in the aiT framework (cf. Section 4.3.3). The difference to the timing models used there

is that Charles Hymans' work does not transform the input hardware model in any way. As in this thesis, the only purpose of the timing models is to be used for the computation of WCET bounds, the original model can (and from a complexity point of view must) be freed of all artifacts that do not contribute to the timing behavior of the specified hardware.

Hymans gives a definition of simulation semantics for VHDL language constructs including all types of expressions. In this context, he defines some predicates that have been used by the simulation semantics in Chapter 7:

- *next*:
  This predicate returns the next statement based on the current simulation position.
- *eval*:
  Evaluation of VHDL expressions is embedded into this predicate.
- *wake*:
  Repeated process execution after a value change in at least one of the signals in the process' sensitivity list is embedded in this predicate. The analogous operator in this thesis is called *repeat*.

2.3 Hardware Simulation

In addition to the different WCET tools described above in Section 2.1, there are simulators for VHDL (or similar hardware description languages) available which can be used for debugging and exploration purposes of a hardware design. In principle, the functionality of these tools is partly equal to synthesis tools as the design has to be loaded and elaborated before it can be simulated. Available tools come from two different communities, namely the academic world and some commercially distributed simulators. One representative for each community is shortly described in the following.

2.3.1 GHDL

is an open-source implementation of a VHDL frontend which has a high acceptance, i.e., nearly the whole synthesizable VHDL subset is supported. The design is directly compiled into an executable using the GNU compiler

collection (GCC)[1] whose execution represents the simulation of the hardware circuit. There is no integrated visualization feature like a graphical user interface for an exploration of the model, but timing diagrams for signal values over simulation time can be generated.

Due to the integration with GCC, there is neither any direct access nor an interface to the parsed VHDL design, although this is actually not needed for a simulation, it effectively prevents the tool's use for the timing model derivation purposes introduced by this thesis. For that, dedicated control over the generated intermediate representation is required because the different analyses and transformations need to store custom data.

2.3.2 ModelSim

The tool *ModelSim* is distributed by Mentor Graphics [Men08] and is one representative solution for an industrially available simulation tool. It supports even the newest version (from 2008) of the VHDL standard as well as (System-)Verilog and can be used for interactive debugging and exploration by featuring a sophisticated graphical user interface. Furthermore, ModelSim has an integrated heuristic to determine a topological order of the different VHDL modules. This is a convenient feature because most of the existing tools let the user determine the order in which the different VHDL files are fed to the parser. And depending on the hardware to be modeled, such dependencies can become rather complex and unclear. Due to its features, ModelSim can be used for a testing-based functional verification in addition to "simple" model simulation. But as with , there doesn't exist any externally accessible interface to internal data structures of a loaded hardware model.

2.3.3 Summary

Besides the two given examples for hardware simulators, there are others available. For a potential usage for the derivation of timing models from formal hardware specifications, there was always the problem of not having access to the internal data structures of a loaded design. This actually is the reason why Vhdl2Crl2 (cf. Chapter 10) has been developed. Due to cost reasons, an evaluation of different synthesis tools has not been performed.

[1] `http://gcc.gnu.org`

Another common problem especially for the freely available tools is that they most often do not accept the full synthesizable VHDL subset.

In general, hardware simulation is no alternative for analyzing the timing behavior of systems because:

- Modern real world processors (like Freescale PowerPC 755 [Fre01]) are far too complex. The computational complexity of the simulation would render the method infeasible for industrial usage.
- Due to the complexity of the problem, nondeterminism is introduced into the computation model. No traditional VHDL simulator can cope with that.

To the best of the authors knowledge, nobody has tried before to automate (at least partly) the development process of a timing model for processors with the goal of WCET determination.

3

Embedded Systems

"I wanted to change the world. But I have found that the only thing one can be sure of changing is oneself."

(Aldous Huxley)

This chapter gives an introduction to *embedded systems* in general, starting from a definition, listing common characteristics of such systems in Section 3.1 and describing application areas for them in Section 3.2. Moreover, an overview of commonly used hardware architectures is given with a focus on the structure and properties of the processor pipelines in Section 3.3.

3.1 Overview

According to [Mar05], an embedded system is an information processing system embedded into a larger product. This means that the system itself normally is not directly visible to its user. For example, there are embedded systems within modern cars processing sensor inputs but the driver itself is not aware of them besides some notification icons in the cockpit.

From a historical point of view embedded systems represent the newest of three eras of computer systems:

- *Mainframe computing systems* mainly used by scientists and large companies have build the starting era until the late eighties.
- *Personal computing systems* then were introduced during the nineties offering computing power to "everyone".
- *Embedded systems* represent the usage of computers within enclosing products of every day life to provide omnipresent information.

As stated by Mark Weiser [Wei95], the growing demand for ubiquitous computing will more and more cause the disappearance of personal computers in favor of the "Internet of Things". This means that smaller and smaller objects which are connected to different networks (like the Internet), simplifies information retrieval.

Technically, an embedded system is not necessarily always decomposed into the actual application code and an underlying operating system. Especially for small systems, the latter might be missing so that the application needs to take care about low-level functionality like the communication to peripheral devices. Otherwise, employed operating systems are typically specialized and have been specifically developed for their purpose [Win, QNX]. But there are also customized versions of standard operating systems like Linux [Lin] and Windows 7 [Mic]. The actual application is often structured into a set of

different processes. If the processes are scheduled by an operating system, they are usually called *tasks*.

3.1.1 Characteristics

Embedded systems are designed for specific purposes, so there are different characteristic properties of such systems. The following list presents an overview of such properties. Depending on the concrete application scenario, the importance of the different properties varies which also influences particularly required limits.

Reliability The reliability is the likeliness that a system will not fail, i.e., the system does not work as expected. This is often expressed in terms of probabilities.

Maintainability The maintainability of a system describes how much effort is needed to keep the system in a working state. This either means to prevent failures by providing regular support as well as the average time to repair a failing system.

Availability The availability of a system is the percentage of time when it is in an operational condition and is strongly connected to the reliability and maintainability. Without a high reliability and maintainability, the availability of that system cannot be high.

Safety The safety of a system is a metric that measures the probability of the occurrence of a failure that lead to hazards for either human life, property or the system's environment. Safety standards like the ISO 26262 [ISO11] define corresponding limits for these probabilities.

Security The security of a system defines whether failures in that system can reveal any confidential data stored in it.

Efficiency The efficiency of an embedded system addresses different aspects:

- *Energy*
  The power consumption of an embedded system is important especially if the power source is a battery. On the one hand, the computational demands of embedded applications are growing rapidly but on the other hand, the battery technology only increases at a slow rate. So, the power consumption needs to be minimized in order to achieve long runtimes for the systems.

- *Code size*
  The executed software needs to be stored within the embedded system. Often the storage capacities are restricted due to size and cost reasons, i.e., because the final system cannot effort to exceed a certain size and more storage capacities increase the production costs. This is also important for so-called systems on a chip (SoCs) integrating all components of a computer or other electronic systems into a single integrated circuit (chip). Storing the instruction memory on that chip raises the requirement to minimize the code size.

- *Runtime efficiency*
  This topic covers different aspects:

  - the system only contains the minimum amount of resources or hardware components,

  - the clock frequencies as well as the supply voltage are minimized and

  - existing timing constraints must be met.

- *Weight*
  The weight of the system often needs to be minimized, especially for mobile devices.

- *Cost*
  The production costs – especially for high-volume products in consumer electronics – are an important property. Often, these costs decide over the competitiveness of the system compared to similar products of another manufacturer.

The different aspects are not independent of each other and partly counter-productive. For example, reducing the weight of a system can be achieved by smaller batteries. But ignoring possible technological improvements these batteries provide less energy.

Application Embedded systems are designed for a single purpose and not for general purpose computing. Once produced, they will work for that purpose during their product life time. The reasons for that principle are simply that the more functions a system provides, the smaller is the dependability, reliability, maintainability, etc. Furthermore, this would have the consequence that resources which are only used for providing one of the multiple functions, often remain unused.

User Interface In contrast to personal computers embedded systems often do not have keyboards, mouses or monitors. Instead, there is a dedicated user interface like buttons, steering wheels, pedals, etc. This is due to the fact that such systems are not designed to process arbitrary kind of input data. They work within a restricted operational environment which influences their input/output interfaces.

Real-time constraints Real-time constraints define a time window for the completion time of a task. Depending on the consequences of not meeting a deadline for task completion, two categories of real-time constraints are distinguished:

- *Hard real-time constraints*:
  A time constraint is called *hard* if not meeting that constraint could result in a catastrophe [Kop97]. If an airbag control software in a car does not activate the airbag fast enough, this might cause severe injuries.

- *Soft real-time constraints*:
  All time constraints that are not hard are called *soft* time constraints. If a DVD player cannot deliver a video frame in time, the playback starts jerking. Despite lowering the quality of the player, this does not endanger human life.

Hybrid Systems An embedded system is called a hybrid system if it contains both analog as well as digital parts. Analog parts are specified in continuous time, e.g., by differential equations, so that the system state changes continuously within certain invariants. In contrast, digital parts are captured in discrete time, e.g., by control graphs. State changes happen according to specific conditions and typically result from corresponding events.

Reactive Systems A reactive system is an embedded system that is in continuous interaction with its environment. It is usually waiting for an input from its environment, then performs some computations on the input to produce some output. The computation results in a new state of the system again waiting for input. Therefore, automata can be used to describe the behavior of reactive systems.

3.1.2 Real-Time Systems

The response time of a system denotes the time span between the activation of a task and the end of the associated computation together with a potential output. As mentioned in Section 3.1.1, real-time constraints define bounds on the response time of a system. These bounds have to be fulfilled regardless of the current load of the system and the absolute values of the bounds.

Definition 3.1 — Real-time System:
An embedded system for which real-time constraints exist is called a *real-time system*.

Because of the timing effects of swapping, heap accesses and hardware interrupts, timing constraints cannot be guaranteed in general so that scheduling and memory management have to be treated specifically. Operating systems providing timing guarantees for system calls are named *real-time operating systems (RTOS)*.

As mentioned above, a mentionable number of embedded systems can be characterized as safety-critical. This means that system failures, e.g., due to not meeting a hard real-time constraint, are not acceptable since they may cause severe damage or even loss of lifes. A hard real-time system for which missing a deadline could cause damage to the product and/or its users is called *safety-critical*.

Because of the severity of a system failure the manufacturer of such systems are forced by law to certify their product before they are allowed to ship it. The certification aims at ensuring system safety by demonstrating that no safety hazards can occur. Furthermore, often there are specific requirements for the development process to ensure that the complete product life cycle satisfies quality assurance requirements [Wol00]. For example, the ISO 26262 [ISO11], a modern safety standard for road vehicles, demands for a limitation of the "observable incident rate" to be less than one billion (10^9) per hour of operation at the highest safety level. Incident rate in this context refers to a failure that has the potential to lead to the violation of a safety goal.

The software of a real-time system often consists of a set of tasks which are executed periodically and implement a certain sub-function of the whole system. Among the tasks, often there are dependencies, i.e., one task must finish its execution before another one starts. Besides such dependencies, there are other parameters which could be used to define an order among the set of tasks:

- *release time*: the minimal amount of time before a task can be started.
- *priority*: imposes a total order on the set of tasks and is used to decide which task to execute among multiple active tasks. The priority can be assigned statically or dynamically which means that they are assigned in a fixed way or that they could change during execution.
- *period*: the total time between two invocations of a task.

Based on these parameters, different scheduling strategies can be used to decide the task's execution order.

Depending on the schedule creation time, algorithms can be categorized as *online* or *offline* scheduling. Offline means that the task execution order is determined before the start of the whole system. In contrast, online scheduling algorithms decide that during system runtime.

Another property of the scheduling algorithm is whether it is *preemptive* or not, i.e., whether the execution of a task can be interrupted by the execution of another task that has become ready.

Static-priority scheduling algorithms operate on a fixed set of processes and computes a fixed schedule that does not change whereas these restrictions does not hold for the *dynamic-priority* variant.

An example for a static scheduling algorithm is *rate monotonic* scheduling [LL73]. Here, priorities are assigned by the inverse length of the period. This means that the task with the smallest period is assigned the highest priority. A dynamic scheduling algorithm is called *earliest-deadline-first (EDF)*. It assigns the highest priority to the task whose deadline is nearest. EDF with preemption is known to be optimal on a uni-processor system, cf. [Liu00].

To show whether a set of tasks with given parameters and a given scheduling algorithm is feasible, i.e., whether all deadlines, release times, ..., hold, the worst-case execution time of each single task has to be computed first. And in order to get this time, it is not sufficient to rely on statistical arguments [Kop97]. More details about the timing analysis of embedded systems can be found in Chapter 4.

3.2 Application Areas

In order to give an overview, the following list covers the most important application areas of embedded systems:

3.2.1 Automotive Electronics

There are more and more electronic devices in modern cars. Example applications are:

- *technical assisting systems*: brake-by-wire, adaptive-cruise control, powertrain management, or exterior light control systems,
- *safety assistant systems*: electronic stability program (ESP), anti-lock breaking system (ABS), emergency break assistant or airbag controllers,
- *driver assistant systems*: head-up displays or navigation systems or
- *multimedia systems*.

All these systems are highly integrated and communicate over different bus systems in the vehicle. Consequently, a modern car contains more than a hundred microcontrollers and different communication networks which impose a high system complexity.

3.2.2 Aviation

The term aviation combines the aircraft manufacture, development and design. It represents a global industrial sector with lots of subcontractors and a growing number of electronic systems specifically developed for the use in aerospace vehicles. These systems are called avionics and nearly all of them are safety-critical:

- *flight control systems* in fly-by-wire controlled airplanes which are responsible for the whole flight state of the aircraft,
- *engine control systems* which maintain the engine status including thrust level and alarm detection,
- *flight planning systems* which compute fuel consumption forecasts, navigation information or
- *cabin pressure control systems*.

Failures in such systems often directly lead to severe damage or even loss of lifes. Because of that some sort of redundancy often is required. Typical strategies are:

- *Hardware redundancy:*
  Employed methods are *Dual-* and *Triple-Modular-Redundancy (DMR/TMR)*. For DMR, two identical systems run in parallel and the second one serves as fail-safe for the first. In the case of TMR, three identical systems run in parallel and there is a so-called voter that compares the results of the three systems. The voting mechanism is a so-called "two-out-of-three voting", i.e., a failure of one system can be overridden by the two others.
- *Information redundancy:*
  Information redundancy is achieved by error checking and correction methods. One example for such a redundancy is the ECC protection of memory chips.

3.2.3 Railway Electronics

Because train traffic is controlled in a centralized way, i.e., trains are not able to change their track autonomously, there are different control applications in the railroad market:

- *Automatic train supervision*:
  They order and manage the traffic of complete railroad networks by signal controlling and interact with an operator.
- *Automatic train control*:
  Such control systems assist the platoon leader or even drive the train automatically including stopping at the right positions at stations, opening/closing of the doors as well as speed and distance control.
- *Automatic train protection*:
  These systems constantly control the position and speed of all trains in a network and initiates appropriate actions to prevent accidents, e.g., by automatically stopping trains that would collide otherwise.

Besides those control applications, there are other not directly safety-related system in train vehicles like passenger information systems. In the past, such systems were installed on mainlines and metropolitan mass transit networks, but they are more and more introduced on regional tracks.

3.2.4 Telecommunication

Telecommunication is about sending and receiving messages and in the past, visual and acoustical signal have been the transmission instruments. Nowadays, with electric devices, the word-wide web, and the invention of different communication protocols like wireless networks, the term telecommunication is more than just end-to-end communication. Developed for practical reasons, communication devices like mobile phones are today merchandised with emphasis on the consumer's emotions. And recently, smartphones have started capturing the market by promoting the integration with social networks. Therefore, this area is one of the fastest growing markets of the recent years as stated by the marketing research company "Insight Research Corporation" [Ins10].

3.2.5 Medical Engineering

The area of healthcare is a quickly growing market for embedded systems as well. The technology requirements here go beyond avoidance of failure and can even facilitate medical intensive care (which deals with healing patients),

and also life support (which is for stabilizing patients). The challenges for these systems typically are:

- *Data rates* are increasing tremendously.
- *Real-time constraints* exist, especially for imaging systems.
- Medical equipment is expected to take *enormous scientific knowledge* into account with decreasing human interaction.
- Users need a *highly interconnected and integrated* set of systems.
- *Robustness*.
- *Sterility*.
- *Updatability* due to changed or augmented knowledge.

3.2.6 Military Applications

Military systems probably were one of the first application areas of embedded systems but that development has not been published widely for confidentiality reasons. In this field, there is a variety of application scenarios. Some of them are

- *weapon systems*,
- *weapon guidance systems*,
- *radar and sonar technology*,
- *military avionics systems*,
- *encrypted communication systems*.

The challenge for the development of those systems often is their demanding working condition where they need to withstand extreme heat, humidity, altitude and multiple other extreme environmental conditions.

3.2.7 Authentication Systems

Embedded systems can be used for authentication purposes, for example in:

- *advanced payment systems*,
- *finger print sensors*,
- *face recognition systems* or
- *full body scanners*.

There is a growing demand for such automated authentication because business transactions are more and more shifted into the world-wide web.

3.2.8 Consumer Electronics

Systems of this area are produced for our every day life and nowadays, there is a broad perception of this terminology. Consumer electronics products are typically used for the entertainment, communication or office productivity. For example, personal computers, telephones, MP3 players, audio/video equipment, televisions, calculators, digital cameras or mobile phones belong to this application area which has merged to a large extend with the computer industry due to the increasing digital technology.

Due to better quality and new service requirements, the information processing in such systems needs to make use of more and more sophisticate digital signal processing techniques as well as high performance processors and memory systems. Furthermore, the connectivity plays an important role for streaming applications. As a consequence of this connectivity, this and the telecommunication area merge more and more.

3.2.9 Fabrication Equipment

This is a traditional area for the usage of embedded systems. Already since decades, machines are used to automate the manufacturing process more and more. Nowadays, they are programmable reducing the human involvement to monitoring tasks. Less and less work has to be done manually. This is especially the case for automotive industry.

3.2.10 Smart Buildings

Smart buildings can increase the comfort level, reduce the energy consumption and improve the safety and security of the inhabitants. The main requirement for this is the integration of traditionally independent systems or sensors into one big system: air-conditioning, lighting, access control and more. For example, the air-conditioning can alter the temperature in empty rooms, activation and deactivation of lights can be done automatically when entering/leaving the room, access controls can make the traditional key obsolete and similar functionality. Currently, such techniques are only used for modern high-tech buildings but might get introduced in private houses in the future as well.

As can be seen by this list of application areas, embedded systems are distributed among different industries and their usage and relevance is growing faster and faster. According to Jim Turley from the *Linley Group*: "...only two percent of the world's microprocessor chips go into PCs. The other 98 % are used in ...embedded systems..." [Tur09].

3.3 Architectures

3.3.1 Overview

As mentioned in Section 3.1.1, there are characteristic properties of embedded systems. This has lead to rather specific system architectures designed for special purposes whereas the goal of personal computers was to provide general purpose computing power.

All this has influenced the design of hardware architectures for embedded systems which can be categorized by the employed microprocessor architecture as well as the connected peripheral components.

The internal structure of a microprocessor is often referred to as *processor pipeline* and defines how its assembler instructions are executed. Overall goal is to maximize the throughput of executed instructions per time unit. The microprocessor architectures can be grouped into different processor families.

VLIW processors VLIW stands for "very large instruction word" because one VLIW instruction consists of several[1] micro-operations. VLIW processors belong to a modern architecture with multiple functional units to which all micro-operations of one VLIW instruction are issued. This causes execution parallelism on the instruction level. The advantages of this architecture are simple control paths and good compiler optimizations [Kä00, PK05]. Such processors are mainly used for digital signal processing in media applications. An example of a modern VLIW processor is the TriMedia TM3270 processor [Wae06] which has been developed by Philips Semiconductors.

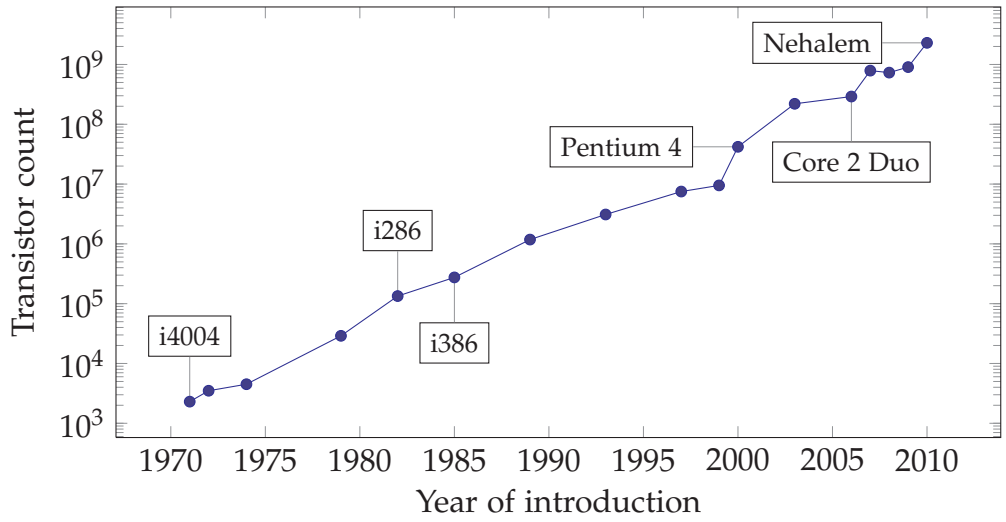
RISC processors RISC – reduced instruction set computer – architectures have a rather simple instruction set that allows a good balanced pipeline with high clock frequencies. Simple instruction set means that complex instructions combining slow memory accesses with arithmetic operations are left out. Instead, there are special instructions (`load`, `store`) for memory accesses. Complex addressing modes are omitted. All this decreases the pipeline depth, reduces the risk of pipeline hazards (cf. Section 3.4) and speeds up the instruction decoding. RISC processors are probably the most popular processor architecture for embedded systems due to their high efficiency in energy consumption and size as well as to their performance. An example of a modern RISC processor is the Freescale PowerPC 755 [Fre01].

CISC processors CISC is an abbreviation for "complex instruction set computer". In contrast to RISC processors, this category represents architectures with complex addressing modes and combinations of memory accesses with arithmetic operations within one instruction. The design philosophy is orthogonal to the family of RISC processors. In general, for a given level of performance, a CISC design will typically consists of more transistors than a corresponding RISC design. An example for a CISC processor is the Motorola M68020 processor [Mot92].

FPGA FPGA stands for "Field Programmable Gate Array" and represents a configurable integrated circuit. It's components are called logic blocks and contain programmable logic that can perform complex combinational logic or represents simple logic gates. Often, blocks are associated with storage elements like flip-flops or more complex memory blocks. The components

[1] a fixed number for a specific processor

Figure 3.1 – CPU Transistor Counts 1971-2010 - Moore's Law

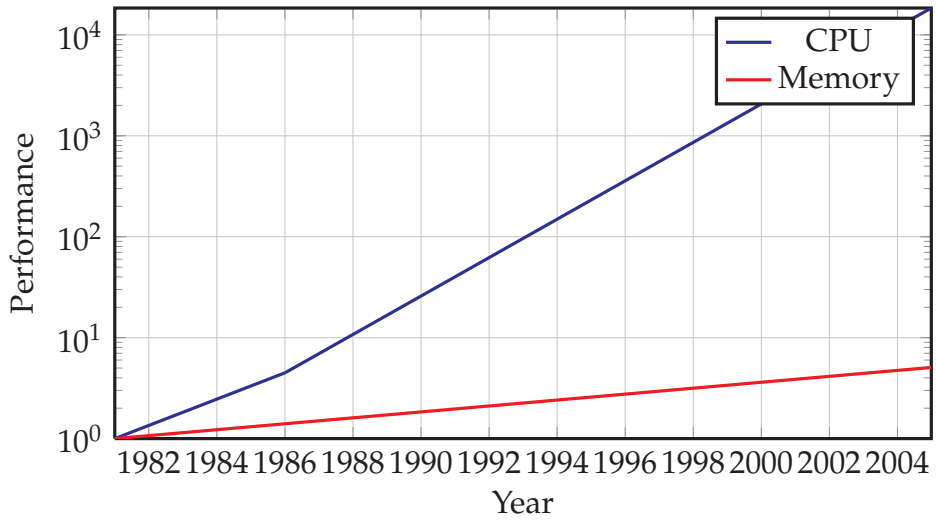


of a FPGA circuit are wired over configurable interconnect buses. By this, simple circuits like counters as well as highly complex ones like whole microprocessors can be realized. The advantages of this method are its flexibility and low development costs because chip fabrication is replaced by programming the logic blocks and their wiring. Design changes can be quickly incorporated without an expensive new manufacturing contract.

In the last decades, the number of components in integrated circuits had doubled every year from their invention until today as it has been predicted by Moore's Law [Moo65]. Based on a listing of transistor counts published on `en.wikipedia.org/wiki/Transistor_count`, Figure 3.1 shows the evolution of the number of transistors from 1970 up to now. The vertical axis is in logarithmic scale so that the nearly linear increase corresponds to an exponential growth of the number of transistors in relation to the years. Each point in this figure marks the introduction of a new processor architecture, some rather popular ones are marked.

This claim for computing power has lead to a trend of using older generations of desktop and server processors as microcontrollers within embedded systems although they were not originally designed for that purposes. Architectures with sophisticated processor pipelines and performance-improving features (cf. Section 3.4 on page 53) have been introduced. Now, the bottleneck in the computing power is not the processor core anymore, but the discrepancy between the processor and memory bandwidth. Depending on

Figure 3.2 – Performance gap between CPU and main memory 1981-2005



the application[2], a processor core spends most of the execution time waiting for memory accesses to finish. Figure 3.2 shows the evolution of processor and memory performance in the years 1981 up to 2005 starting with a basic value of one in 1981. This picture has been originally published by Mahapatra [MV99].

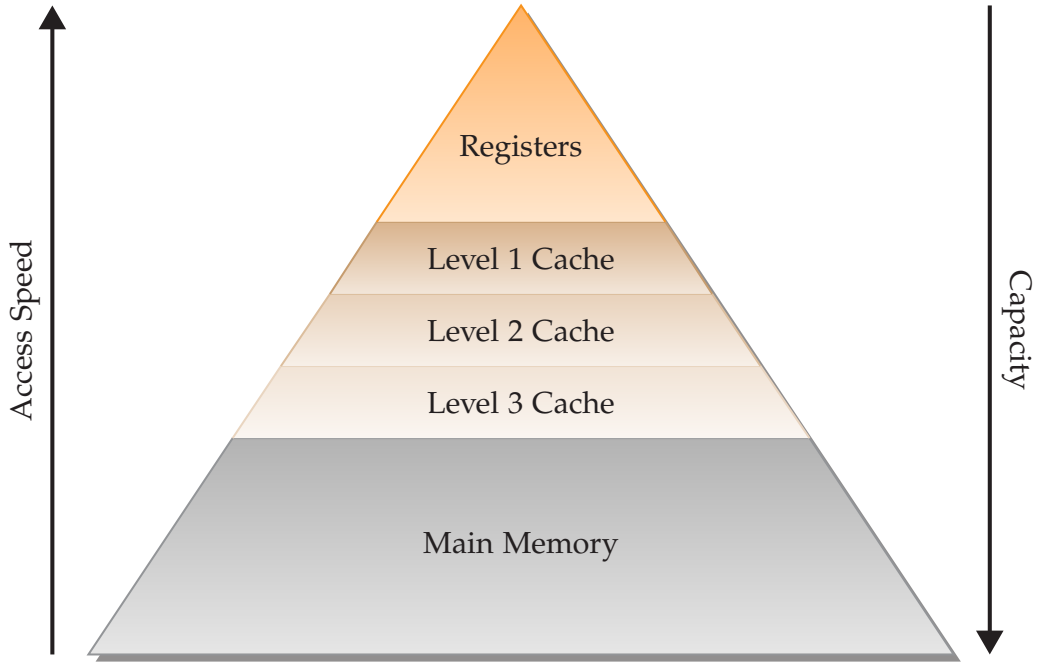
As stated by Hennessy [HPG06], the processor performance was increasing yearly by 35 % between 1981 and 1986, from then the growth rate has been even 55 % yearly. In contrast to that, the memory performance only has been increasing by 7 % yearly in the average.

This problem has been solved by computer architects by introducing memory hierarchies. Here, small but fast memories try to cache data loaded by or written from the core before accessing the larger but slow main memory. The next section describes the functionality of such caches and memory hierarchies.

The last innovation of demand for computing power was the introduction of multi-core architectures for embedded systems. Despite its average system performance increases, this step really complicates the determination of safe upper bounds on the worst-case execution time of tasks on such systems.

[2]This is especially true for most embedded software due to their input/output heaviness.

Figure 3.3 – Memory Hierarchies



3.3.2 Memory Hierarchies

As mentioned above, there is a gap between processor and memory performance. Fast memory is expensive in construction and arbitrary small access latencies cannot be guaranteed for arbitrarily large memory. To bridge this gap at least to a certain amount, memory hierarchies were introduced. This term represents a hierarchical composition of technologically different memories with increasing capacity and decreasing access speed.

Figure 3.3 illustrates this memory hierarchy. The register file of a processor is the fastest accessible storage element. It is part of the processor core and directly connected to the execution units. A register file consists of registers where each is able to store a single data, i.e., a field of – usually 8, 16, 32 or 64 – bits. The access latency to a register usually is just one processor core clock cycle.

Data, that is not stored in any processor register, is retrieved from the next higher level in the memory hierarchy, which typically are the processor caches. They are subdivided into one, two or three different levels on their

own where the levels differ in their sizes and access latencies according to the above mentioned hierarchy principle. A detailed description of processor caches can be found in the following section.

In the end, main memory represents the largest but also slowest memory. There are different types of main memory which differ in their access times. Furthermore, an embedded system is typically provided with different memory types in parallel. For example, more critical data is stored in faster memory. Typical memory types and their determining characteristics are described in Section 3.3.4.

3.3.3 Caches

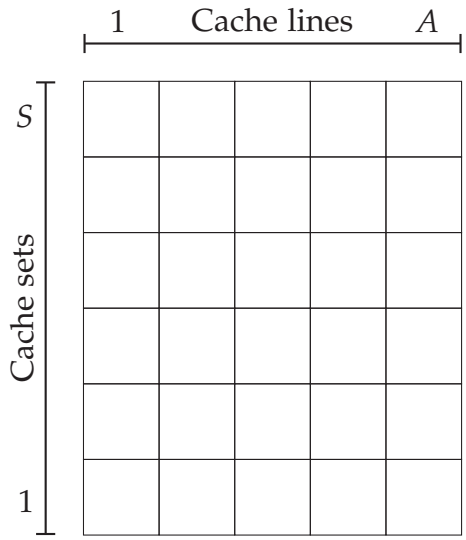
Processor caches are small but fast memories interposed between the processor core and the slow main memory. Their location is either on the processor's circuit board or they are even integrated into the core. Depending on that, the caches operate on different clock frequencies. They are managed by the hardware, i.e., their functional behavior is transparent to the software level. There can be different levels of caches that are inclusive, i.e., the content of cache level i is retrieved from cache level $i + 1$. The highest cache level then is connected to the main memory over a bus. For a typical personal computer, there are three levels – *L1*, *L2* and *L3* – of caches where the L1 cache is the fastest but smallest and the L3 cache is the slowest but largest memory among the cache levels.

The idea behind the caches is to temporarily store data from the main memory in order to deliver it with low latency to the processor core. If the result of a memory request is already located in the *Li* cache, this is called a *Li cache hit* and the requested data can be directly retrieved from that cache level. Otherwise, the situation is called *Li cache miss* and the data has to be retrieved from the next cache level or the main memory respectively.

The *cache hit rate* is the average ratio between the cache hits and the overall number of requests to the cache. Having a high hit rate results in an average memory performance near the latency of the L1 cache but due to the size of the caches the average costs per bit of the whole system are near the costs per bit of the (cheap) main memory.

The organizational structure of a cache is a matrix with S rows – *cache sets* – and A columns – *cache lines* as illustrated by Figure 3.4 on the facing page. Each line contains the following data:

Figure 3.4 – Cache structure



- the *tag* which is a part of the address (cf. below at the cache location computation),
- the actual data and
- some *status bits*: dirty, valid and cache coherency bits.

If the dirty bit is set, the data in the associated cache line has not been written back to the next cache level or main memory. An activated valid bit shows whether the corresponding cache line contains loaded data. Cache coherency bits [HPG06] are needed for the consistency of data in local caches that has been loaded from shared resources like a shared L2 cache. For example, assuming there are two processors, p_1 and p_2, that share an L2 cache and both have loaded the same data d to their private L1 caches (in line l_1 and l_2, respectively). If p_1 writes into the allocated cache line l_1, p_2 has to be notified that its "copy" of d is not up-to-date anymore. Such synchronization information is stored in the coherency bits.

The characterizing properties of a cache are:

- the *associativity* A is the number of cache lines that are contained in each cache set,
- the *line size* L is the size of a cache line,
- the *number of sets* S,

- the *replacement policy* (see below) and
- the *capacity* C which is computed by $C = S * A * L$.

If each cache line represents its own cache set, i.e., $A = 1$, the cache is called *direct-mapped* because a memory address is mapped to one single cache line. The cache is called *fully associative* if the associativity is equal to the number of cache sets, i.e., $A = S$. This means that an arbitrary memory address can be mapped to an arbitrary empty cache set. If the associativity has been chosen somewhere between these limits, this is a *A-way set-associative* cache which is a compromise between direct-mapped and fully associative.

The physically addressable memory space is distributed among the cache sets depending on a hash function which takes the associativity into account. Typically, the hash function is just a modulo operation to the number of sets, i.e., an address a can be stored in one of the cache lines of set $s = (a \div L) \bmod S$. By this, a cache line only need to store a significant number of bits of a. Those bits are called tag. In the case of a cache miss, not only the requested amount of data (e.g., a byte) is retrieved from the next cache level or the main memory. Instead the whole cache line is filled.

Due to the nature of the modulo computation and depending on the size of the main memory, different addresses are mapped to the same cache set. If all cache lines an address a is mapped to are occupied by other data, one cache line needs to be emptied before retrieval of the requested data. The decision which cache line will be emptied then depends on the so-called *replacement policy*. There are different heuristics for the replacement:

- *Least Recently Used (LRU)*
  The strategy is to replace that line in the set whose referencing is the oldest, i.e., the memory cells stored in that line have not been accessed for a longer time than the cells of all other cache lines. LRU is easy to implement by maintaining an age for each cache line and old lines are specifically chosen which leads to good hit rates in average. Furthermore, the predictability of LRU is the best according to [Rei08]. But this policy produces just a moderate hit rate and requires status bits for higher associativity [Rei08]. Therefore, this policy is only rarely implemented.
- *Pseudo-Least Recently Used (PLRU)*
  PLRU is an efficient tree-based algorithm to implement the LRU policy. The computational model behind PLRU is a binary search tree where the cache lines are represented by the leaves of the tree. Each node in

the tree stores a bit whose values indicates the direction to go when walking from the root node to a child. The values 0 and 1 point to the left and right sub-tree, respectively. Thus, the bits in the search always mark the cache line that is chosen for the next replacement. If that happens, the bits on the path to the replaced line are inverted, so that the next line to be replaced is selected. PLRU's low implementation costs and only slightly worse hit rate compared to LRU do explain its popularity among the processor manufacturers.

- *First-in First-out (FIFO)*
  This policy works like a queue. The choice of the line to replace falls to the line that was first filled in the particular cache set. FIFO produces slightly worse hit rates then LRU in the average, but has lower hit latencies [Rei08]. The disadvantage of this replacement policy is that its performance is not predictable. But the implementation costs are low.

- *Most recently used (MRU)*
  MRU *discards*, in contrast to LRU, the most recently used items first. It is most useful in situations where the older an item is the more likely it is to be accessed. According to [Rei08], the most recently used policy is impossible to predict at all.

All layers in the memory hierarchy need to be synchronized, i.e., if data is stored in the cache, the hardware needs to ensure that this data is written back to main memory at last when the corresponding cache line will be replaced. Due to performance reasons of data transfer over the system bus, there are two different strategies – *write policies* – how to do this:

- *write through*: for a store operation the data is always written back to the next higher level in the memory hierarchy. This ensures consistency of data by construction. Usually there is a write buffer for such writes so that the core does not need to wait.

- *write back*: the data is not directly written back to the next higher memory hierarchy level. The write back operation takes place when the cache line is replaced from the set. In order to remember whether a line has been written back, there is the dirty bit as mentioned above.

The write back policy might impose more difficulties on the timing predictability depending on the employed cache replacement policy because the eviction of a cache line cannot be predicted precisely. Further details on this can be found in Chapter 9.

In general, two types of caches are distinguished: *instruction* containing executable instructions and *data* caches for increasing data read or store operations. Additionally, there is the so-called *unified* cache which can contain both instructions and data. Typical cache configurations separate the data paths between instruction fetching and data accesses, i.e., there are two separate L1 caches, one instruction cache and one data cache. Often, the second cache level is a unified cache.

3.3.4 Memory

The main memory is the highest level in the memory hierarchy (cf. Figure 3.3 on page 45) and connected to the processor (over the cache components). Typically, embedded systems have a high load in their input/output behavior which results in a high number of memory accesses. Therefore, the main memory is the component in the system with the greatest influence on the execution timing.

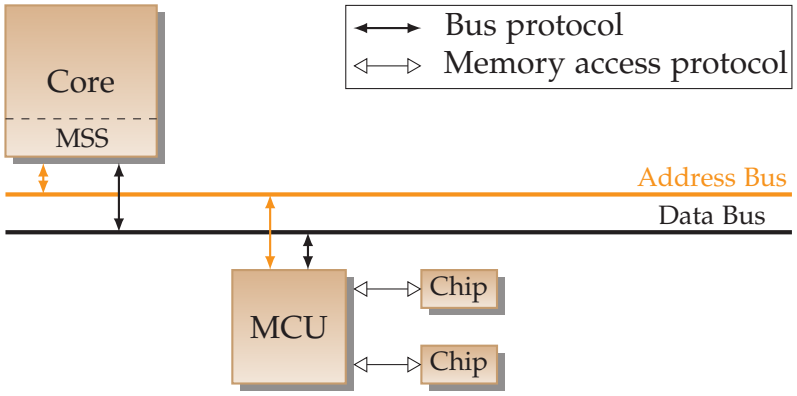
Memory accesses to the main memory[3] are started by the processor core's memory subsystem (MSS). From there, the request is passed over to the system's memory controller which is typically integrated into the main system controller – *the northbridge*. The communication between the core and the memory controller is done over the system bus which typically is divided into an address bus and a data bus. Here, a specific communication protocol, e.g., the *60x bus protocol* for some PowerPCs, is employed. The memory controller translates the cores request into the access protocol of the particular memory chips and potentially delivers the answer of the memory to the processor again over the system bus. This communication structure is illustrated in Figure 3.5 on the facing page.

There are different types of memory devices which all are different in their behavior and performance:

Static Random-Access-Memory (SRAM) The internal storage cells are implemented by transistor technology. Therefore, this memory type has low access latencies – about five to 100 nanoseconds – which are constant. Unfortunately, this technology is expensive and its needed circuit space per bit is high. This type of memory is often used for caches or systems with low

[3] after all cache handling

Figure 3.5 – Bus and memory communication



memory consumption. The timing predictability (cf. Chapter 9) of SRAM is high due to the constant access latencies.

Dynamic Random-Access-Memory (DRAM) The internal storage cells are implemented via small capacitors which allow a tight packing of the cells. By this, cheap and large memories can be constructed compared to SRAM, but the access latencies are typically about 16 times slower than SRAM. Because this memory has a dynamic characteristic as described below, the access latencies are not constant and depend on the history of the last accesses.

As the technology has evolved over the years, there are different types of DRAM concerning when exactly a chip delivers requested data: *synchronous dynamic* RAM (SDRAM) returns data only at rising clock edges, *double data-rate* RAM (DDR-SDRAM) is able to send data beats both on rising and falling clock edges. Analogously, there are also so-called *quad data-rate* RAM (QDR) and even *octal data-rate* RAM (ODR). Both share the idea to combine the technology of DDR-RAM and two or four read/write ports respectively.

The structure of dynamic memory modules often underlies the same principle. Logically, the addressable memory for one memory chip is divided into equally sized banks, each one represents their part of the memory as a matrix. To access a specific address, the responsible memory bank must be determined. Within the bank the specific row in the matrix of that address has to be selected first. All memory cells in the same row form one so-called *page* and the selection process is called to *open a page*. The memory controllers typically store the currently selected row for each logical bank, so

a subsequent access that goes into the same page can be answered without a previous row selection (*page hit*). Otherwise, another row has to be selected before the request can be answered (*page miss*).

As the memory cells in dynamic RAM are implemented by capacitors, their content needs to be refreshed from time to time. These refreshes are asynchronous events which do complicate a precise timing analysis of such memories (cf. Chapter 9).

Scratchpad Memory Scratchpad memories, often just called scratchpads, are internal (on-chip) memories where a processor core can privately use for local and temporary data storage. They can be compared to L1 caches in terms of their small capacity and small access latencies. But where caches are self-controlled by the circuits, scratchpads are software-managed, i.e., the running program needs to explicitly decide about what data to place into the scratchpad memory.

Marwedel et al. [MWV+04] shows that architecture-aware compilers can increase the overall system performance in the presence of scratchpads by about 50 %. They also state that the timing behavior of accesses into scratch memory is well predictable rendering this type of memory ideal for WCET analysis.

3.3.5 Buses

A bus is a subsystem for transferring data between different components inside a computer, between a computer and its peripheral devices, or between different computers. In contrast to point-to-point connections, a bus typically connects more than two peripherals over the same set of wires. Independent of the number of connected devices, dedicated protocols control and synchronize the transactions. In general, buses can be classified by the involved components:

- *system buses* like the 60x bus on the PowerPC [Fre04],
- *memory buses* connecting the memory controller with the memory slots,
- *internal computer buses* like Peripheral Component Interconnect (PCI) [PCI98] and
- *external computer buses* like CAN [CAN03] or FlexRay [Fle10].

Typically, the buses are clocked with lower frequencies than the processor core, e.g., the PCI bus is specified to 33 MHz in Revision 2.0 and to 66 MHz in Revision 2.1 of the PCI standard.

In contrast to the categorization by the involved components, buses can be divided into serial and parallel buses. Parallel buses support the interleaving of consecutive accesses resulting in an enhanced performance and reduced idle time. This interleaving is called bus pipelining and the possible number of interleaved and therefore pending accesses is the pipeline depth. To implement this parallelism, such buses are separated into an address bus and a data bus.

3.3.6 Peripheral Devices

An embedded system interacts massively with its environment, i.e., input data is received and computed results are emitted. This environmental interaction is implemented by the communication with peripheral devices. Mainly, these are sensors and actuators which are accessed by special input/output instructions or memory accesses in case of memory mapped devices.

Other peripheral devices are memory controllers which are connected to the CPU over the system bus.

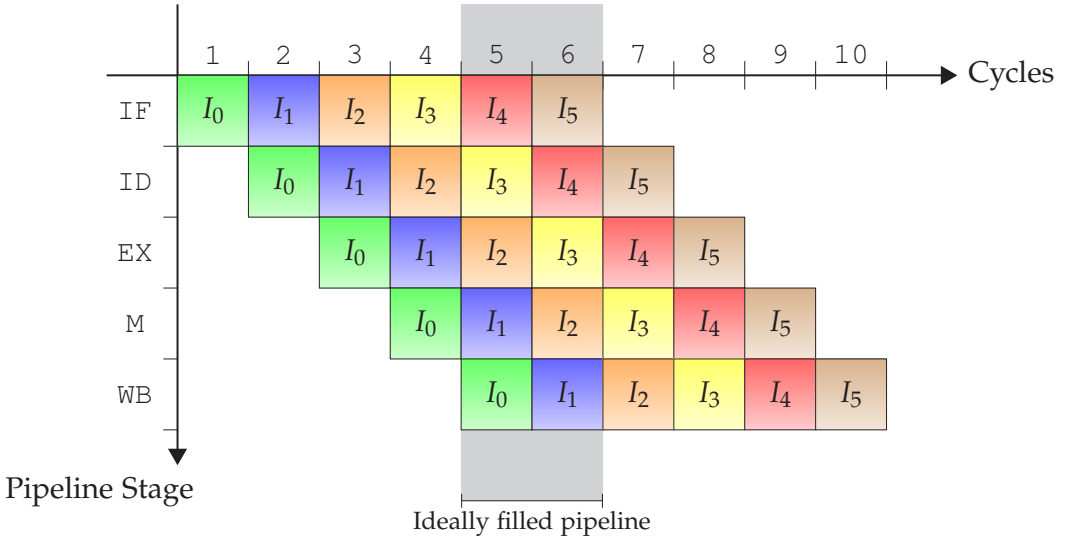
3.4 Processor Pipelines

3.4.1 Overview

Processor pipelines have been introduced in order to increase the computing performance of processors as already mentioned above. The idea behind pipelining in general is the overlapping of consecutive actions. In this context, the execution of a machine instruction is divided into parts (*pipeline stages*), i.e., an instruction has to pass through the stages according to specific rules. For example, the pipelined DLX from [MP00] consists of five stages:

- *Instruction Fetch (IF)*: Instruction sequences are retrieved from memory into a queue according to the program flow.
- *Instruction Decode (ID)*: Instructions are decoded for dispatching to the appropriate execution unit.

Figure 3.6 – Ideally pipelined execution on the DLX



- *Execute (EX)*: Instructions are actually executed on the execution unit.
- *Memory (M)*: Memory accesses are performed here.
- *Write Back (WB)*: Computed results are written back to memory or registers.

All instructions pass through all the mentioned stages in descending order. Figure 3.6 shows the execution of an ideally pipelined instruction sequence. In cycle 1, instruction I_0 has entered the processor pipeline at the IF stage. In each processor cycle, each instruction can advance to its next pipeline stage so that another instruction can enter the pipeline. By the end, all pipeline stages are occupied by different instruction at cycle 5 and the processor pipeline is ideally filled. Now, in each cycle, an instruction is finished and can leave the processor pipeline which is called instruction retirement. In this example, the instruction sequence ends with I_5, so the pipeline runs empty until cycle 10 when I_5 retires. Assuming that no instruction overlapping would happen, the execution would end in cycle 50. So the performance gain in this example is a factor of five which illustrates the theoretical potential of pipelining the execution of instructions within the processor.

3.4.2 Pipeline Hazards

The illustration of a pipelined execution as shown in Figure 3.6 on the facing page is idealized. In practice, such a perfectly overlapped instruction execution often cannot be achieved because of side conditions. These could be induced by the executed program itself, e.g., by dependencies between the instructions, or by the processor pipeline. Such situations where the pipeline stalls, are called *pipeline hazards* and they can be grouped into three categories depending on the stall indication.

Structural Hazard

Structural hazards are a consequence of resource shortages among the units within the processor pipeline. For example, this could happen if an instruction fetch and a memory operation both want to occupy the memory bus. Some delay emerges either at the fetch or the memory unit and the result is a delay at one of these pipeline stages. Another example could be if there are more integer instructions in a sequence than there are integer units available in the processor. As long as all available units are occupied by some instruction, the instruction dispatch stalls.

Data Hazard

Depending on the program code to execute, there are more or fewer data dependencies between the instructions to execute. Such dependencies then lead to stalls in the pipeline that are needed to preserve the semantics of the program. There are three different kinds of such data dependencies which are depicted in a small assembly program in Listing 3.1 on the next page.

- *Read-after-write*
  There is an instruction accessing a register or memory cell that is written by a previous instruction. As long as the first instruction is not finished, the second instruction is stalled and the pipeline stage, too. The instructions in the lines one and two of Listing 3.1 on the following page show such a dependency.
- *Write-after-read*
  There is an instruction that wants to write to the same register as a previous instruction. In this case, the processor must ensure that the

Listing 3.1 – Pseudo assembler instruction sequence

```
1   r8  = r9  + r9
2   r10 = r8  - r11
3   r11 = r7  + r7
4   r6  = r12 - r12
5   r11 = r8  + r8
```

writing instruction does not complete before the reading one does. This is no problem for so-called in-order executing processors which execute the instructions in the order of their occurrence in the program. For out-of-order execution, there must be a mechanism to prevent the writing instruction to complete before the reading one does. Typically, this is implemented by the consequent usage of rename registers [HPG06]. The in-order and out-of-order properties of processor pipelines are explained in Section 3.4.3. The instructions in line two and three of Listing 3.1 show such a dependency.

- *Write-after-write*
  For multiple writing operations to the same register or memory cell, it must be ensured that the last instruction in program order is actually written back last to the target location. An example for such a dependency is shown by the instructions in line three and five of Listing 3.1.

Control Hazard

Control hazards occur if the target of a branch instruction is not directly known to the processor. This happens for computed branches where the target is not hard-coded but stored in a register or memory location and therefore computed by previous instructions. Analogously, branches can be conditional, i.e., a condition is associated with a branch instruction and the result of their evaluation is stored in special control flags of the processor. Depending on the control flag's value, the branch is taken or not. The lack of such condition or target information might stall the fetch unit so that the pipeline runs empty.

3.4.3 Performance Improving Features

In addition to the general idea behind instruction pipelining there are a lot of features introduced for further performance improvements by avoiding pipeline stalls.

Prefetching

In general, prefetching represents the heuristic load of data before its demand gets evident. This technique is used within the memory hierarchy where not only the requested data are retrieved from the next hierarchy level, but a whole cache line. Another example for prefetching can be found in the fetch units of processors. There is a so-called instruction queue which temporarily stores fetched instructions until they are dispatched to an execution unit. This bypasses stalls caused during instruction fetch, by dependencies, etc. For the determination of worst-case execution time bounds, a prefetching behavior must be taken into account as it influences the global timing.

Branch Folding

To avoid control hazards in the pipeline, the processor tries to fetch further instructions even after having fetched a branch instruction. For this, branches are already decoded when they are fetched and if the target of a branch is known, the branch instruction is removed from the instruction queue and the prefetching address is redirected to this target address. Such a removal of a branch instruction is called branch folding.

Branch Prediction

A branch is called conditional if the decision whether the branch is taken depends on some condition which is computed by previous instructions. If the branch is additionally not computed, i.e., the taken branch target is fixed, there are two possible successors: the fixed branch target address and the direct successor address of the branch (fall-through). The target address of a conditional and not computed branch can be predicted by the fetch unit in order to determine the address for the next fetch after the branch. When the condition becomes known to the processor the prediction resolves either to be correct or false. The latter is called a *misprediction*: the corresponding

instructions then have to be removed from the instruction queue and the pipeline stalls until the instructions at the correct target address are fetched.

There are two kinds of branch prediction:

- *Static branch prediction* does not change the prediction direction during the execution of a program and is based on known facts. For example, for loops, the backwards branch to the loop header is more likely to be taken as there usually is more than one iteration within a loop. Some instruction set architectures (like the PowerPC) support a static branch prediction by a bit in the machine code of the branch instruction. By this, the compiler can give the processor a "hint" about the likeliness of the branch directions. Due to the static nature of this method, the branch prediction is not sensitive to the actual executed program and yields misprediction rates not less than 30 % – 40 % (according to [HPG06]).
- *Dynamic branch prediction* is able to compute the predicted direction of a branch during the programs execution. Therefore the prediction is sensitive to the actual program and can reduce the misprediction rate down to about 2 % (according to [HPG06]). This method can be implemented by caching the actual branch direction (taken or not taken) in so-called *branch history tables*. The computation of the branch prediction is then based on the history of the executed branches of the program.

Delay Slots

Delay slots are another possibility to avoid pipeline stalls due to dynamic branches. Depending on the concrete architecture, a specific number of instructions directly following a branch are called the delay slots of that branch. These instructions are even executed if the branch is taken. The idea behind this technique is that the processor has computed the branch target before the instructions from the delay slots have finished their execution so that execution can seamlessly continue.

Forwarding/Shortcuts

To avoid read-after-write hazards mentioned above, fast data forwarding has been introduced. The idea is to forward the result of an operation from one stage in the pipeline to any other stage that holds an instruction depending

on that result. By this, the depending stage does not need to be stalled until the result is written back.

Shortcuts are hardware optimizations for special cases like the multiplication with a zero argument. Therefore, the execution time of such an instruction can depend on the concrete value of its arguments.

Superscalarity

Scalar processors can only issue zero or one instruction per clock cycle. Thus, they can only finish one instruction per clock cycle at most. In contrast to that, superscalar processors can issue more than one instruction per clock cycle. This increases the maximal number of instructions that can be finished within one clock cycle which again increases the performance of the processor. For example, the Freescale PowerPC 755 can issue/finish up to two instructions per clock cycle.

Properties like the issue structure (static vs. dynamic), the hazard detection (hardware vs. software), the scheduling strategy and more are used to further categorize superscalar processors. Hennessy describes further Details about that [HPG06].

Out-of-order Execution

Out-of-order execution has been introduced to overcome structural and data hazards. The general idea is that the execution of an instruction can be started and/or finished earlier than the execution of its predecessor instructions. This means that the issue step is still in program order but for example a long running memory instruction does not block the whole processor pipeline. Instead, consecutive independent instructions can already start their execution on other execution units so that only the memory unit is stalled. Completion time of an instruction and its retirement have to be distinguished. The completion time is when their execution has been finished and the retirement represents the time when the instruction has written back its results and leaves the processor pipeline. In order to keep the programs semantics, the retirement time of an instruction sequence still need to follow the program imposed order which is called in-order retirement.

Speculative Execution

As branch prediction tries to fetch instructions over branch boundaries, the idea behind speculative execution is to actually even execute such instructions which are then called speculative instructions or speculatively executed instructions. Such a speculative execution implies that the processor must be able to roll back all results of speculative instructions. This is achieved by the usage of so-called shadow registers which are a temporal storage in order to cache the results of register write operations. For memory accesses the instruction can proceed until a memory transaction over the memory bus would be started where the instruction stalls. If the branch prediction has been resolved, the contents of the used shadow registers are just written back or transactions over the bus can be started. In the case of a misprediction, the allocated shadow registers are just freed again and all traces of the speculative instructions are removed from the pipeline.

Modern processors like the Freescale PowerPC 7448 can speculate more than one level, i.e., there can be multiple branch instructions in the fetch unit whose target is predicted in order to continue instruction fetch. As the PowerPC 7448 is able to execute up to 16 instructions in parallel, there would be pipeline stalls due to missing new instructions without such a deep speculation depth.

Store Gathering

Store gathering is a feature that reduces the memory bus utilization and is employed within processor cores with a rather complex load/store unit. If there is more than one outstanding store operation to the same cache line and all of these accesses result in transactions on the bus (due to cache misses), the processor tries to combine the accesses to larger ones. For example, two word stores would be combined to a double-word store on the bus. This saves memory bus bandwidth and therefore increases the overall processor performance because other accesses waiting for such stores can be processed faster.

Memory Protection

Memory protection is not a performance increasing feature but often used and needed in multi-processor systems. In such systems, each process assumes

that it runs in its own address space solely. This is implemented in the processes by using virtual addresses. The underlying operating system then needs to translate each virtual address into the actual physical correspondent. The process specific mapping is stored in memory in the so-called page table where each entry represents such a mapping. If there is a process switch, the page table entry for the new process is loaded into memory. To speedup this process, modern processors cache recent accessed page table entries. Such caches are called *Translation Lookaside Buffers (TLB)*.

Depending on the executing processor features, different attributes can be associated with an entry in the page table. For example, a page can be annotated as "guarded" so that any speculation into such memory regions is deactivated.

System Configuration

Modern processors are highly configurable, i.e., they have features that can be deactivated. By this, the processor can be adjusted to the needs of the customer. In the area of embedded systems, the configuration often is done by the provider of the embedded operating system. For example, the following features[4] usually are configurable:

- cache activation,
- cache locking,
- branch prediction,
- power management,
- address translation,
- store gathering,
- branch folding,
- branch history table,
- bus operation modes and
- bus pipelining.

Using (or not) these features influences the timing behavior of the whole system as well as its predictability (cf. Chapter 9).

[4]The list is not meant to be exhaustive in any way.

3.5 Summary

This chapter gives an introduction to embedded systems in general alongside a short categorization within the historical development of computer systems. Characteristic properties are listed and shortly explained with a focus on safety-criticality and real-time, followed by a description of typical application areas.

After this more general overview of embedded systems, their underlying hardware architectures and employed processors together with their pipeline structures and performance enhancing features are presented.

4

Timing Analysis of Embedded Systems

"Technology doesn't save time, but it redistributes it."

(Helmar Nahr)

4.1 Overview

Why do we need to know the precise timing behavior of a system?

The answer is as simple as short. The correctness of safety-critical embedded real-time systems decides about human lifes. This implies the demand to ensure a correct behavior of such systems and in the presence of real-time constraints the timing behavior strongly influences the functional correctness. Software development standards like the *DO-178B/ED-12B*[1] [DO92] try to force the integration of quality assurance as well as system validation and verification into the software life cycle. Before the official launch of a safety-critical system, the manufacturer has to convince certification authorities about the system's development process, i.e., that they have fulfilled the requirements demanded by the standard. Among other things, this means to prove that a system fulfills its timing constraints by computing safe approximations on its execution time.

To determine such approximations, a sophisticated timing analysis of the whole system has to be performed. This is done on different levels:

- on the *system level* and
- on the *code level*.

System level timing analysis comprises the analysis of task scheduling. For this, the so-called *worst-case response time (WCRT)* of each task as well as the *worst-case end-to-end communication delays* have to be computed.

Definition 4.1 — Worst-Case Response Time:
The worst-case response time of a task is the longest possible time between the task activation and its completion.

It takes information about code level timing analysis, scheduling of the underlying operating system, bus arbitration, possible interrupts and their priorities into account. The goal is to show the schedulability of the task system, i.e., whether the employed scheduling algorithm computes a feasible schedule that satisfies all deadlines and dependencies of the system's task (as mentioned in Section 3.1.2).

The code level timing analysis deals with the non-preempted execution times of single tasks.

[1] Software Considerations in Airborne Systems and Equipment Certification

Definition 4.2 — Worst-Case Execution Time:
The Worst-Case Execution Time (WCET) of a task is the longest possible time the task's execution can take.

In general, the computation of the exact WCET is not possible for general programs because it can be reduced to the halting problem: As a non-terminating program (for example with an endless loop) has a WCET of ∞, the computation of the WCET would have to decide first whether the program terminates at all. For a restricted form of programming with limited recursion and loop bounds the termination could be guaranteed so that the WCET computation becomes feasible again. These restrictions are usually implemented within embedded systems so that this problem remains a theoretical one.

Remark:
For the sake of simplicity the term WCET is used as a synonym for a safe upper bound on the actual concrete worst-case execution time of a code snippet.

But there are a lot of practical limitations in the field of timing analysis of embedded systems. On the system level the scheduling strategies of some operating systems prevent the computation of precise time slots of the processes which renders schedulability analysis impossible. This problem is addressed with specific real-time operating systems (cf. Section 3.1.1) or a system implementation without an operating system layer. Code level timing analysis has to cope with complex processor architectures that are used nowadays (cf. Section 3.4). The usage of some modern features could increase the computational complexity dramatically so that the analysis itself is time-consuming.

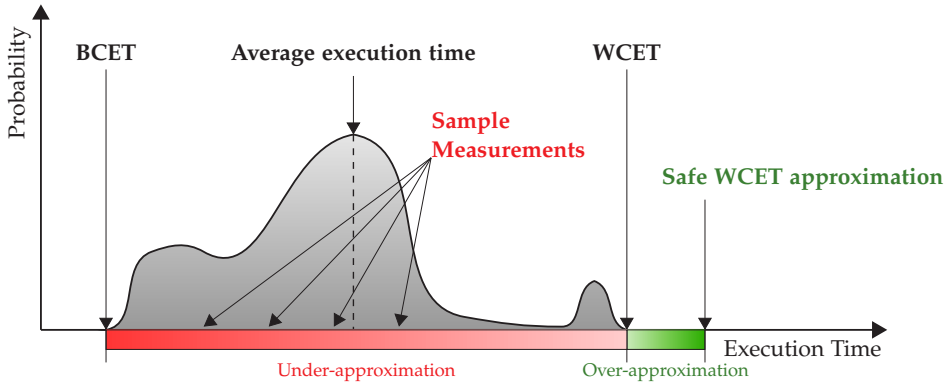
In this chapter, the different existing approaches to timing analysis of embedded systems are outlined. Then, one of the existing industrial tools for WCET determination – aiT – is detailed.

4.2 Classification of Approaches

As illustrated by the curve in Figure 4.1 on the next page[2], a task shows a certain variation of execution times between a shortest and longest possible execution time. Both corner cases are called *best case execution time (BCET)* and

[2] also shown in Chapter 1

Figure 4.1 – Execution time distribution



worst-case execution time (WCET) respectively. The most probable execution time is the average case.

There are different reasons for this runtime distribution:

- different input (e.g., other operating mode),
- different initial system state or
- some change in the environment, e.g., different sensor values that lead to more complex computations.

In order to derive safety guarantees, all possible executions have to be taken into account and the state space leading to that is typically large which renders simple end-to-end measurements infeasible.

In general, the existing approaches can be categorized into two classes: static and dynamic methods.

4.2.1 Static Methods

Static methods do not rely on the execution of the code on the real hardware or a simulator. Instead, the binary executable is analyzed independently of any input or initial state. By construction, all possible executions of the analyzed program are taken into account. Static methods share a common scheme of work flow:

1. Reconstruction of the binary's control flow

2. Static Analyses of the set of possible control-flow paths through the task
3. Combination of paths with an abstract model of the concrete hardware architecture

The employed abstractions reduce the state space to explore so that the whole timing analysis gets feasible even for complex processor architectures. Together with the nature of an input independent – static – analysis, this leads to an overestimation of the exact worst-case execution time of a task.

Despite its safety, static methods have the disadvantage of requiring a deep insight into the timing behavior of the system in order to minimize the above mentioned overestimations. The development of the hardware models often relies on processor manuals and other documentation which unfortunately often contains errors. Moreover, the behavior of undocumented features has to be reverse engineered.

4.2.2 Dynamic Methods

In contrast to their static counterparts, dynamic methods rely on the execution of the code on the real hardware or cycle-accurate simulators with given inputs. These measurements are either for the whole task execution (end-to-end) or only for snippets. For the latter case, the measured times are combined to derive execution times for the whole task. They lead to a minimal and maximal observable execution time as well as the measured runtime distribution.

One way to perform such measurements is to add additional instrumentation code that collects timestamps or CPU cycle counter information. Fully transparent measurement mechanisms are possible using logic analyzers, e.g., using the NEXUS standard [IEE00], or cycle-accurate hardware simulators.

An advantage of dynamic timing analysis methods is their fast setup time because it relies on execution or formal hardware design simulation in contrast to the development of a sophisticated hardware model. So, they do not rely on erroneous processor documentation.

In contrast to that, the main and disqualifying disadvantage of all dynamic methods is that their results might be unsafe without any evidence that all possible executions or at least the worst-case path have been measured. For measuring the worst-case path, its corresponding worst-case input has to be known as well as the worst-case initial state. For complex software in

combination with modern processors and their complex pipelining features, such a validation is hard to obtain. Neglecting the problem of knowing the worst-case input, another problem raises for simulators sometimes provided by the processor manufacturer. They claim their simulators to be cycle-accurate. But at least the underlying hardware model of Freescale's sim_G4plus simulator [Fre05b] for the MPC7447A/7448 (also known as G4) does not correctly represent its behavior [SP09].

Furthermore, logic analyzer traces can be used to validate the hardware models employed in the static methods (cf. Section 4.2.1).

4.3 aiT Worst-Case Execution Time Framework

As the last section has given an overview and categorization of existing timing analysis methods, this section now describes a particular static method in more detail.

4.3.1 Overview

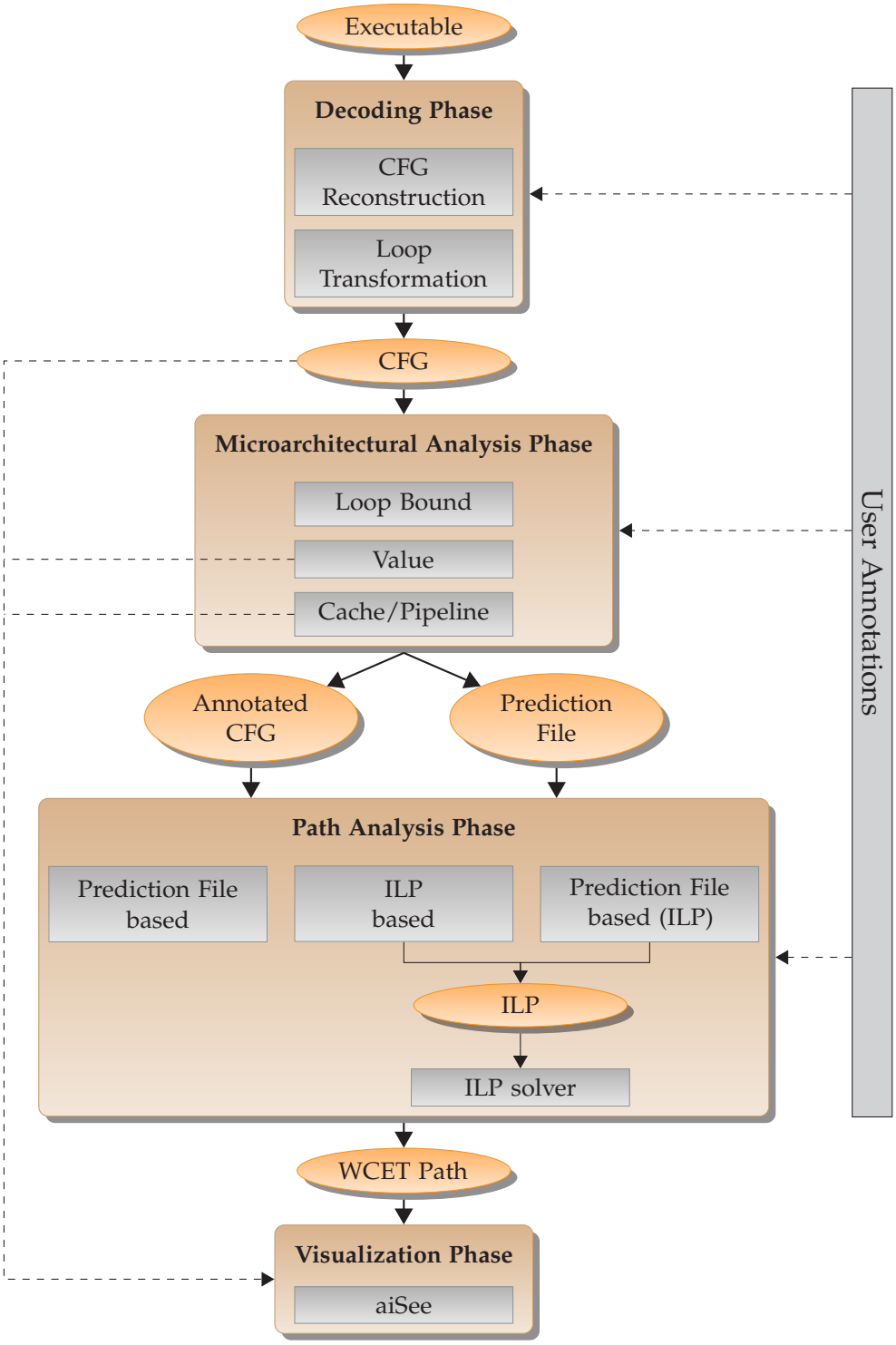
The aiT WCET analyzer is a generic framework for the determination of upper bounds on the worst-case execution time of tasks. In general, abstract interpretation [CC77, CC79, CC81, CC92a, CC92b, NNH99] gave the theoretical background for a couple of doctoral theses at the Saarland University [Fer97, Mar99, The03, The04]. Therefore, aiT is classified as a static timing analyzer with respect to the categorization of Section 4.2. Based on these research results, a first version of aiT has been designed by AbsInt Angewandte Informatik together with Saarland University[3] in the European project *Daedalus* [Dae]. Since then, aiT is industrially used for more than 10 years now and has constituted within the certification of safety-critical industrial systems, e.g. [SPH+05].

The analysis assumes the non-preempted execution of the analyzed task. The time bounds computed by aiT are both provably safe and precise [The04] and can be used as input for further system-level timing analysis [KWH+08].

The whole timing analysis is subdivided into phases where each one consists of different actions as depicted by Figure 4.2 on the facing page. The starting

[3] at the Compiler Design Lab headed by Prof. Dr. Reinhard Wilhelm

Figure 4.2 – Structure of the aiT framework



point is the binary executable. In the first phase – the decoding phase – the control-flow graph (CFG) of the binary is reconstructed and translated into an intermediate representation (CRL). In the second phase, different micro-architectural analyses are performed whose results are stored in the CRL graph. These results include the timing information at the basic block level. The path analysis phase then uses the results of the previous phase to compute the actual worst-case execution path together with its corresponding execution time – the worst-case execution time for the analyzed task. Finally, not only the computed timing bound is given to the user, but also the call and control-flow graph together with all computed results are visualized. The following sections now explain the different phases of aiT in more detail.

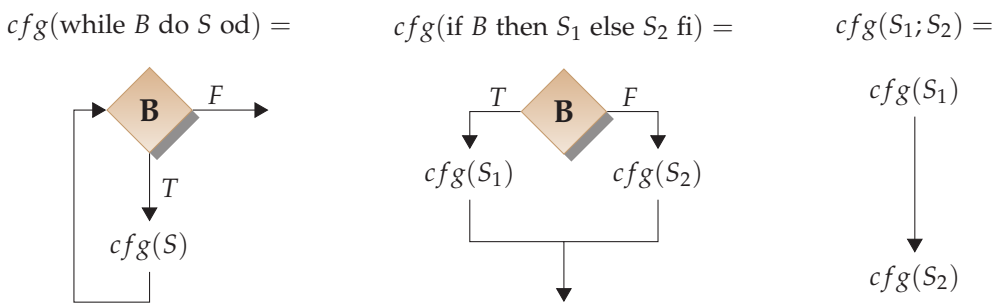
4.3.2 Decoding Phase

The input for the decoding phase is the fully linked binary executable that needs to be analyzed or that contains the task to be analyzed. There are reasons why the whole timing analysis works on the executable level: the analysis at higher levels of abstraction like the source or intermediate code level is not feasible since this code undergoes further transformations by the compiler and linker which influence the timing behavior. In other words, the only way to analyze the real timing behavior of a piece of software during execution is to analyze the actual binary that is loaded and executed on a particular system. Another advantage for the timing analysis are fixed parameters that can be extracted from the binary executable. For example, known code addresses simplify the analysis of the cache behavior.

Fundamental Program Representations

The result of the decoding phase is a machine-level representation of the binary program. Therefore the concept of machine operations needs to be defined. In the terminology of [WM95] a *micro-operation*, or *machine operation*, is an elementary operation that can be executed on the processor. The notion of machine *operation* has to be distinguished from the concept of machine *instructions*. In some architectures exhibiting intra-processor parallelism, especially in VLIW (cf. Section 3.3) architectures, multiple machine operations can be combined to form one machine instruction. The execution of all operations contained in the same instruction is started in parallel. In the following

Figure 4.3 – Control-flow graph for composed statements



(a) Loop (b) Conditional (c) Sequence

a summary of the most important concepts is given; Wilhelm [WM95] give more detailed explanations and additional literature references.

The control-flow graph of a procedure indicates which instructions can be executed one after the other. Whether this actually occurs during program execution may depend on conditions which in general cannot be evaluated at compilation time.

Definition 4.3 — Control-Flow Graph (CFG):
A control-flow graph is a graph $G = (V, E)$ with a set V of vertices and a set $E \subseteq V \times V$ of edges. The nodes are labeled with program constructs: $prog(v)$ denotes the label of a node $v \in V$. The edges which are written as $e = v_1 \rightarrow v_2$ $(v_1, v_2 \in V)$, are labeled with *edge labels* denoted by $lab(e)$ for an edge $e \in E$.

It is assumed that G has unique *start* and *end* nodes which are denoted by s_G and e_G respectively. In addition, there must exist a path (see below) through G from s_G to every node v. Likewise, a path must exist from every node v to e_G.

The subgraphs representing loops, conditional branches and sequential program flow are shown in Figure 4.3. Here, a subgraph $G' = (N', E')$ is inserted as follows: all incoming edges lead to $s_{G'}$, all outgoing edges to the successor node uniquely determined by the execution context. Nodes with more than one predecessor are called *joins* and nodes with more than one successor are called *forks*.

Definition 4.4 — Path:
A path π from node v_1 to node v_k in a directed graph $G = (V, E)$ is a

sequence of edges, beginning with a node $v_1 \in V$ and ending in $v_k \in V$ where $\pi = v_1 \rightarrow v_2, v_2 \rightarrow v_3, \ldots, v_{k-1} \rightarrow v_k$ and $v_i \rightarrow v_{i+1} \in E$ for $i = 1, \ldots, k-1$. The length of π is defined as the number of edges on π, i.e., $l(\pi) = k - 1$.

Definition 4.5 — Basic Block:
A basic block in a control-flow graph is a path of maximal length which has no joins except possibly at the beginning and no forks except possibly at the end.

If the first instruction of a basic block is executed, then in case of error-free execution (no hardware exceptions) all other operations of the basic block are executed as well.

CFG Reconstruction

The *CFG reconstruction* (cf. Figure 4.2 on page 69) action in the decoding phase reconstructs the CFG from a given binary executable. In fact, the reconstructed graph is a combined call and control-flow graph, i.e., the graph consists of two levels. The "outer" level is the call graph of all found routines in the analyzed code where the "inner" level represents the control-flow graph of each routine.

Remark:
The term CFG is used synonymous for this combined call and control-flow graph.

The resulting CFG may also contain nodes that are not directly mapped to program constructs. To ensure unique end nodes for each routine, special nodes need to be added to the control-flow graph. These special nodes are labeled with special symbols to distinguish them from program nodes. Another program construct that needs special care are routine calls and their returns. Due to the way how the micro-architectural analyses work, a special call and return node is added for each call in the program (cf. Section 4.3.3). To ensure safety of later analysis results, this reconstructed CFG must be safe, i.e., all possible paths that can occur during execution of the program must be represented. This is difficult for dynamically computed successors, e.g., indirect calls via function pointers or the implementation of high-level programming language constructs like switch tables. But often the possible successors can be deduced knowing which compiler has generated

the code because code generation is done in a fixed way. If the target(s) of a branch or call cannot be determined statically during CFG reconstruction, there are two possibilities: Emitting a decoding error forcing the user to supply information about all possible successors via user annotations (cf. Section 4.3.6) or assuming that all routines in the executable are possible successors. The latter leads to imprecise analysis results.

Technically, decoding is realized iteratively by two nested loops. The outer loop gathers reachable routines starting from the user-specified entry address and thereby computes the call graph of the read executable. It starts the inner loop that actually classifies byte streams as instructions. The result is the control-flow graph of the particular routine. Decoding of call instructions might detect new routines that haven't been decoded so far. They are communicated to the outer loop. Theiling [The03] explains details about the decoding algorithm.

Although the inner loop makes use of constant propagation, slicing and pattern matching, there still may be imprecise knowledge about parts of the control flow, e.g., because of function calls via function pointers whose content is not precisely known. In that case, the value analysis of the following analysis phase (cf. Section 4.3.3) is queried to analyze the current control-flow. With its results, the decoding phase is restarted to refine the control-flow. By this establishment of a feedback loop between decoding and value analysis, the precision of the control-flow reconstruction generally is increased.

The CFG reconstruction represents the CFG in a special intermediate language, the so-called *Control-Flow Representation Language (CRL)*. This language is hierarchically organized in operations, instructions, basic blocks and routines, i.e., basic blocks are enclosed within routines, instructions within basic blocks and analogously operations within instructions. Each entity in the CRL graph is annotated with so-called attributes. They contain information from the decoding phase like the processor resources used by an instruction[4], instruction classes and opcodes and similar data. The micro-architectural phase makes heavily use of this information.

Listing 4.1 on the next page shows an extract from a sample CRL file that illustrates the mentioned hierarchical composition of control-flow information.

[4] like source and destination registers

Listing 4.1 – Sample CRL file extract

```
1 ...
2 routine r74: address=0x10b0, instruction_set="ppc", name="min",
      section=".text", surface_address="0x10b0" {
3     block b75 (start) {
4         edge e78 (linear) -> b77;
5     }
6
7     block b76 (end);
8
9     block b77: address=0x10b0, instruction_set="ppc",
          surface_address="0x10b0" {
10        edge e80 (true) -> b124;
11
12        edge e79 (false, linear) -> b105;
13
14        instruction i81: address=0x10b0, bytes=?vb(4*[0x94 0x21 0
              xff 0xe0]), surface_address="0x10b0",
15            width=4 {
16            operation o82 "stwu r1, -32(r1)": arch="UISA",
                  assembly="stwu $, $($)", cat={ mem_write=1 }, dst
                  =4*[ ?v, ?v, "r1", "Mem" ], form="D", genname="
                  stwu", op_id=0x194000000, optype=3*[ "GPRegAll", "
                  signed", "GPRegBase" ], src=3*[ "r1", -32, "r1" ];
17        }
18 ...
```

Loop Transformation

After CFG reconstruction, the *Loop Transformation* converts each loop in the CFG into a tail-recursive routine. The original loop is replaced with a call to the generated loop. Similarly, each branch back to the start of the loop (a so-called "back-edge") is replaced by a recursive call. These calls do not correspond to code instructions of the executable and are only virtually added, i.e., in the control-flow graph structure.

By this, different iterations of the loop are transformed into consecutive recursive calls so that each iteration corresponds to exactly one execution of the loop routine. This transformation does not change the semantics of the program but enables a more precise analysis of loops because different iterations can be analyzed separately and thereby enables a configurable virtual inlining and unrolling [MAWF98]. Moreover, the loop transformation is only

done on the reconstructed control-flow graph, so the original executable is not changed at all.

4.3.3 Micro-architectural Analysis Phase

Program Analysis by Abstract Interpretation

This phase performs static analyses of the system architecture with respect to the execution of a given program. For this purpose, the precise effect of the program execution has to be formalized which is done by its semantics, a mathematical characterization of the program's possible behavior. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program. Here, the system is represented by a state, i.e., the contents of registers, memory cells, etc. Program execution is then defined as consecutive transformations starting with an initial state and leading to a specific end state. The execution of a single instruction is not mapped to a single transformation as the semantic effect of the instruction depends on the execution history as well as its control-flow predecessors, at least for modern processor pipelines.

Remark:

There are two different logical layers in such a program analysis that deeply incorporates the internal behavior of the underlying system. The outer layer is the control-flow graph consisting of program points that represent assembler instructions. And the inner layer represents the system state evolution caused by the execution of the current instruction. In principle, the inner layer is the essential part for the computation and the outer layer maps the sequence of system states back to the control-flow graph.

Definition 4.6 — Trace:

A *trace* is a sequence of system states connected by state transformations according to a given semantics.

In general, the concrete semantics of a program, i.e., all possible state transformations are not computable efficiently even under the restriction of terminating programs and a finite number of start states. Therefore approximations for the information provided by the concrete semantics of a program are needed. Such approximations will be incomplete but required to be safe, i.e., a program property that is claimed to be held by an analysis must be satisfied for all possible executions of that program.

Definition 4.7 — Abstract Interpretation:
Abstract interpretation is a framework that defines relationships between a concrete semantics and an abstract semantics that results from approximations.

Where the concrete semantics computes a set of system states, i.e., its working domain is the superset of all possible traces, the abstract semantics works on an abstract domain, leading to sets of abstract system states. Each abstract system state represents a set of concrete system states in the concrete semantics.

The goal of these approximations is to derive a computable semantic interpretation of the program. Depending on the employed abstraction, a loss of precision for the analyses results may be the consequence in favor of the decidability and tractability. For example, lets assume an analysis that abstracts from concrete values of program arithmetic and only stores the signs of expressions. This abstraction does not necessarily lead to a loss of precision: to compute the sign of a product, it suffices to know the signs of its operands. But in contrast to that, it is impossible to know the sign of a sum whose operands are respectively positive and negative. So, an abstraction always represents a tradeoff between the precision and efficiency in terms of computational complexity.

Definition 4.8 — Data-Flow Analysis:
Data-flow analysis is a program analysis technique where abstract values computed by data-flow equations are propagated between the nodes of a control-flow graph. The computation is repeated until the propagated values stabilize. This is called a fixed point iteration.

As a special application of abstract interpretation, correctness proofs for data-flow analysis can be given. In Section 4.3.3 there is an example illustrating the way how such a data-flow analysis works.

Four different data-flow analyses are employed in this phase of the aiT framework:

- the *loop analysis*,
- the *value analysis*,
- the *cache analysis* and
- the *pipeline analysis*.

Loop and value analysis are implemented within a single analyzer as well as the cache and pipeline analysis. Where the first two analyses might also be separated from each other, the combined cache and pipeline analysis has to be combined because the pipeline analysis models the control flow of instructions within the processor pipeline and therefore computes the precise instant of time when the cache is queried and its state is updated.

The analyzers are not solely manually programmed. Instead they are generated using the so-called *Program Analyzer Generator (PAG)*. PAG is a framework used to generate program analyzers from a concise analyses specifications [Mar99]. It combines the advantages of two worlds: abstract interpretation and data-flow analysis [Mar98]. Abstract interpretation enables the specification of provable correct analyses where the latter offers efficient implementation methods like the fixed point iteration. The generated code is ANSI-C compliant and efficient. As the analyzers can be specified in a concise and specifically developed high-level specification language, they can be easily and therefore timely adapted to other semantics like a different instruction set. By this, the loop, value and pipeline analyzers mentioned above can be adopted to new processor architectures in short time.

Loop Analysis

The *loop analysis* tries to statically determine iteration bounds for loops in a program by detecting code patterns that identify loop conditions as well as counter variable initializations and modifications. The actual loop bounds are computed by data-flow analysis on matching code pieces. For example, compiler generated code for loops could look like this: loop counter initialization just before the loop by a register load with a constant, a comparison of the register value with another constant at the beginning of the loop and an addition of this register with an intermediate value somewhere at the end of the loop body. Having identified these assembly statements by given compiler patterns, the loop analysis interprets the code and can compute intervals[5] for possible register contents. Using this information and the offset identified for loop counter incrementation, the actual loop bound is computed. Here, the analysis greatly benefits from the loop transformation done in the decoding phase as different loop iterations can be distinguished. The so-called VIVU (virtual inlining, virtual unrolling) approach invented by Martin and Alt [MAWF98] facilitates the analysis to gain precision by virtually unrolling

[5] actually more complex abstract values, cf. next section about value analysis

loop bodies. Otherwise, analysis results would be joined at the loop header because the control flow of both the last instruction before the loop and the last instruction of one iteration coalesce at that point.

A drawback of this code pattern-based approach is the need of adjustments for each compiler, sometimes even for different optimization levels in the same compiler. The quality, i.e., the ratio between the number of automatically computed loop bounds and the total number of loops in the program to analyze strongly depends on the precision of the patterns. This approach is limited with respect to the complexity of loops with multiple modifications of the loop counter where not all loop bounds in the program can be detected. Recent work [FMC+07] has enhanced the loop analysis by a method using intra-procedural data-flow analysis to derive invariants for the loop bounds. This method relies on the semantics of the machine instruction set solely, so there is no dependency on the compiler or code patterns. Combing both methods, bounds for most of the loops in typical real-world embedded applications can be automatically determined. In order to determine worst-case execution times, all other loop bounds need to be specified by the user via user annotations (cf. Section 4.3.6).

The computed loop bounds must not necessarily be precise values. In general, they are intervals whose precision can be disturbed by two situations: if an unknown value is involved in the loop counter's initialization, modification or comparison, the loop bound potentially cannot be determined at all. This might be the case if some register values are involved in the computation that are not set within the analyzed code, e.g., in system startup routines. Another reason for imprecise loop bounds are abstraction losses. For example, they can occur at control-flow joins where contradicting value information from different paths must be combined so that contents of machine registers might become unknown or imprecise. Martin [Mar99] gives further details on that topic.

Another recent work by Wegener [Weg11] tries to determine loop bounds even in the presence of imprecise data-flow information.

Value Analysis

Whereas the loop analysis determines intervals for loop bounds, the value analysis computes addresses of memory accesses as well as register contents. It operates not only on code snippets extracted by pattern matching. Instead,

the implementation is an abstract interpretation of the machine instruction semantics as documented in the instruction set architecture manuals of the particular processor family. As for the loop analysis, the results are safe interval approximations based on [CH78]. So both analysis techniques are similar in their principle of operation. The results represented by a domain of abstract values standing for certain sets of concrete values. A value in this abstract domain is represented by an interval of addresses enriched with additional relational information about the interval. *Modulo* information is an example for such relations and encodes the alignment of possible values in an interval of addresses. To be more precise, a modulo information m specifies that each possible element x within an interval has to fulfill an equation of the form $x \,\%\, p = m$, where p is a power of 2. More details of the modulo information have been described by Grewe [Gre08].

In embedded systems, data is often either loaded from configuration tables or memory-addressed peripheral devices (cf. Section 3.3.6). This means that the accessed addresses are often offsets to base addresses, either the base address of a configuration table or the start address of a memory-mapped device. So, knowing such addresses is more or less crucial for precise analysis results. Fortunately, they are often coded as constants for the executing tasks.

Moreover, properties of the different memory areas are important, e.g., read-only areas. Knowing that a section of the executable is read-only means that the value analysis can precisely determine the results of load operations accessing these sections as the section contents can be extracted during the decoding phase. Of course, this greatly improves the precision of dependent value computations.

To further improve the precision of the results, the so-called *context separation technique* is used to distinguish analysis results for program points depending on their execution context. For example, the path through a routine might depend on one of its parameters. Not separating the execution context would lead to precision loss due to contradictory information from each callee. There are publications [AM95, AM97, Mar99] that give more information on the separation of the execution context.

A side product and consequence of the determination of memory access addresses and register contents is the detection of so-called *infeasible paths* in the control-flow graph.

Definition 4.9 — Infeasible Path:
A path in the control-flow graph that is not executed by any possible

execution of the program is called an infeasible path.

Whether a computed branch is taken or not depends on a condition value, e.g., the contents of a register. By computing the possible values of such a register, the analysis might prove that one[6] of the control-flow successors is never executed. This knowledge has advantages: whole paths through the analyzed program can be marked as not executed which reduces the number of control-flow joins. As those points potentially introduce abstraction losses (due to contradictory value information) the precision can be increased in general. Another advantage is the reduction of possible execution paths through the program. Depending on the program, the following pipeline analysis can greatly benefit from this reduction in terms of its runtime and space consumption as it does not need to simulate those paths.

Concerning the precision of the analysis results, the value analysis underlies the same restrictions as the loop analysis, i.e., the existence of unknown register contents written during system startup and general abstraction losses. In general, value analysis' results are more precise if the analyzed task allocates memory statically. Dynamic allocation usually leads to unpredictable memory accesses because the allocating operations (like `malloc/new` in C/C++) typically do not give any guarantee about the particular returned address. Current research by Herter and Reineke [HR09] tries to overcome this limitation. Fortunately, dynamic memory allocation is only rarely used in hard real-time systems because developers otherwise need to cope with sufficiency, garbage collection, fragmentation and timeliness questions.

The results of the loop and value analysis are annotated to the program points in the control-flow graph as illustrated in Figure 4.2 on page 69. The following cache/pipeline analysis heavily uses and relies on that information.

Cache/Pipeline Analysis

The combined cache and pipeline analysis represents an abstract interpretation of the program's execution on the underlying system architecture. The execution of a program is simulated by feeding instruction sequences from a control-flow graph to the timing model which computes the processor state changes at cycle granularity and keeps track of the elapsing clock cycles. The cache analysis presented by [Fer97, FW98, FMWA99, FW99] is incorporated into the pipeline analysis. At each point where the actual hardware would

[6] or more in case of switch statements

query and update the contents of the cache(s), the abstract cache analysis is called, simulating a safe approximation on the cache effects. In addition to that, precise analysis of cache behavior still is an active field of research [GRG09, GR09, AMR10, GR10].

The underlying simulation model of the processor architecture has to be elaborated which is the reason why the pipeline analysis is the most complex in the aiT framework. The system state is approximated for computational complexity reasons which lead to precision losses as mentioned in Section 4.3.3. The correctness proofs according to the theory of abstract interpretation have been conducted by Thesing [The04]. Figure 4.4 on the next page exemplarily shows the correspondence of the control-flow graph with system states and their transformations as computed by the pipeline analysis.

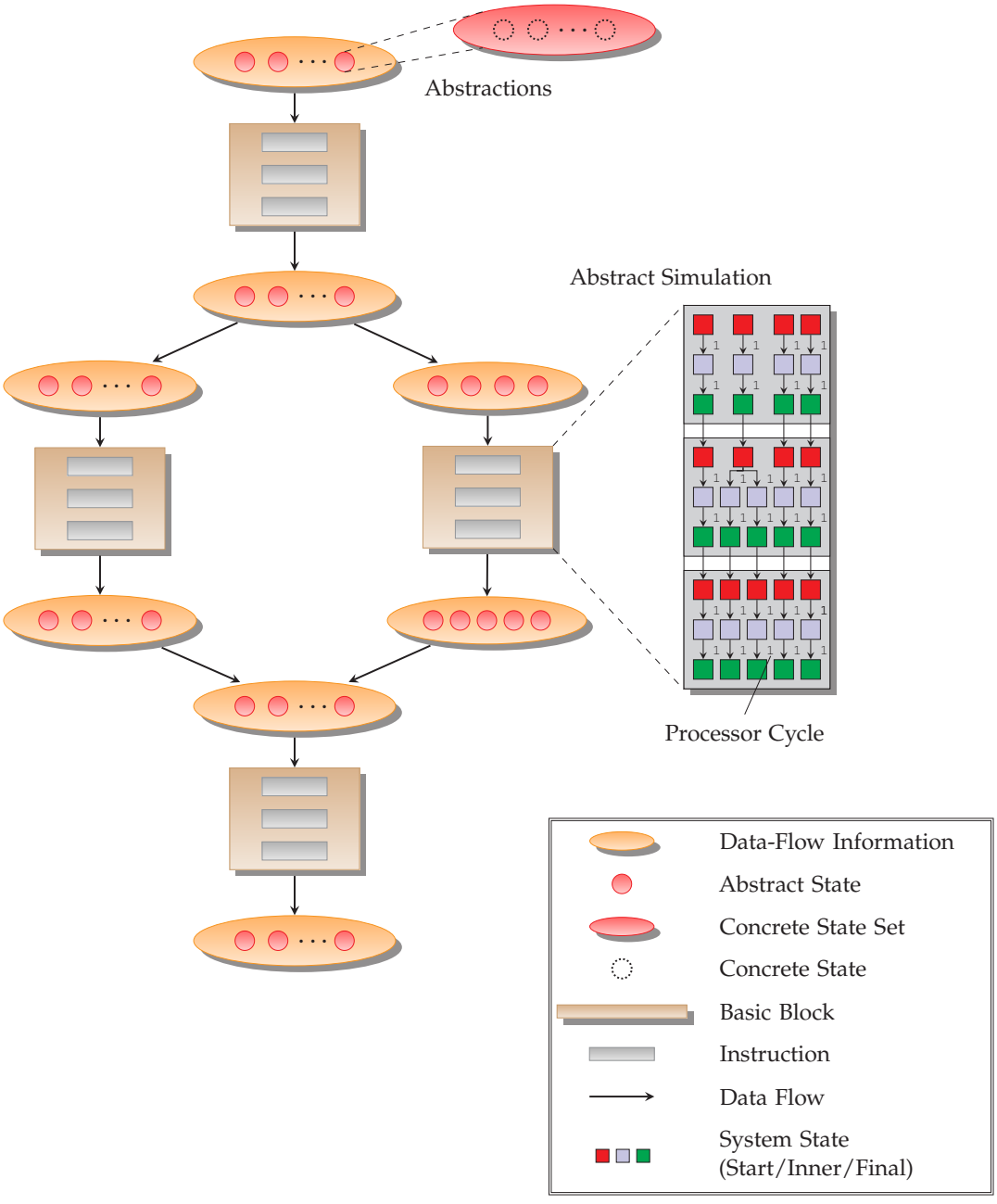
Each basic block has an incoming and an outgoing data-flow value, called *data-flow information (DFI)* each. The incoming DFI contains all possible abstract states before starting execution of the instructions in that particular basic block. The outgoing DFI analogously contains all possible final states after the execution. Each abstract state within a DFI is a result of the employed abstractions and represents a set of concrete system states. The abstract simulation computes the relation between the starting and final states as a sequence of processor cycle updates on each state in the incoming DFI of a basic block. These processor cycle updates and their corresponding state transformations are based on a timing model of the processor architecture and are explained in more detail in Section 6.2.

At program points where control flow is joining (cf. Figure 4.4 on the following page) the corresponding outgoing data-flow values are combined to the incoming value of the next basic block. This is a property of the underlying data-flow analysis equation (cf. [The04]) and implies potential abstraction losses if contradicting information has to be combined in a safely manner.

The result of the cache/pipeline analysis is the execution timing information, i.e., the number of processor clock cycles computed by the abstract simulation. To be more precise, the pipeline analysis generates an abstract system state graph – the *prediction graph*.

Definition 4.10 — Prediction Graph:
A *prediction graph* is a graph $G = (V, E)$ where each node $v \in V$ represents a system state computed by the abstract simulation of the pipeline analysis (cf. Section 4.3.3). Each edge $e \in E$ represents a system state transformation at processor core clock granularity.

Figure 4.4 – Pipeline Analysis of the aiT framework



Depending on the chosen path analysis variant (cf. Section 4.3.4), the prediction graph is generated with different levels of detail:

- only at basic block level (*basic block-level prediction graph*), i.e., the nodes represent the resulting system states after having simulated the execution of all instructions of a basic block or
- at instruction level (*instruction-level prediction graph*), i.e., the nodes represent the resulting system states after the retirement of each instruction whose execution has to be simulated.

This timing information is later used by the path analysis phase to determine the actual worst-case execution path.

Even with the powerful methodology of abstract interpretation, the pipeline analysis might be computationally complex and demanding especially for modern and complex processor architectures like the Freescale PowerPC 7448 [Fre05a]. Recent research activities therefore try to transform the computational model into a model for symbolic state traversal in order to benefit from the efficient state representation capabilities of binary decision diagrams. Please refer to [WW09] for details on this topic.

4.3.4 Path Analysis Phase

The path analysis phase uses the results of the combined cache/pipeline analysis to compute the worst-case path of the analyzed code with respect to the execution timing. So, the execution time of the computed worst-case path is the worst-case execution time for the program. Within the aiT framework, three different methods for computing this worst-case path are available as illustrated in Figure 4.2 on page 69. Each of this methods will be described in the next sections with its advantages and disadvantages.

ILP Based Path Analysis

This method has been introduced by Theiling [The03]. The idea is to generate an integer-linear program from the basic block-level prediction graph (cf. Section 4.3.3). They are combined into an objective function together with derived constraints on the control flow of the code.

The objective function has been described by Heckmann [Hec10]: Let $T(e, c)$ be the estimated WCET for edge e and context c as determined by the combined cache and pipeline analysis. Furthermore, let $C(e, c)$ be the execution count which indicates how often control passes along edge e in context c. If one knows for a specific run of the code the execution counts $C(e, c)$ for each edge e in each context c, then one can get an upper bound for the time of this run by taking the sum of $C(e, c) \cdot T(e, c)$ over all edge-context pairs (e, c). Thus, the task of obtaining a global WCET estimation can be solved by finding a feasible assignment of execution counts $C(e, c)$ to edge-context pairs that maximizes the following:

$$\sum C(e, c) \cdot T(e, c)$$

The value of this sum is then the desired global WCET estimate.

Constraints on the execution counts of basic blocks to each other then describe the control flow of the execution. For example, if a basic block b_1 has two successor blocks b_2 and b_3, then the number of executions of b_1 equals to the sum over the executions of b_2 and b_3 because the control either passes from b_1 to b_2 or to b_3. More details on the constraint system are given by Theiling [TFW00, The02]. Such constraints are automatically generated and improve the precision of the computed WCET by restricting the computation to paths in the control flow which are not infeasible, i.e., for which the micro-architectural analyses could not prove infeasibility.

The ILP based path analysis method is the oldest and has been used successfully in practice with precise results, for example for avionics software [TSH+03]. A drawback is the usage of the worst-case path through the pipeline states for each basic block. It therefore combines execution traces which might not represent an actual execution of the program. This potentially results in an overestimation.

Prediction File Based Path Analysis

To overcome potential overestimations of the purely ILP based path analysis method, the prediction file based path analysis has been developed [Mat06]. It does not rely on the execution time information on the basic block level. Instead, the instruction-level prediction graph (cf. Section 4.3.3) output of the pipeline analysis is used.

Such a prediction graph contains all possible system states that can be reached during execution of the analyzed code. Therefore the problem of computing the WCET path of a program can be reduced to the determination of the longest path in its prediction graph.

By basing on the pipeline state graphs, the method has knowledge about the relation between the end states of a basic block and the starting states of the successor block. This enables an overestimation reduction of the ILP based path analysis method. Another advantage is that the prediction file based method can be implemented efficiently. Assuming that the prediction graph is acyclic, its longest path can be computed in $\Theta(V + E)$ according to [CLRS01]. But that property also sets up a drawback of this method. In the presence of loops, the pipeline state graph might become cyclic depending on the analysis settings concerning virtual unrolling of loops [MAWF98]. In such a situation, the prediction file based method cannot be employed.

ILP Based On Prediction Files Path Analysis

There is another variant of the ILP based path analysis that operates on the instruction-level prediction graph analogously to the prediction file based method presented in Section 4.3.4. This method combines the ideas of both other path analysis approaches and has been developed by [Ste10]. Now, the longest path is not computed using graph algorithms directly. Instead, an ILP constraint system is generated from of the prediction graph.

By this construction, the advantages of both existing methods (cf. the two last sections) can be combined, i.e., the precision improvements as well as the possibility to incorporate loop constraints in the integer-linear program. The latter enables the handling of cyclic prediction file graphs.

A disadvantage is the size of the generated ILP: it might become large due to the structure of the prediction graph. For this reason, further implementation optimizations are currently developed.

ILP Solver

The integer-linear programs generated by two of the above mentioned three path analysis variants (cf. Section 4.3.4 and Section 4.3.4) can be solved using an arbitrary available ILP solver as sketched in Figure 4.2 on page 69.

The aiT worst-case execution time framework currently supports the following ILP solvers:

- *CPLEX*
  The IBM ILOG CPLEX optimizer is probably the most advanced and powerful ILP solver software available today [Int11]. The aiT framework supports usage of CPLEX as a solving module although the license for using this solver has to be provided externally.
- *clpsolve*
  This solver is an open-source implementation from the COIN-OR project [LH03] and is not as sophisticated as CPLEX, but available at no costs. For medium size programs the integer-linear program generated by the path analysis can be solved rather fast using clpsolve. Therefore it is a good alternative to CPLEX.

4.3.5 Visualization Phase

By contrast to all previously described phases the visualization phase does not directly compute any result that contributes to the worst-case execution time. This phase collects the results of all other phases so that they can be interactively explored.

The graphical representation is realized using the *graph description language (GDL)* [EB09]. GDL is a hierarchical language similar to CRL that can be read by the aiSee graph viewer [Abs11].

The following data can be visualized this way:

- Decoding phase results:
  The result of the binary decoding phase – the combined call and control-flow graph – is written in its GDL representation. In addition, the decoded machine instruction can be seen in a code listing view including cross references to the actual source code files (if available).
- Micro-architectural analyses results:
  All results of the loop-, value- and cache/pipeline analysis can be explored interactively. For the value analysis this means the presentation of incoming and outgoing data-flow values for each program point separately. The abstract simulation performed by the pipeline analysis can be viewed in basic block granularity which enables to track the system evolution during execution of the analyzed code.

- Path analysis phase results:
  The determined WCET path is highlighted within the CFG representation together with its costs, i.e., timing information for each basic block is provided.

The visualization feature of the aiT framework greatly supports the understanding of the timing behavior of a program. "Worst-case performance bottlenecks" can be identified and examined, i.e., often small changes in the source code can affect the precision of the timing bound [FH08, GCH11].

4.3.6 User Annotations

Due to the nature of the whole timing analysis framework, there is some information which has to be provided by the user because it cannot be extracted/derived from the input binary alone. Such information may be missing loop bounds, targets of unresolved computed calls, memory timings and properties, etc. In order to provide such information, the aiT framework can be supplied with different kinds of specifications and annotations in a specific language, called *AIS* [Hec10]. Depending on the input program and the underlying target architecture, these annotations are crucial for performing a WCET analysis at all or they improve the precision of the computed timing bounds.

In any way, the provided information replaces results of automatic analysis although warnings are emitted if the provided information is provably wrong. But in general, the correctness of the determined timing bound strongly depends on the correctness of the annotations.

Examples for such user annotations are:

- resolution of unresolved computed calls:
  If there are calls whose targets are computed via array references and the memory area storing the array contents is not marked as read-only by the compiler, the control-flow reconstruction might not be able to determine instruction successors precisely. Assuming all known routines as possible successors is practically impossible as the resulting precision degradation is not acceptable. Here, the user can mark a specific memory area as read-only and/or specifies further information about the structure of the array descriptor so that the call targets can be deduced automatically.

- missing loop bounds:
  If loop bound(s) cannot be determined automatically (cf. Section 4.3.3 on page 77), the user needs to provide them in order to make a timing bound determination possible at all.
- memory map specification:
  Depending on the underlying target architecture, more or less information about the used memory hierarchy must be provided. Most importantly, this includes the address ranges of all memory areas that are possibly accessed by the analyzed software. As such memory areas have configurable attributes in the hardware, WCET bound precision is increased in general, if such attributes are known to the analyzer. For example, it is important to specify whether a memory area is read-only or writable because the value analysis is allowed to propagate data contents of read-only areas during analysis. This information can lead to more precise value analysis results and, in consequence, might lower the computed WCET bound as more execution paths in the binary can be statically excluded. Other attributes indicate whether a particular region is served by a cache or specify cycle latencies of accesses.

For further details on the concrete syntax of AIS and its semantics, please refer to the specific tool manual of the particular timing analyzer like [Hec10].

4.4 Summary

This chapter starts with a general introduction to timing analysis for embedded systems. It explains the importance of this topic in the presence of safety-critical systems, i.e., the necessity of determining safe and precise upper bounds on the execution time of tasks in such systems.

Existing approaches to achieve this goal are categorized into static and dynamic methods with their different characteristic properties.

Afterwards, the aiT WCET analyzer framework is presented in detail because this tool has been used for the implementation and practical evaluation in the remainder of this thesis. Its tool design is illustrated together with a brief introduction of program analysis by abstract interpretation because one contribution of this work is the generation of aiT-compatible pipeline analyzers from the derived timing models.

5

Formal Hardware Specifications and Synthesis

"Tell me where a human is superior to a machine and I will build a machine that disproves your opinion."

(Alan Mathison Turing)

5.1 Overview

This chapter addresses the development of hardware components based on formal specifications. Until the late eighties, hardware has been designed using computer-aided design (CAD) systems that were based on so-called *netlist* languages. These languages define the hardware wires between single components of a chip. Their disadvantages are:

- They have a low level of abstraction which renders the design process complex.
- It is difficult to handle the growing complexity of modern architectures this way.
- Due to the complexity, the development process is slow.
- Support for automated optimizations concerning the wire locations is missing.

These disadvantages have been eliminated by the introduction of formal hardware specifications.

Definition 5.1 — Hardware Description Language (HDL):
A language that allows the specification of the functional, temporal and topological behavior of a piece of hardware is called a *hardware description language (HDL)*.

One of the most important properties of such languages is the explicit support for expressing concurrency and time as both are integral properties of electronics.

The advantages over manual net-list-based implementations are:

- the usage of high-level language constructs enabling to abstract away from the complexity of gates and flipflops,
- the possibility for simulating the resulting hardware model,
- the ability to cope with a magnitude of billions (10^9) of transistors,
- the possibility of an automated model verification and
- a tool-supported and therefore automated hardware synthesis process from a more abstract specification.

Despite the existence of a bulk of different and application-specific hardware specification languages, there are only four major HDLs that are widely used.

Verilog Verilog has been introduced in 1985 by the company "Gateway Design Automation" (GTA) and has been standardized in the IEEE-1364 standard [IEE95b]. In 1989, GTA has been acquired by "Cadence Design Systems". Verilog is widespread in the USA.

VHDL VHDL has been developed by IEEE on behalf of the U.S. Department of Defense in 1987 and represents the de facto standard for the specification of hardware designs in Europe. The following section gives a detailed description about its syntax and semantics.

SystemC SystemC has become an IEEE standard in 2005 [IEE05] and is a language for the specification of systems with both hardware and software components and is not a pure HDL like VHDL or Verilog. Syntactically, SystemC is realized as a C++ class library and augments the C++ language by macros and functions that provide features to express inter process communication, synchronization and parallelism. A disadvantage of this close relation to C++ is the syntactical overhead. On the other hand, the level of abstraction is higher than the corresponding level of VHDL or Verilog so that the simulation speed for SystemC specifications is faster by a factor of 10 according to Calazans et al. [CMH+03]. By this, simulations of larger and more complex software are possible compared to real HDLs like VHDL or Verilog.

SystemVerilog SystemVerilog represents a combination of a hardware description and a verification language. The hardware description part is a further development of Verilog and the verification part has been inspired by the verification language Vera [Syn12]. SystemVerilog has, similar to SystemC, inherited language concepts from C/C++.

In the following, VHDL is introduced in more detail as an example of a modern hardware description language.

5.2 VHDL

VHDL abbreviates *Very High Speed Integrated Circuit Hardware Description Language*. It is a hardware description language developed as an alternative to huge complex manuals describing the detailed behavior of a system. Its focus ranges from specifying circuits at transistor level to describing large system behaviors with high-level constructs. VHDL originally has been introduced in 1987 in the standard IEEE-1076-1987 [IEE87] and got augmented by a multi-valued logic data type within the standard IEEE-1164 [IEE93]. Besides revisions of IEEE-1076 in 1993, 2000 and 2002, some of the developed sub-standards are:

- *IEEE-1076.1* [IEE99a]
  VHDL-AMS for mixed analog and mixed signal circuit design
- *IEEE-1076.2* [IEE96]
  Mathematical packages with support for real and complex numbers
- *IEEE-1076.3* [IEE97]
  Synthesis packages that define the interpretation of IEEE-1164 (cf. above) for synthesis tools
- *IEEE-1076.4* [IEE95a]
  Additional support for detailed specification of pin-to-pin propagation delays and timing constraints

The newest revision of IEEE-1076 has been published in 2008 and is also known as VHDL 4.0. One of the major achievements here is a partial integration of the above sub-standards.

Although all language constructs within VHDL are suitable for a simulation, some are not usable for the direct synthesis of hardware components, e.g., all constructs that deal with timing. For this, another sub-standard – IEEE-1076.6 [IEE99b] – has been developed in order to define a *synthesizable subset* of VHDL. This standard defines a mapping of language constructs to hardware components, e.g., gates and flipflops, with the goal to unify the synthesis results of the different synthesis tools.

The following sections give an overview of the basic concepts and notations of VHDL alongside an introduction to its semantics. For a more thorough explanation consult [Ash08].

5.2.1 Domains and Abstraction Levels

VHDL offers three different domains, in which a system can be specified:

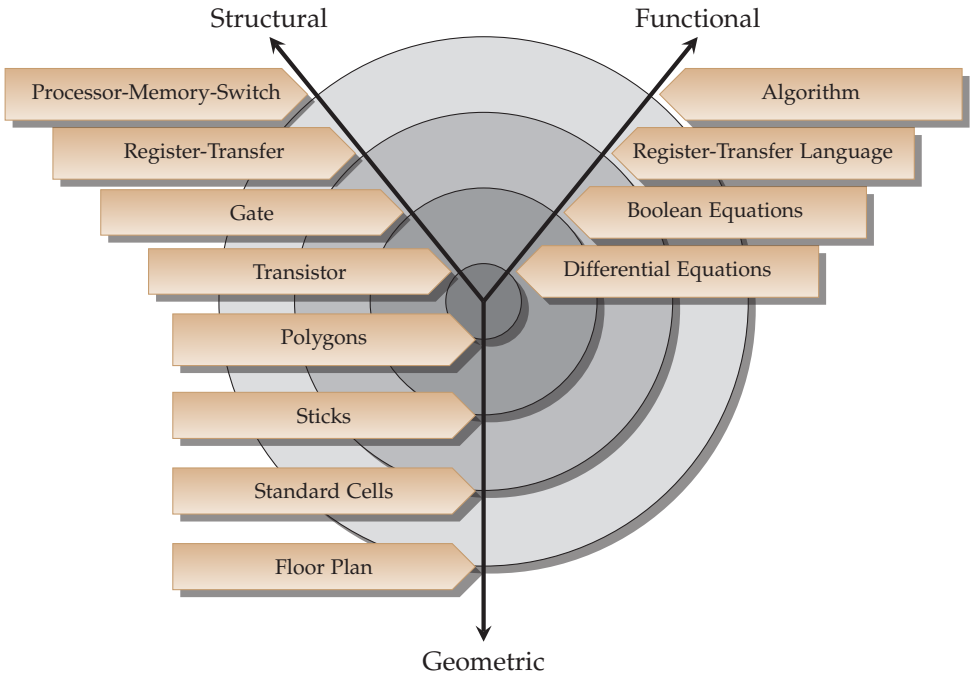
- *Functional domain*
  Here, the functional operations performed by the system are described.
- *Structural domain*
  This domain describes the structural composition of the system, i.e., the interconnection between the different components.
- *Geometric domain*
  The geometric domain specifies how the system is placed in physical space, i.e., wire routing etc.

This enables a wide range of different syntactical specification possibilities. Within each domain the designer can choose between four levels of abstraction starting from more overview-like descriptive text up to detailed low-level specifications suitable for automated synthesis/simulation and analysis. Depending on the user's needs, the hardware circuits can be described in different domains/abstraction levels. Even a mixture of different syntactical constructs within one model is possible as mentioned below. But when modeling the same hardware within disjoint specifications at different domains/abstraction levels, these models are not connected to each other. For example, if there is a structural as well as a behavioral description of the same hardware circuit within different VHDL models and one of the two models is changed, the other model is not modified automatically. The domains and abstraction levels only offer different syntactical specification techniques.

The three specification domains with their four orthogonal abstraction levels are illustrated in Figure 5.1 taken from Gajski [GK83]. The axes represent the three specification domains and the circles the abstraction levels where the outermost circle is the most abstracted level. This level is often called behavioral model because the system is explained as a whole. For example, in the functional domain this corresponds to pseudo-code that explains an algorithm. The geometric domain at this level could consist of a floor plan that shows the arrangement of the components on the die.

The second abstraction level reveals more details. In the functional domain, there is the so-called *register-transfer level* that shows data paths and control

Figure 5.1 – VHDL domains and abstraction levels (Y-Chart from Gajski [GK83])



sections.[1] The geometric domain in this abstraction level addresses transformation units and registers within a floor plan, i.e., augments the previous abstraction level.

In the second-most detailed level the structural domain shows the interconnection of gates, the functional domain is already at the level of Boolean equations representing the hardware logic and there are virtual grid notations in the geometric domain.

The fourth abstraction level is the most detailed one. In the structural domain, the individual transistors are shown. The functional domain derives differential equations that enhance the Boolean equations from the previous level by the relation between voltage and current in the circuit. In the geometric domain, polygons for layout masks of an integrated circuit are shown, which are used to describe the different layers of a circuit board.

Often models are not specified using only one abstraction level or domain. Typically, the topmost entity is a more structural part of the design and de-

[1] An example for such VHDL descriptions can be found in Listing 5.1 on page 98.

scribes how the hardware is composed of different sub-modules. Depending on the size and complexity of the complete model, the sub-modules again consists of other sub-modules. At some deeper level in this hierarchy, the modules then contain the actual functional specification. So, a hardware model often consists of a mixture of different abstraction levels and domains, i.e., structural as well as behavioral parts. Despite this enormous flexibility of the different abstraction levels, it should be noted that most system designers do not need to model at the third or fourth level. The hardware synthesis process supports an automated translation from the higher abstraction levels. That is the reason to concentrate on the register-transfer abstraction level for the purpose of extracting timing models from a system specification. This level is already detailed enough to extract the concrete timing behavior. Moreover synthesis tools are not allowed to change the specified timing behavior, so the synthesized hardware fulfills the specified timing.

5.2.2 Basic Language Constructs

This section shows some basic language constructs of VHDL with respect to the register-transfer abstraction level (see above) in order to give an impression of typical specifications encountered in the area of embedded systems.

Declarative Constructs A VHDL design is hierarchically composed of different language constructs where the topmost is a so-called *entity* declaration. It represents the design's public interface, i.e., its incoming and outgoing signals (see below), which are also called ports. Typically, these are the connected pins of the hardware. The entity only has a declarative character, i.e., it does not directly contain executed statements. For this, the implementation of an entity is specified via an *architecture body* construct which consists of one or more *concurrent statements*. Concurrent here means that these statements run in parallel when executing or simulating the design (cf. Section 5.2.3).

The most fundamental concurrent statement is the so-called *process statement*, which is described separately below. Other types are:

- *block statement*
  Block statements can be used to group different concurrent statements.
- *concurrent procedure call statement*
  Using this construct, a procedure call can be placed at the level of the

concurrent statements. Otherwise, such calls are only allowed as a sequential statement (see below).

- *concurrent signal assignment statement*
  A signal assignment can be lifted to the concurrent statement level analogously to procedure calls (of the previous item).

- *component instantiation statement*
  A component is a construct for structural information, as it defines the interface of a sub-module within an entity. Using a component instantiation statement, a sub-module is mapped to concrete input and output signals.

- *generate statement*
  A generate statement encloses a sequence of concurrent statements and generates a specified number of copies of these sequence. Syntactically, this looks similar to a "for" loop in C.

In general, all above listed concurrent statement types can be reduced to process statements. Therefore, they only represent a convenient way to express certain patterns of processes. In principle, all concurrent statements of a design are executed in parallel. Partly, their execution depends on conditions like sensitivity lists of processes (cf. Section 5.2.3). Because of that, there are no global variables and each globally visible signal is only allowed to be driven by one concurrent statement.

Implementation (architecture body) and interface declaration (entity) of a design is separated because this enables the definition of multiple alternative implementations for the same component in different domains. For example, an entity may be associated with a behavioral implementation that abstractly describes the component's functionality. Additionally, there might be a structural description of the component that describes how it is composed of other subsystems.

Sequential Executive Constructs At the RTL abstraction level, the architecture body consists of one of more *process* statements. Each of them is composed of one or more *sequential statements* which are executed in order as in other conventional programming languages. A sequential statement can be an evaluation of expressions, a variable or signal assignment, conditional and/or repeated execution or a subprogram call. Subprograms may be

defined within the scope of a process and can be called by a sequential statement. There are two different kinds of subprograms: *procedures* or *functions*. The main difference between both is that a procedure has no return type in contrast to a function. Thus, a procedure call is a sequential statement on its own and a function call is part of an expression.

Storage Constructs There are three kinds of elements for storing values in a model: constants, variables and signals. Constants and variables are process- or subprogram-local constructs, i.e., their usage is only allowed within the bodies of processes or subprograms. Where a constant can be only written once at its declaration, a variable is modifiable by a corresponding assignment sequential statement. At first glance, signals and variables are identical storage elements. The difference between them is the point of time when corresponding assignments take effect. Assigning a value to a variable takes effect immediately, i.e., the next reference to this variable returns the newly assigned value whereas the assignment of a value to a signal is only *scheduled* to be the future value, i.e., the next reference returns the old value. Additionally, signals are used within interface declarations of entities as they are used for the communication between different components.

Example: 3-bit Counter Listing 5.1 on the following page shows the above specification of a simple 3-bit counter. The entity declaration of the component "counter" is shown in lines four to seven followed by its implementation (architecture body). The input ports are the signals `clk` and `rst` where the first one is the clock and the second one is the asynchronous reset signal that is used to reset the counter value. The counter is designed as a synchronous circuit, i.e., all computations except for the reset are synchronized on the transitions of the clock signal. The current value of the counter is provided by the output port `val` which is a 3-bit binary number.

The implementation is given in form of two *processes*, namely `increment` and `output`. During simulation, a process executes its code whenever one of the *signals* contained in its *sensitivity list* (`clk` and `rst` or `cnt` respectively) changes its value. The process `increment` either increments the counter value for each rising clock edge and `output` generates the new output signal of the counter value or it sets the counter value to `"000"` if the `reset` signal is activated.

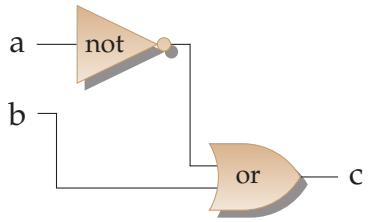
Listing 5.1 – 3-bit counter VHDL design

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Counter is
5      port (clk, rst : in bit;
6            val : out integer (2 downto 0));
7  end Counter;
8
9  architecture rtl of Counter is
10     signal cnt : integer (2 downto 0);
11
12 begin
13     increment: process (clk, rst)
14         begin
15             if (rst = '1') then
16                 cnt <= "000";
17             elsif (rising_edge (clk)) then
18                 cnt <= cnt + '1';
19             end if;
20         end process;
21
22     output: process (cnt)
23         begin
24             val <= cnt;
25         end process;
26 end;
```

Because of the special semantics of signal assignments, the statement "`cnt <= cnt + '1';`" in line 18 of Listing 5.1 schedules the next value of `cnt` to be `cnt` plus one, but the next reference `val <= cnt;` (line 24) schedules the next value of `val` to be the *current* value of `cnt`. These future values take effect as soon as all processes *suspend* their execution. The execution semantics are detailed further in Section 5.2.3.

Example: Implication Circuit VHDL also supports component-based circuit specifications. Figure 5.2 on the facing page shows such a hierarchical composition by a circuit for the logical implication. The result `c` of the circuit is computed by two sub-components, namely a logical-not gate (`not`) and a logical-or gate (`or`). The whole circuit then implements the implication by the formula $c = \neg a \vee b$. The mentioned decomposition into two sub-components can be seen in the VHDL specification of the implication circuit as shown

Figure 5.2 – Implication circuit schema



Listing 5.2 – Implication circuit VHDL design

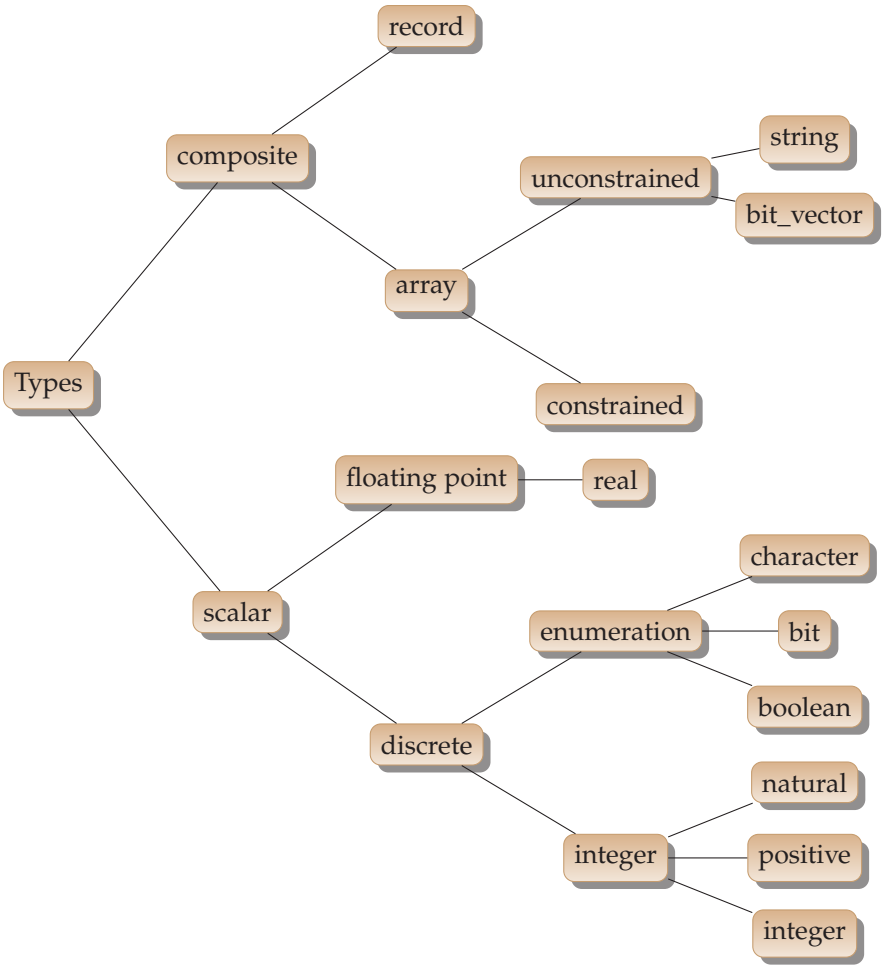
```
1 entity implies is port (a, b: in std_logic;
2                         c: out std_logic);
3 end entity;
4
5 architecture struct of implies is
6     signal int_neg: std_logic;
7 begin
8     not: entity invert
9          port map (a, int_neg);
10
11    or:  entity or2bit
12         port map (int_neg, b, c);
13 end;
```

in Listing 5.2 where the logical-not and -or gate are implemented by two sub-components.

Wait Statements A language construct to explicitly express time is the so-called *wait statement*. The process execution stops at this point and the process suspends for a specified amount of time or until one of its sensitive signals changes its value. This is a special property of the VHDL semantics (cf. Section 5.2.3). Wait statements are forbidden in the synthesizable subset (IEEE-1076.6). There, processes suspend implicitly after having executed all of their sequential statements which corresponds to a wait statement at the end of all sequential statements.

Type System The VHDL standard [IEE87] defines fundamental data types. Figure 5.3 on the following page sketches the most prominent ones alongside their dependencies. In addition, there are other data types which are

Figure 5.3 – VHDL type system



partly defined in complementing sub-standards like IEEE-1164 [IEE93] which augments VHDL by a multi-value logic data type.

Scalar data types are divided into discrete and floating point types. The discrete types represent different variants of integer numbers, where *natural* (≥ 0) and *positive* (> 0) are only sub-types of *integer*. Additionally, there are enumeration types: *character*, *bit* and *boolean*. On the floating point side, there is only the data type *real*.

Based on top of the scalar data types, there are two main composite types: *Arrays* that collect sequences of other data types (scalar or composite) or *records* that can be used to hierarchically group other data types. Two important predefined array types are the *string* and *bit_vector* types.

Of course, there are language constructs to construct custom data types or new sub-types of existing ones. But the existing types are typically sufficient to represent hardware circuits at the register-transfer level.

Ashenden [Ash08] gives more details about the VHDL syntax, its type system, and insights into the different abstraction levels and domains.

5.2.3 Semantics

The semantics of VHDL is not directly comparable with that of a conventional programming language like C or Pascal. In order to specify hardware behavior, it is necessary to have explicit language support for concurrency as it is present within hardware circuits. The simulation or execution of a VHDL model consists of two phases: an *initialization phase* at the beginning of the simulation and an iterative *actualization phase*. The state machine of the concrete simulation semantics described below is shown for illustration in Figure 5.4 on page 103.

Initialization Phase In the initialization phase (state *init* in Figure 5.4) all signals are assigned to their type-specific default values and the simulation time is set to zero. Then, all processes are executed once until they suspend, i.e., until they have reached a wait statement or the execution of all sequential statements has finished. During the process execution, signal assignments[2] schedule a transaction in the so-called *transaction list*. Each element in this

[2] cf. Section 5.2.2

list is a tuple $(t, s \leftarrow v)$ where t is the simulation time when the signal assignment takes effect and $s \leftarrow v$ is the actual signal assignment.

Actualization Phase The actualization phase starts when all processes have suspended their execution. Now, the transaction list is checked and all transactions scheduled for the current simulation time (state *sync* in Figure 5.4 on the facing page) take effect and signal values potentially change. If there are processes that are sensitive[3] to at least one of those changed signals, these processes are resumed and executed (state *exec* in Figure 5.4) until they suspend again. Then the actualization phase is restarted again (by going into state *sync* again). This process rerun step is called a *delta cycle* and it is important to state that the simulation time is not changed during a delta cycle. If all transactions for the current simulation time are done and all processes have suspended again, the simulation time is increased (in state *time* of Figure 5.4) to the value of the minimum over all timestamps t in the transaction list. Then again all transactions of the new current simulation time are executed and processes might be resumed. This shows that the actualization phase is an iterative process and executed until there are no more transactions in the transaction list (transition from state *sync* to state *end*). In synchronous systems where all actions are synchronized to a the clock signal, the simulation would never end because the actualization phase loop is never left. In the real hardware, this is okay because the system stays activated until it is shut down and during a simulation, one can set a time limit as an upper bound.

Thus, the semantics of VHDL can be seen as a two-level semantics: sequential process execution at its first, signal update and repeated process execution at its second level. An interesting effect of this semantics is that there usually exist delta cycles until system components reach a steady state. They are called *activation sequences*.

Example

Figure 5.5 on the next page shows a timing diagram of a simulation of the 3-bit VHDL counter specification from Listing 5.1 on page 98. Each row in the timing diagram represents a signal and shows its values during the simulation. The simulation progress is shown in the first row where each

[3] A changed signal is contained in their sensitivity list.

Figure 5.4 – VHDL simulation semantics state machine

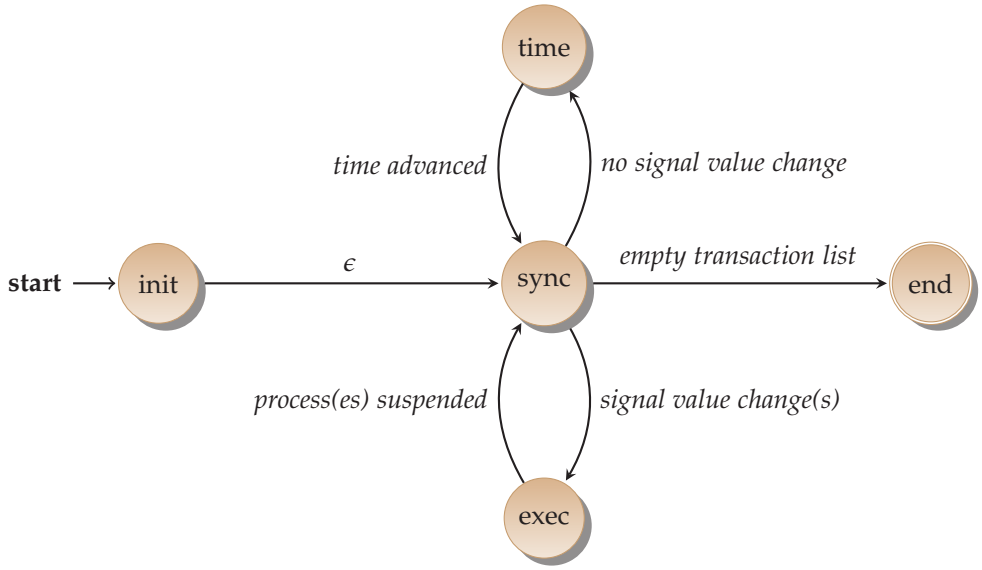
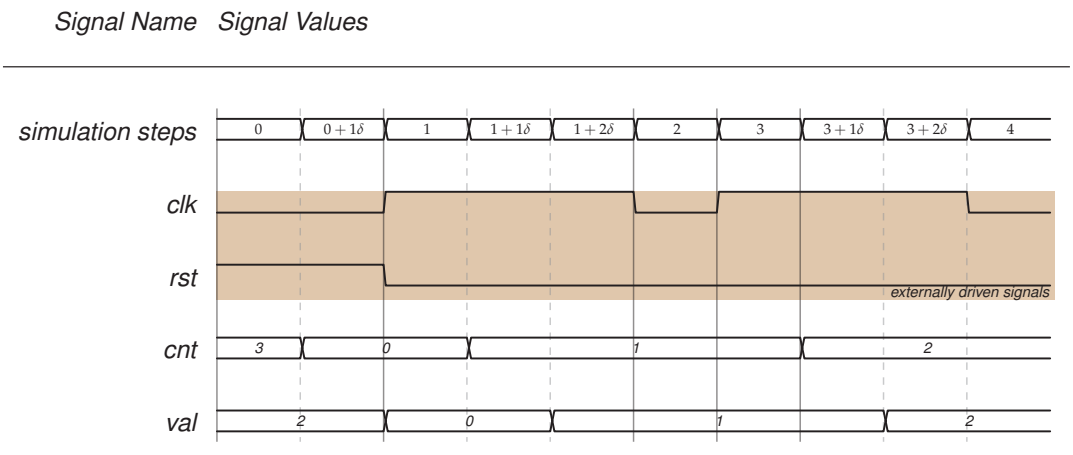


Figure 5.5 – Simulation timing diagram of 3-bit counter from Listing 5.1



number is one simulation step. For the sake of simplicity, the unit of time is in simulation steps so that the whole example is more or less independent of the concrete frequency of the clock signal. Here, the difference between a real simulation step (where time does elapse) and the described delta cycles between them is illustrated. According to the semantics of VHDL, one can see that there can be multiple process executions until some time elapses. This is shown in the delta cycles between the simulation steps 1–2 and 3–4 where the change of the `clock` signal triggers another execution of the process `increment` which itself causes a change of the `cnt` signal that triggers another execution of the `output` process. After that sequence of process executions, the signals become stable and simulation time can be advanced.

5.2.4 Analysis, Elaboration and Simulation

One of the major goals of modern hardware description languages is the possibility to simulate the specified hardware model. To reach this, a VHDL specification undergoes three stages:

- *Analysis*
- *Elaboration*
- *Simulation*

Analysis Stage In the analysis stage, the model is checked for syntactic and static semantic errors. The latter is possible because of VHDL's strong type system. This stage is similar to what is done in a compiler when analyzing the input program.

Elaboration Stage The elaboration stage modifies the VHDL code so that it is more suitable for synthesis and simulation. Hierarchies of data structures are eliminated which means that custom defined types are resolved so that only atomic data types defined by the standards are used. For hierarchically composed specifications, the elaboration results in a flat definition, i.e., components are instantiated at their declaration, record structures are collapsed and functions returning record types are transformed into procedures. In the end, there is only one large entity consisting of a number of processes and some locally defined signals. As a consequence of this, all globally visible

identifiers are renamed for unification, e.g., overloaded functions are resolved, variable/signal names unified, etc. In addition, all structural descriptions are transformed into wire definitions.

The following transformations to the VHDL code are required by the standard:

- *Transformation of subtypes*:
  If subtypes have been introduced in the design, their definitions are reformulated in terms of their atomic base types.

- *Resolving of overloaded functions*:
  Function overloading is resolved by making all function names unique. Moreover, all calls are adjusted accordingly.

- *Transformation of record-returning functions*:
  Functions that return a record data structure are transformed into procedures where the original return value is given a by-reference parameter. Similarly to the resolution of function overloading, all matching function calls have to be adjusted, as well.

- *Record structure collapsing*:
  All occurrences of record data structures are collapsed, i.e., the structures are split up into their elements recursively. Effectively, no record data structure is to be found after this conversion step.

- *Transformation of range attributes*:
  All range attributes are replaced by the particular constant values. This is possible because range attributes must be statically computable according to the VHDL standard [IEE99b].

- *Component instantiation and generate statements*:
  Component and generate statements are used to build up a hierarchical structure in the hardware design, i.e., a component declaration is more or less a reference to an entity definition (including corresponding architecture). During this step, all such references are replaced by their actual implementation.

- *Embedding of block statements*:
  The contents of all block statements are moved to the architecture body of the topmost entity and the corresponding block statements itself are deleted.

The VHDL standard does not require a specific order of application for the listed transformations.

Simulation Stage For the simulation stage, a model must be reducible to a collection of signals and processes. This is guaranteed by the elaboration process. Then, the model is simulated according to the semantics of VHDL as described in the previous section. This enables a sophisticated debugging and visualization flow for a hardware specification. Moreover, the elaborated model serves as input for a hardware synthesis tool as described in the next section.

5.3 Hardware Synthesis

Before the invention of modern hardware description languages, hardware synthesis had to be done by manually designing the hardware components in CAD systems. This process is complex and error-prone as it is conducted at a low level of abstraction. The goal of a tool-supported hardware synthesis is the specification of the circuits with descriptions at a higher abstraction level ignoring their detailed implementation in hardware logic. A synthesis tool then transforms the high-level description into a netlist based language which defines the connections between modules on a computer chip. Such netlists are the result of traditional CAD system usage, as well.

As mentioned above, the majority of the available synthesis tools accept VHDL specifications only at the register-transfer level and generate circuits from these specifications which are composed of standard cells for ASICs (see below) or LUTs for FPGAs (see below). The common basis for the accepted VHDL constructs is the synthesizable subset of VHDL (IEEE 1076.6 standard [IEE99b]). The VHDL specification is used to construct the described behavior in hardware automatically which can be an ASIC (*Application-specific integrated circuit*[4]) or FPGA[5] implementation. ASICs are composed of so-called standard cells: transistors and interconnect structures that either implement Boolean functions, e.g., AND, OR, XOR, XNOR, or provide storage functionality, e.g., flipflops or latches. The key components of FPGAs are called n-bit lookup tables (LUTs) implemented by multiplexers. They can encode any n-bit Boolean function.

[4] which means an application specific hardware component
[5] cf. Section 3.3.1

Listing 5.3 – 2-bit multiplexer VHDL design (taken from Heinkel [Hei00])

```
1 process ( A, B, SEL )
2 begin
3     if (SEL = '1') then
4         OUT <= A;
5     else
6         OUT <= B;
7     end if;
8 end process;
```

The mapping between high-level language constructs and their corresponding hardware implementation is done via pattern matching on the VHDL code. Therefore, synthesis results can differ between different tools, but the timing behavior in terms of clock cycles must not change to what is specified by the VHDL description. This enables the extraction of timing models from such VHDL specifications at the register-transfer level as it is demonstrated by the thesis. The timing model derivation as presented in the next chapter in combination with the generation of pipeline analyzers as described in Chapter 7 can be seen as a VHDL design synthesis with respect to an abstract semantics.

Although the semantics of process sensitivity lists is defined in the VHDL standard, synthesis tools often ignore them according to [Hei00]. In this case, the results of design simulation and execution of synthesized hardware may differ depending on the concrete inputs of the circuits. The simulation might potentially restart a process because one of the signals in its sensitivity list has been updated. This could lead to the mentioned result differences.

Listing 5.3 illustrates this potential difference at the example of a 2-bit multiplexer (taken from Heinkel [Hei00]). The model has two signals in its sensitivity list: `A` and `B`. Although the specified process reads the signal `SEL`, it has been (intentionally) not specified in the sensitivity list. Depending on the particular value of signal `SEL`, the values of `A` or `B` are assigned to the output signal `OUT` (cf. lines 3–6). During simulation of this design, a modification of signal `SEL` does not trigger a process execution so that the output signal does not change. By contrast, the generated multiplexer circuit appropriately drives the generated output signal value if the synthesis tool does not honor the sensitivity list. In general, simulation and generated hardware only show identical behavior if each process lists all signals that are read in its body.

For the remainder of this thesis, it is assumed that input VHDL designs are correct with respect to the sensitivity list specifications.

5.4 Summary

This chapter introduces formal hardware specification languages (HDL) in general. The purpose of their invention with their advantages over existing legacy methods for the specification of hardware circuits is described.

VHDL, one of the most prominent and widely used HDLs, is detailed as an example for a modern specification language with automated circuit synthesis and simulation support. Basic language constructs are depicted and followed by a detailed description of their two-level execution semantics. This is important because the generated pipeline analyzers in Chapter 7 represent (abstract) simulators for VHDL designs.

6

Derivation of Timing Models

"It is not enough time, we have, but it's too much time, we do not use."

(Lucius Annaeus Seneca)

6.1 Overview

In Chapter 4, it has been shown that all static timing analysis methods are based on a sophisticated model of underlying system architecture's timing behavior. Currently, these timing models are developed manually [The06], i.e., they are hand-crafted implementations by human experts. The development is based on the manuals provided by the manufacturers of the hardware platforms as well as a reengineering process for undocumented parts of the system (cf. Section 4.3.3). Both render the model development error-prone and time-consuming. It has been pointed out in Chapter 5 that modern hardware components are automatically synthesized from formal hardware descriptions like VHDL or Verilog. They determine the complete system's behavior – including timing information. So, it is highly desirable to extract the information needed for the timing model creation from such specifications. This would ensure that the resulting timing model is correct by construction and would speed up the creation time massively.

This chapter provides an approach to extract the timing behavior of a system from such formal hardware descriptions in order to overcome the limitations and disadvantages of the manual development of timing models. This thesis focuses on VHDL models nevertheless the approach in general can be applied to any hardware description at the register-transfer level. The only requirement is that the model is representative concerning the timing, i.e., it is detailed enough to contain cycle-accurate information.

Unfortunately, VHDL specifications contain a lot of details on processor state changes during program execution. If a timing model would be derived from a processor specification without any model transformation, the resource consumption of the subsequent pipeline analysis would be prohibitive. Therefore, the size of the model has to be reduced.

The remainder of this chapter is structured as follows: The next section introduces the term timing model and lists computation approaches at different levels of abstraction alongside their particular advantages and disadvantages. Nondeterminism and timing anomalies are detailed with their effects and implications on the safety of the computation approach employed within aiT. Section 6.3 shows how to analyze VHDL specifications using abstract interpretation by translating a hardware model into an equivalent sequential program. The goal is to derive safe model constraints. After that, Section 6.4 introduces the so-called *Timing Model Derivation Cycle* that defines a process

for the extraction of timing information from a formal hardware specification in VHDL. Section 6.5 describes common workflow for such a timing model derivation together with sample application scenarios. Additionally, a categorization of the presented steps into mandatory and optional one is presented.

6.2 Timing Models

As described in Chapter 4, one of the most important parts of the aiT framework is the combined cache and pipeline analysis which computes the timing information at the basic block level in the granularity of processor clock cycles. The computation of such timing information is possible in different ways:

Simulation of low-level hardware models is an approach where a hardware model is specified in a HDL at the gate level. This can be used to capture the elapsed execution time during simulation. Unfortunately such models contain too much information which is not necessary for the determination of the timing behavior, e.g., there are internal signals which are only useful for the functional behavior. In addition, the lack of hierarchy at the gate level prevents the invention of efficient abstractions which renders the whole approach to be computationally way too complex for industrial applications.

Another approach would be the **simulation of RTL-level hardware models** in VHDL or Verilog where the level of abstraction is higher than in gate-level models. But still, there is too much information in the description to be useful in a timing analysis directly, i.e., without any kind of transformation. Both for the gate- and RTL-level models their availability is a political issue that has to be solved, i.e., processor manufacturers often are afraid to provide their hardware specifications as they contain their intellectual property.

Effect semantics, i.e., specifying the semantic effect on the global system state at the instruction level together with combination rules would be a way to globally describe the timing effect of a program's execution. This approach cannot be realized for modern processors that can execute multiple instructions in parallel. Here, a more fine-grained model based on processor cycle-wise updates is needed. But for rather simple in-order processors, this approach is an option.

Designing a higher level model that represents the effects of the program execution manually is the method currently used. Such models are not designed at the gate level. Instead design components that more or less directly map to the functional units of the actual system are designed by hand. The pipeline analysis within the aiT framework as presented in Section 4.3 is the implementation of such RTL-level models with incorporated abstractions. It has been introduced by [The04]. This approach combines the needed level of detail – cycle-accurate timing information – from the above-mentioned RTL-level hardware models with the abstraction-implied analysis efficiency of the theory of abstract interpretation.

This chapter introduces a novel approach to extract the timing information needed to implement such pipeline analyzers by the transformation of an original and representative RTL-level specification of the system to analyze.

Definition 6.1 — Timing Model:
A *Timing Model* is a finite state automaton where each state represents:

- the state of the inner components of the processor, e.g., caches or queues,
- the values of timing relevant control signals of the system,
- the contents of the main memory during program execution (abstracted by the value analysis) and
- the state of the main chip set connecting the processor to peripheral devices.

The transitions between two states of the automaton is the effect of one single processor clock cycle.

6.2.1 Nondeterminism

For computational complexity reasons, state abstractions are introduced into timing models which over-approximate the set of all possible concrete states for an execution of a program (cf. Section 4.3.3). These abstractions lead to a nondeterministic automaton if the successor depends on a part of the state which is not precisely known. Another cause for such nondeterminism is the nature of static analysis, i.e., analyzing independently of any input which could result in the same unknown or imprecise state information.

Example (Nondeterminism). A timing model often needs to cope with at least two clock domains, namely the processor clock and the clock of the main system bus which is used for communication between the processor core and main memory or peripheral components. Despite the possibility of asynchronous communication, data transfers over the system bus are typically synchronized to rising clock edges. In this case, neither the core clock nor the system bus clock are modeled explicitly within the timing model. The reason is that a transition from one state to the next corresponds to a full processor clock cycle in a synchronous design. Despite this, within the model the information about the current offset between the core clock and the system clock needs to be incorporated. During the analysis, there might arise the situation that a bus transaction over the system bus has to be scheduled by the load/store unit of the processor and the clock offset information is not precise[1]. In this case, the timing model would be *nondeterministic* as it is not known precisely whether a transaction can be scheduled on the bus or not in this update cycle. Consequently, all possible successor states have to be considered so that one system state might have more than one successor state. In the above mentioned example case there are two states: one for the case that bus communication is possible in that particular clock cycle and one for no communication.

Definition 6.2 — State Split:
If a state transition in a timing model computes more than one successor state, this is called a *state split*.

Some typical reasons for states splits in the timing model for the Freescale PowerPC 755 are shown in Table 6.1 on the following page[2].

The cache and pipeline analysis (cf. Section 4.3.3) supports different computation modes influencing the number of state splits in the abstract simulation.

Definition 6.3 — Global Worst-Case:
If the pipeline analysis follows *all* possible successor states, i.e., it performs state splits whenever imprecise (or even unknown) information influences the state transition, this is called the *global worst-case computation*. The computed WCET bound/path in this mode is often called *global WCET* and *global worst-case path* respectively.

[1] Reasons are the merge of states with different offsets at control-flow joins.
[2] The table is not meant to list all possible reasons for state splits.

Table 6.1 – Freescale PowerPC 755 timing model state split types

Split Type	Description
Bank register hit/miss	SDRAM access into an already opened bank (hit) or not (miss).
Branch prediction	Imprecise knowledge of the resolution of static branch prediction.
Bus jitter	Offset between processor and main system bus clocks.
Cache hit/miss	Memory access that hits or misses the cache.
Multiple branch/call targets	There are multiple possible successors in the CFG, e.g., for switch statements.
Multiple memory areas	Imprecise addresses touched by memory accesses might access memory areas with different access latencies.
PCI/async memory jitter	Clock offset between the main system bus and the PCI controller.
Variant execution time	Execution time of an instruction might vary depending on the concrete values of its operands.
...	...

Definition 6.4 — Local Worst-case:
For some types of state splits, the pipeline analysis can weigh the transitions with local costs concerning the simulation time result. This means that the analysis can decide which possible successor leads to the locally slowest or fastest execution time. Whenever the global worst-case computation would perform a state split, the *local worst-case computation* only accounts for the locally slowest successor state. Analogously to the global worst-case, the computed WCET bound and path are called *local WCET* and *local worst-case path* respectively.

The difference between both computation modes is twofold. Certainly the global worst-case computation can be computationally much more complex

as each state split increases the search space of possible system states. This influences not only the pipeline analysis but all following steps in the aiT framework tool chain. For example, the generated prediction file contains much more states to be considered by the path analysis. In contrast to that the local worst-case computation is usually not as complex and much faster in computation time. But due to the presence of timing anomalies (cf. Section 6.2.2), local worst-case decisions might not lead to a safe approximation of the concrete worst-case execution time. Therefore this computation mode must be used carefully. More details on the implications of timing anomalies on the worst-case execution time computation are now discussed.

6.2.2 Timing Anomalies

In the area of WCET computation, the term timing anomaly intuitively is defined as:

Definition 6.5 — Timing Anomaly:
If there is an execution path where a locally faster execution (e.g. a cache hit) leads to the global worst-case execution, this path exhibits a timing anomaly.

Remark:
Within a WCET analysis, the occurrence of a timing anomaly can be observed by comparing the local with the global WCET computation: if computing a WCET bound for a task using the local worst-case computation leads to a lower time bound than using the global worst-case computation mode, this is called a timing anomaly. *In other words, the WCET path computed in global worst-case mode uses at least one local non-worst case decision.*

Remark:
In the presence of timing anomalies, a local worst-case WCET computation is not safe, i.e., the computed time bound does not necessarily represent a safe upper bound to the concrete WCET of the analyzed task.

Reasons for timing anomalies often are complex hardware features introduced for average case performance enhancements as for example:

- Cache replacement strategies:
  As Berg [Ber06] has shown, the FIFO, round-robin and PLRU cache replacement strategies are able to cause timing anomalies. Furthermore,

Thesing [The04] revealed that the pseudo round-robin replacement strategy of the Freescale ColdFire can trigger anomalies, as well.

- Complex pipeline features:
  Modern out-of-order pipelines have parallel execution units with different timing behavior as for example the Freescale PowerPC 755 (MPC755). For this processor, Schneider [Sch03] found the existence of timing anomalies.

But timing anomalies are not restricted to complex processors like the MPC755. Even processors with a rather simple pipeline architecture can exhibit timing anomalies as shown by Gebhard [Geb10]. This article gives a nice overview of the whole topic of timing anomalies as well as illustrating examples.

For some kind of timing anomalies, called *k-bounded timing anomalies*, their effect can be bounded by a constant k that represents the maximal possible timing difference that can be produced by the anomaly. For these, the local worst-case computation mode could be safely used together with adding the constant k to the computed time bound. Timing anomalies whose actual effect on the execution time cannot be bounded by any constant are called *domino effects*.

As a consequence of this, the computationally much cheaper local worst-case mode cannot be used without proving the complete absence or at least the k-boundedness of timing anomalies for safety reasons. Their safe identification still is an unsolved problem and active field of research [EPB+06, RWT+06, RS09, Geb10]. First work on the detection of possible anomalies has been conducted by Schlickling [Sch10]. There, the goal was to find state split types which cannot exhibit any anomaly so that one can use local worst-case decisions at least for those state split types.

6.3 Analyzing VHDL Models

In order to extract information about the timing behavior of a hardware component from its VHDL specification, it is necessary to analyze the hardware description. The idea is to apply proven program analysis techniques, namely abstract interpretation [CC77] and data-flow analysis [NNH99]. But these techniques have been developed to analyze programs written in languages with sequential semantics like C/C++. This means they cannot be applied

Table 6.2 – VHDL component to CRL mapping

CRL Element	VHDL Element
Routine	Process, Function, Procedure, Loop,
	Concurrent signal assignment
	Concurrent procedure call
Routine call	Function/procedure call
Instruction	Sequential statement

directly to a hardware language like VHDL with its two-level semantics (cf. Section 5.2.3). The solution is to transform an elaborated VHDL model so that it can be expressed as a sequential program rendering the application of abstract interpretation-based techniques to VHDL feasible. Based on that, [Sch13] introduces a general approach for the definition of static analyses on VHDL which are used within the timing model derivation explained in Section 6.4.

The next sections describe how a VHDL description can be expressed as a sequential program by a transformation into a semantically equivalent representation in the control-flow representation language (CRL). After that Section 6.3.3 summarizes the applicability of abstract interpretation on VHDL as introduced by [Sch13].

6.3.1 Mapping VHDL to CRL

In order to express a VHDL description using semantically equivalent CRL constructs, a mapping from VHDL to CRL components has to be defined. There is no design hierarchy left after the elaboration process, i.e., there are only processes and connections between them via definitions and usages of signals. A mapping from the basic language constructs to CRL has to be defined to show the connection between VHDL and CRL on the syntactic level. This mapping is listed in Table 6.2.

VHDL Processes, functions and procedures can be easily mapped to routines. Special CRL attributes are used to distinguish them later on. The statements inside of the processes are mapped to nodes forming the basic block structure in CRL, i.e., sequential statements are transformed into instructions

surrounded by a basic block (cf. Section 4.3.2). Concurrent signal assignment and concurrent procedure calls are a bit special. The former are converted into routines with a basic block containing a single instruction, namely the assignment. Concurrent procedure calls are handled similarly, but the single instruction here consists of the procedure call.

To improve the precision of the analysis [MAWF98], VHDL loops are extracted from their location and transformed into self-recursive routines. The original location of the loop is replaced by a call to the new loop routine. The same transformation is done in the decoding phase of the aiT framework (cf. Section 4.3.2).

The correspondence of function/procedure calls to routine calls as well as the mapping of sequential statements to instructions is rather intuitive.

Additionally, meta data from the original VHDL language constructs are stored at the corresponding CRL attributes[3]. For example, the sensitivity list of each VHDL process is stored in a special attribute `sensitivity_list` at the CRL routine.

For simplicity, VHDL case statements are transformed into "if-then-else" cascades as they are simpler to reflect in the basic block structure of CRL.

6.3.2 Semantic Level Reduction

Besides a syntactical mapping of VHDL language constructs to their correspondents in CRL, the concrete execution semantics of VHDL (cf. Section 5.2.3) needs to be reflected by the generated intermediate representation in CRL. Additional simulation code that implements the two-level VHDL semantics as described by the state automaton in Figure 5.4 on page 103 is required.

Variables in VHDL are process-local and processes run in parallel, and signal assignments only take effect after all processes have finished their execution. This semantics directly excludes side effects between the different VHDL processes. So, their execution can be serialized without changing the semantics of the whole model. If a process has at least one signal in its sensitivity list with updated value, it is executed again. This is iterated until a steady state is reached. Process execution is then represented by choosing an arbitrary execution order among the processes and iteratively executing

[3] arbitrary key–value pairs

them in this order. The simulation code for process execution (state *exec*) then consists of a routine `simul` with a loop whose body contains routine calls where each called routine represents one of the former VHDL processes. Additionally each such routine call for an original process is guarded with a conditional statement that evaluates the sensitivity list of the process. If at least one of the signals in the sensitivity list has changed, the call is taken.

The signal value synchronization (state *sync*) is encoded by an artificial program point at the end of the loop in routine `simul` where the scheduled signal assignments actually take effect. In addition to that, environmental signal assignments are represented by a routine call to a special routine `vhdl_environment` where external signal assignments can be modeled.

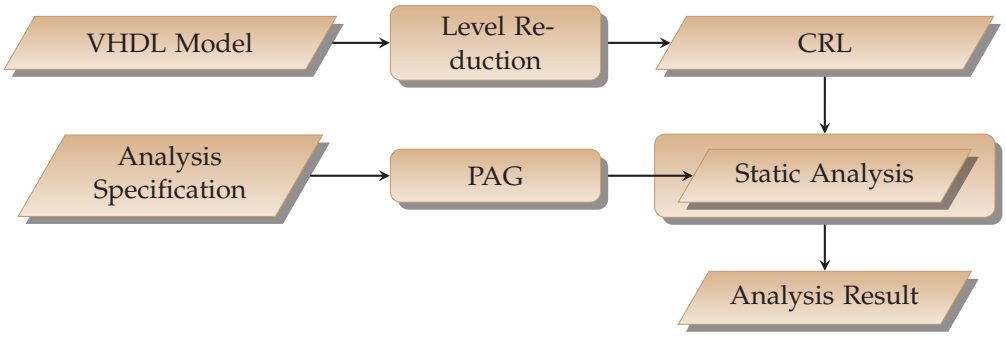
The clock cycles are represented by a routine `vhdl_clock` containing a loop whose body simulates a full clock cycle, i.e., the routine `simul` is called twice. Once for the simulation of a rising clock edge and once again for simulating a falling clock edge. More precisely, this is a routine that calls the code generated for the process execution state with an attribute showing whether there is a rising clock edge or a falling one. By this, an analysis can simulate the behavior of clock signals which are external signals and therefore not explicitly set in the VHDL model. Both, synchronous as well as asynchronous designs can be analyzed and multiple clock domains are supported, too.

This simulation code is called a VHDL analysis framework as it enables the static analysis of the model's execution.

6.3.3 Abstract Interpretation of VHDL

Abstract interpretation as mentioned in Section 4.3.3 enables the derivation of safety properties of a VHDL model. Performing abstract interpretation on an elaborated VHDL model surrounded by the analysis framework code as presented in the previous section now allows the static analysis of the hardware model's execution [SP07]. The needed data-flow analyzers can be generated from concise specifications using the *Program Analyzer Generator (PAG)* which enables the efficient usage of the whole power of abstract interpretation while minimizing the implementation efforts. Martin gives further details on the usage of PAG [Mar98, AM95]. Figure 6.1 on the next page illustrates this structure of the VHDL analysis framework.

Figure 6.1 – VHDL Analysis Framework – Structure



As the results of the analyzers hold for the whole simulation process of a model[4], they can be used for semantic preserving transformations on the model. Section 6.4.3 describes those transformations that are related to the derivation of timing models. Schlickling [Sch13] provides more details on the analyzability of VHDL models by static data-flow analysis.

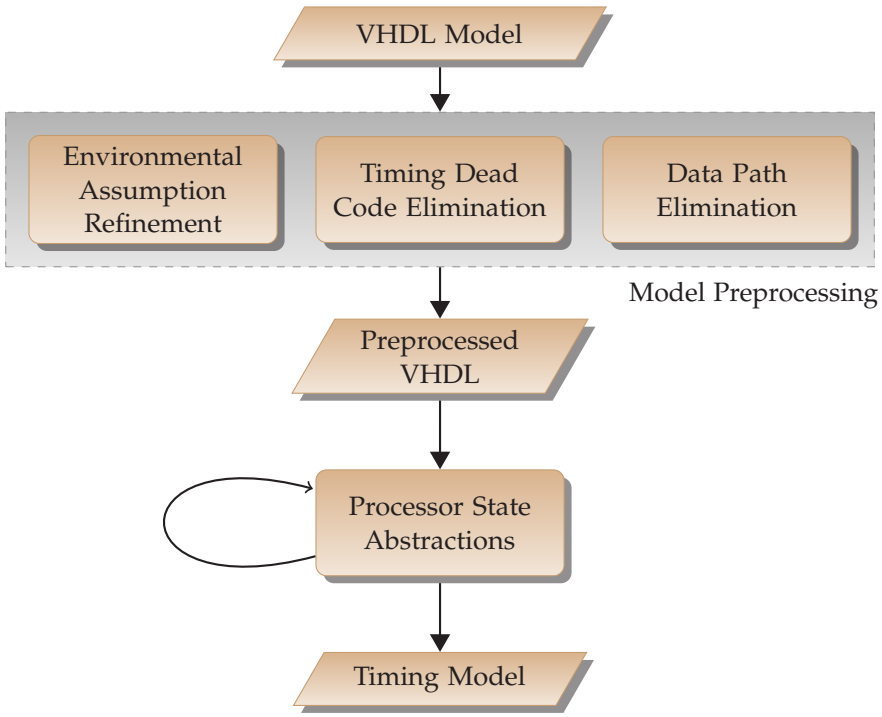
6.4 Semi-Automated Timing Model Derivation

This section describes the derivation process for timing models [PSM09, SP10] starting from the VHDL model and ending in the corresponding timing model which serves as input to the pipeline analysis generator (cf. Chapter 7). First, the theory and idea are presented. Afterwards, Section 6.5 shows the workflow of the derivation process.

Figure 6.2 on the facing page illustrates the overall process flow of the timing model derivation. The flow starts at the top with the incoming VHDL model in its CRL representation (cf. Section 6.3). As already mentioned in the overview of this chapter, the size of the model needs to be reduced. Most of this is done by the *Model Preprocessing* step as described in Section 6.4.1. Then the preprocessed model is the starting point for the search and application of *Processor State Abstractions* (cf. Section 6.4.2 on page 125). This iterative step results in the *Timing Model* which is used for the generation of a pipeline analysis fitting into the aiT framework (cf. Section 4.3). This generation process is detailed in Chapter 7.

[4] which means under all circumstances

Figure 6.2 – Timing Model Derivation Process – Overview



As the derived timing model is the basis for the determination of WCET guarantees for safety-critical systems, it has to be proven that the resulting model after the application of our size-reducing transformations and state abstractions correctly approximates the timing behavior of the examined system. An overview of this can be found in Chapter 8. The next sections detail about the preprocessing step as well as the processor state abstractions.

6.4.1 Model Preprocessing

The model preprocessing step is responsible for a major size reduction of the VHDL model which is essential as the resulting pipeline analysis would otherwise simply be not tractable in terms of computational complexity. A VHDL specification of a complete processor can be large, e.g., about 70 000 lines of code for the LEON2 processor [Gai05]. Simply transforming the specification into a timing model would render the resulting timing analysis infeasible, especially in terms of memory consumption.

Reasons for complexity problems are:

- Large parts of the VHDL model are only relevant for the functional behavior and do not affect timing properties.
- Although VHDL has been explicitly designed to allow simulation of the model's execution (cf. Section 5.2.4), WCET determination by abstract interpretation is performed without relying on input whereas a VHDL simulation is meant to be done with precisely known values for the external signals of the specified hardware component. Thus, the timing analysis has to examine all statically possible execution paths increasing the impact of a large underlying hardware model to both analysis time and space consumption. This second point can be seen as a consequence of the first point.

A solution is to remove parts of the model which do not affect the timing behavior. This is done in three steps:

- *Environmental Assumption-Based Refinement*,
- *Timing Dead Code Elimination* and
- *Data Path Removal*.

These steps are explained in the sequel.

Environmental Assumption-Based Model Refinement

The general idea behind this step is to remove parts of the VHDL model that are either unused or can be ignored for timing analysis. Parts of a VHDL model can be unused for different reasons.

A common cause is the gap between a specific application context within an embedded system and the high configurability of the employed hardware components. This means that certain features of modern processors can be deactivated by the user program or via external pins. The deactivation of such features renders some parts of the VHDL model unused which then can be safely removed from the model. Examples for such features are dynamic or static branch prediction, bus pipelining enabled or disabled, write-through or write-back cache write policies. But depending on the application scenario, even whole functional units of a processor can be unused. For example, if the analyzed program does not make use of floating point instructions, an existing floating point computation unit in the processor would be unused. Although a timing model is not generated for the analysis of a single program,

a class of safety-critical applications (e.g. flight guidance software) might be known not to make use of certain features like special instruction set modes, etc. Then, it makes sense to cut those unused features out of the model. The same argumentation holds for systems that support different types of memory like SDRAM, DDR, Flash or SRAM. For each type, there exists a dedicated memory controller on the system board. Not using some type of memory results in an unused memory controller for that type.

Another example why parts of a VHDL model can be ignored for timing analysis is the existence of asynchronous events in the system, e.g., hardware interrupts, DMA accesses, refreshes for dynamic memory (DDR), hardware exceptions or ECC errors for protected memory. Currently, a static timing analysis cannot fully deal with such events as the precise time of their occurrence is not known within the analysis. A hardware exception or interrupt brings the whole system to a state where no timing bound for the current task is needed. So for timing analysis, it can be safely assumed that those events do not happen. Some of these asynchronous events occur in a certain frequency. Their effect on the execution time can be incorporated, e.g., by adding penalties based on the computed WCET and the worst-case occurrence of the events or by statistical means (cf. Section 9.2.4). Certainly, this does not cope with their global effects on the timing behavior as they are only added after having computed a WCET bound while ignoring them during analysis.

In the VHDL code, such *configurable parts* of the hardware are guarded by control signals that indicate whether the particular hardware feature is enabled or disabled. Therefore, ignoring parts of a model is reflected by assuming the correspondent control signal to be deactivated.

Definition 6.6 — Model Assumption:
A constant value assignment for a specific VHDL signal is called a *model assumption*.

Certainly, assuming signal values to be constant is not restricted to a certain scope within the VHDL model, i.e., not only whole features like an L2 cache can be ignored. Instead, even small snippets everywhere in the model can be pruned by such assumptions. As a result, parts of the VHDL become unreachable, i.e., control never reaches these statements. So, dead-code elimination is employed to remove those parts from the specification.

Timing Dead Code Elimination

A similar optimization that reduces the size of the original VHDL model is the *Timing Dead Code Elimination* which means the removal of model parts that do not contribute to the timing behavior of the system. In contrast to the Environmental Assumption-based Model Refinement as presented in the previous section, specification code of components that are enabled and actually used/exploited by the analyzed program can be removed.

In order to restrict the model to "*timing-alive*" code, all instruction retirement points, i.e., locations where instructions can leave the pipeline, need to be identified. Please note that there can be multiple such locations due to early-out-conditions of some instruction classes. Having identified the instruction retirement locations, backward slices from these locations are computed.

Definition 6.7 — VHDL Backward Slice:
A *backward slice* contains all instructions that *may* influence the value of a signal (or variable) at the end of process execution, i.e., at suspension time. Any instruction not in the slice is guaranteed to not influence the value of the signal (variable).

Computing VHDL backward slices from all retirement locations, i.e., from the set of points where instructions are retired in the VHDL model, results in a set of slices. All instructions not contained in the union over these slices do not have any influence on the timing of the processor.

This yields all VHDL statements that influence the instruction retirement and therefore contribute to the instruction flow through the pipeline. All VHDL code that is not contained in the union over all computed backward slices is called "*timing-dead*" and can be safely removed.

Schlickling [Sch13] gives further details on how to perform slices on VHDL code and how VHDL can be analyzed statically.

Data Path Removal

The third model preprocessing step is the so-called *Data Path Removal* that removes the data paths from the VHDL model. This step is required by the aiT framework because there, the data paths have been factored out into a separate analysis – the loop/value analysis (cf. Section 4.3).

The removal of data paths is a generic term for two abstractions: the *memory abstraction* that removes the actual representation of memory cells and the *address abstraction* which is a specific domain abstraction from concrete addresses to address intervals (cf. Section 6.4.2). These abstractions are both described in the next section. Although data path removal consists solely of abstractions, it is listed in the model preprocessing part because its application is mandatory to produce a timing model suitable for the aiT framework. And the same holds for the other preprocessing steps.

6.4.2 Model Abstractions

So far, the model preprocessing as described in the previous section is responsible for removing those parts of a VHDL specification that are either deactivated by the system configuration or just unused due to the specific operational context of the target hardware. Additionally, all data paths have been removed to reduce the memory consumption (see above).

The model size reduction achieved by the steps described in the previous section might suffice for handling rather simple system architectures. But analyzing real world applications running on modern and thus complex superscalar architectures like the Freescale PowerPC 755 [Fre01] is a different challenge. This has been the reason for the introduction of state abstractions by Thesing [The04].

A *state abstraction* is the replacement of a concrete part of a system state by an approximation where the specific type of approximation depends on the underlying architecture.

In principle, such a lack of information about the system state might result in a loss of precision in the computed WCET bound because the timing behavior is not modeled exactly anymore. Fortunately, the timing is often not affected by this as for example an addition always takes the same amount of time independent of the argument values. By contrast, multiplications can be finished faster if one argument has leading zero bits. The lack of precise information about this condition requires to assume the entire range of execution times for multiplication. But this might be acceptable as multiplications are rare. Thesing's experiments [The04] has shown the industrial applicability of such an abstract interpretation for timing analysis. In the average, the computed time bounds in this work for representative Airbus software are about 13% higher than any measured execution time. This is comparatively precise for

such a sophisticated processor architecture as the bounds are still below the bounds derived with the Airbus' legacy methods.

The introduction of suitable abstractions still is an engineering task and cannot be fully automated. One contribution of this thesis (together with the work by Schlickling [Sch13]) is to provide tools supporting the introduction of state abstractions as much as possible. Model abstractions are incorporated iteratively until the resulting timing model is computationally tractable. Section 6.5 details about the application of the ideas presented in this section.

In general, three different abstractions to the processor state are offered and explained in the following.

Domain Abstraction

Domain abstractions are reflected syntactically as type changes in the model. The most prominent and common example is the *address abstraction* where the address domain is substituted by a domain of abstract values standing for certain sets of concrete values. Effectively, concrete addresses are now represented by intervals of addresses enriched with additional relational information (cf. Section 4.3.3 on page 78). Address abstraction is a consequence of the data path removal done in the model preprocessing (cf. Section 6.4.1). Domain abstractions are directly realized as a model transformation, which is described in Section 6.4.3.

Process Substitution

The active parts of a VHDL specification are processes. Normally, a process drives at least one signal containing the result of its computational task, e.g., the next address to be fetched from memory. Process substitution allows for replacing a concrete VHDL process by a custom implementation modeling less details or using a powerful abstraction.

The process substitution abstraction is directly realized by a corresponding model transformation that is described in Section 6.4.3.

Process substitution has been successfully done for caches and is known as the *cache abstraction* [Fer97, FW98, FMWA99, FW99]. Thereby, the concrete

cache is replaced by an abstracted cache storing the maximal ages of all lines that are definitively in the cache.

Memory Abstraction

The domain abstraction and process substitution can be applied multiple times until the resource consumption of the resulting pipeline analysis is acceptable. In contrast to that, the memory abstraction is employed only once.

Processors execute programs that are kept in memory. For execution, instructions have to be fetched, decoded, executed and the results must be written back to memory. But especially large memory arrays like main memory blow up the timing model so that they have to be extracted. Fortunately, contents of registers, memory addresses being accessed by a program, and contents of memory cells can be computed using the value analysis and do not require a cycle-wise simulation of the processor's behavior.

The memory abstraction makes use of this fact and removes the concrete representation of storage, i.e., registers and memory cells. This is possible for the same reasons as in the aiT framework; There are cases where the timing behavior of a hardware component is independent of the concrete values of registers and memory cells at that point of execution. For example, the latency of instructions is normally not affected by the content of their operands although there might be exceptions depending on the concrete hardware architecture, e.g., early out conditions of multiplier units. But for memory reads and writes, the affected memory areas are certainly of special interest as different areas often have different access latencies because either different areas are served by different types of memory or types of peripheral devices in the case of memory-mapped input/output communication. Thus, the value analysis has to be queried whenever values affect the timing behavior.

This abstraction is one of two parts of the data path removal from the model preprocessing phase (cf. Section 6.4.1). Its realization by suitable model transformations is described in Section 6.4.3.

General Properties and Requirements

All of the above mentioned abstractions of the original VHDL model must only change the timing behavior in safe manner, i.e., they may lead to over-

estimations but never to under-estimations. Moreover, they are architecture specific in terms of applicability and necessity. As described in Section 4.3.3, imprecise values lead to multiple possibilities in the execution of a program. If the number of alternatives is large, the computational complexity might be higher than without the particular abstraction. So the introduced transformations must be chosen carefully and timing model development requires some general experience with abstract interpretation.

The above mentioned possibility of multiple successor states for one state due to abstractions render the timing model nondeterministic. In that case, the micro-architectural analysis computes multiple execution paths through the program and annotates each path with the execution times. The generation of a cache and pipeline analysis as described in Chapter 7 needs to cope with this nondeterminism.

6.4.3 Model Transformations

The application of the model preprocessing steps as well as state abstractions form a feedback loop between static analyzers as defined by Schlickling [Sch13] and transformations based on their results. This means that the applied transformations are chosen depending on the particular results of the analyzers. Previous sections have discussed in theory whose parts of an input VHDL model are of interest for a timing analysis and others that could be abstracted. Now, concrete model transformations are presented that modify the model accordingly to reach the desired effect. In general, these transformations are either code removals or replacements as will be described. Details about the tool implementation of the presented transformations are described in Chapter 10 whereas their usage and practical application is shown in Section 6.5. The supported transformations all need to fulfill the property of not affecting the timing behavior of the analyzed system in an unsafe way (see above in Section refsec:derivation-general-properties-requirements).

Of course, this only holds under certain assumptions on either the analyzed architecture and/or the context of the concrete application being executed on that architecture. For example, one must not remove parts of a VHDL specification which are used by the analyzed program, i.e., a floating-point unit cannot be excluded from the model if analyzed binaries do contain corresponding floating-point instructions. Or the provided custom simulation routines (cf. process substitution in Section 6.4.2) need to describe the timing

behavior of the replacing components correctly. Otherwise the resulting pipeline analysis would compute bounds which are not safe.

The following general transformations on VHDL models have been identified and supported so far.

VHDL Timing Dead Code Elimination

This transformation removes all code snippets that have been marked as timing dead by static analysis. Moreover the "timing-deadness" is getting transitively propagated for resolving additionally dead code due to the removals. Theoretically, this is not a complex problem. The tool VhdlTimingDeadCodeEliminator implements the timing-dead code elimination and technical details about the realization can be found in Section 10.4.1.

VHDL Domain Abstraction

As mentioned in Section 6.4.2, the domain abstraction mainly is a type change, i.e., a source-to-source transformation. Based on a given mapping from a source into a destination domain, all affected variables and signals are transformed. This certainly renders existing operators on the source type invalid. All such locations are collected and the interfaces for the needed operators on the destination domain are computed and emitted. The user has to provide the new operators.

Moreover, there is the possibility to incorporate "common" abstractions like the address abstraction and different transformation scopes. The latter means that the transformation can be restricted to a given list of identifiers or for all identifiers of the source type. If the transformation scope is restricted to a list of identifiers, all not transformed identifiers are reported in order to give an impression of additional model parts affected by the source type. This feature can be used for partially transforming the model so that the user can first focus on specific pieces of a specification. The application of this transformation is further explained in Section 6.5.

The type transformation is supported by the tool VhdlDomainAbstractor (cf. Chapter 10).

VHDL Process Replacement

This transformation supports the replacement of original VHDL processes by custom simulation routines provided by a human expert. A custom simulation routine means that the user provides an alternative implementation of a VHDL process. The most important reasons for that are:

- *Performance*
  As only the timing behavior of the VHDL process is of interest, an alternative implementation might be much more efficient in terms of simulation speed.

- *Refactoring*
  The structure of hardware components is often designed hierarchically, i.e., it is composed out of other smaller components. If one is able to implement a custom and generic simulation routine for the timing behavior of the original VHDL process, the simulation code can be used for other timing models, as well.

If one specifies the timing behavior of a hardware component directly in another language (like C/C++), the original VHDL process is removed in favor of the custom specification which is inserted. In general, the implementation language of the custom simulation routine is not important as long as the timing effect can be described. But the most important implementation language is C/C++ as the aiT framework is written in that language. Therefore the pipeline analyzer code generator described in Chapter 7 generates compatible C code.

VHDL Memory Abstraction

In Section 6.4.2, the memory abstraction has been introduced as the removal of data paths from the model. Thus, the VHDL design must be adapted to utilize the information from the value analysis instead of the real computation of addresses. For this purpose, all places where memory addresses are generated by instructions have to be identified. Partly, these locations already have been examined during address abstraction, so the identification step is rather simple. At these places pseudo VHDL processes have to be added that interface with the value analysis to retrieve its results.

For a processor specification, there are typically three locations to patch:

- *code fetch*,
- *data access generation* and
- *system bus interface*.

Code fetches are generated in the fetch unit and used to request new instructions from memory. Here, the existing address computation code is replaced by queries to the control-flow graph mapping. Corresponding functions provided by generic parts of the pipeline analysis (cf. Section 4.3.3) realize these queries to get information about the instructions that are to be fetched.

Another place is the load/store unit where data accesses are triggered. In that case, code must be inserted that queries the value analysis and replaces the existing address generation code. Corresponding functions in the pipeline analysis framework serve this purpose, similarly to the CFG queries mentioned above.

The system bus interface manages the bus transactions and generates requests to peripheral devices, where such requests originate from the above mentioned fetch and load/store units. Typically, this unit is aware of the cache architecture and adjusts requested data addresses to match double word or cache line boundaries, etc. Analogously to the load/store unit, address generation code is replaced by corresponding queries to the value analysis' results.

The identification of code locations to patch as well as the modification itself are so far not supported by any automatic transformation and therefore have to be done manually. Section 6.5.2 shows an example code location that has to be patched according to the description in this section.

6.5 Timing Model Derivation Workflow

The previous sections have shown how to extract the timing behavior of a given VHDL model as well as how to reduce its complexity by model transformations based on introduced state abstractions. Although transformations and abstractions are specific to the hardware as well as the analyzed application context[5], this section presents some common working patterns

[5] for example concerning the hardware configuration

of the above described method for deriving a timing model from a formal hardware description in VHDL.

In principle, a common understanding of the underlying hardware model is still required in order to properly extract the needed information. And working with VHDL designs of modern processors/systems is a complicated task as such models can consist of hundred thousands lines of code. To support the interactive exploration and understanding of the hardware behavior by examining its formal specification, there exist two tools:

- aiSee
  The interactive graph viewer aiSee [Abs11] can be used for the interactive exploration of the graph representation of a VHDL model which is generated by the VHDL parser described in Chapter 10. It has been developed by AbsInt GmbH [Abs12].

- VhdlSlicer
  A generic VHDL slicing tool has been developed by Schlickling [Sch13] and can be used to compute slices interactively in the above mentioned graph representation. This support is useful for the visualization of dependencies among the different control signals in such VHDL models. Most of the actions described below require an understanding of such dependencies and side effects in the analyzed VHDL code which renders the slicer an important tool.

The next sections now present a common application workflow for the extraction of timing information from VHDL models. After the application of the transformations and abstractions the resulting model can be fed into the pipeline analyzer generator described in Chapter 7 in order to see whether the model is feasible for timing analysis.

6.5.1 Application of Model Preprocessing

The model preprocessing is mainly responsible for a significant size reduction of the analyzed model. Different analyses and transformations are performed in order to remove parts of the model that either do not contribute to the timing behavior or can be ignored due to the operational context of the analyzed application, e.g., ignoring hardware interrupts.

Listing 6.1 – Iterative model refinement workflow – pseudo-code

```
1 Loop
2     * State assumptions for signal values
3     * Run the model refiner with that assumptions
4     * Run timing dead code eliminator
5     * finish or restart with further assumptions
6 until resulting model is suitable
```

Reset Analysis

The first analysis, the so-called *reset analysis*, is mandatory for the next step (see below). Its goal is to obtain initial values for VHDL signals. These default values are set when the system reset is activated in order to bring it into a defined starting state. To support this task, the so-called VhdlResetAnalyzer has been developed by Schlickling [Sch13]. It performs a simulation of the model execution under the assumption of an activated system reset and computes constant signal values.

Here, "constant" refers to a slightly unusual scope, namely the whole model. In the context of program analysis, "constant" only refers to a specific program point. The results of the reset analysis are used as initial assumptions for the iterative model refinement described in the next section.

Iterative Model Refinement

While the reset analysis described above computes initial signal assignments, the iterative model refinement step is actually responsible for the identification and marking of code snippets that can be removed from the specification. Based on assumptions of signal assignments, the tool VhdlAssumptionBasedModelRefiner performs an abstract interpretation of the VHDL model which identifies so-called "timing-dead" code that can be removed by the tool VhdlTimingDeadCodeEliminator.

This removal can be applied iteratively. The corresponding workflow has been illustrated by the pseudo-code in Listing 6.1. Assumptions about the analyzed VHDL model must be stated by the user. Initial assumptions are the ones computed by the reset analysis described above. The VhdlAssumptionBasedModelRefiner then automatically detects all inactive parts of the model. Using the VhdlTimingDeadCodeEliminator, these code snippets are removed

from the model effectively reducing its size and complexity. Afterwards, the user has to decide whether additional assumptions are required. Of course, the concrete assumptions depend on the analyzed VHDL design, but typical scenarios are discussed in Section 6.4.1.

Listing 6.2 on the facing page shows a VHDL code snippet from the LEON2 [Gai05] specification that is concerned with the SDRAM refresh counter. The code snippet shows VHDL statements that are guarded by an "`if`" statement. Its condition includes a check for an enabled signal `r.cfg.renable` which serves as a configuration signal for the SDRAM refresh handling. If this signal is *assumed* to be deactivated, i.e., hard-wire its value to `0`, as a result the code block becomes "timing-dead" and can be removed by the timing dead code elimination.

Identification of Pipeline Instruction Flow

As mentioned in Section 6.4.1, the resulting timing model mainly needs to represent the instruction flow through the processor pipeline. This is achieved by the timing dead code elimination described in Section 6.4.1.

Figure 6.3 on the next page illustrates how the control structure of a processor's pipeline looks for a DLX [Hor97] design. The pipeline consists of five stages:

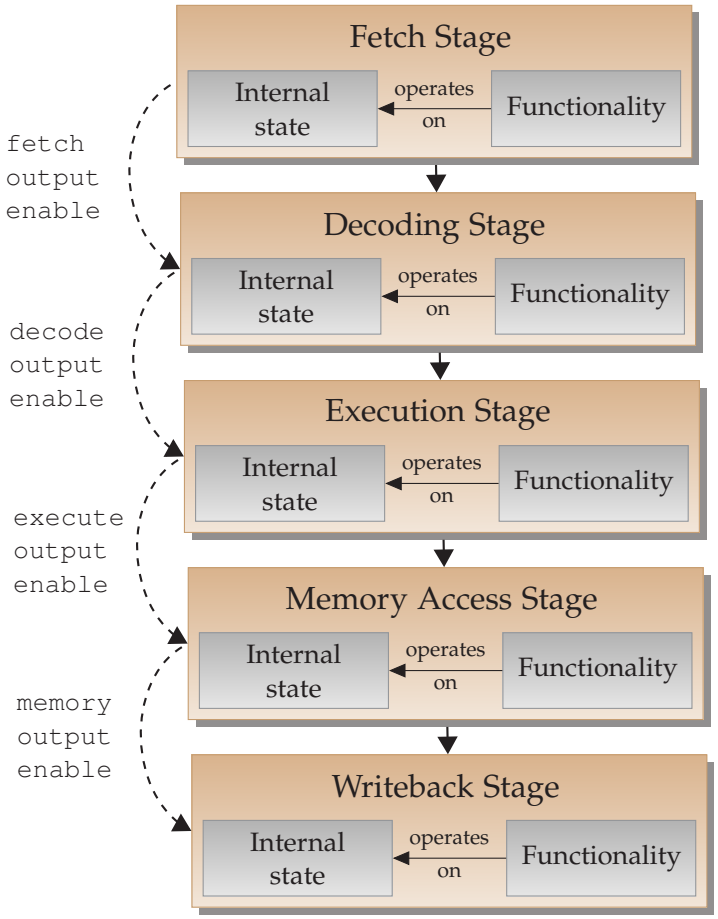
- *Fetch*: This stage is responsible for fetching the instructions to execute.
- *Decode*: In this stage, all fetched instructions are decoded.
- *Execute*: The actual execution of each instruction is done here.
- *Memory Access*: This stage is responsible for all kinds of memory accesses as a result of the instruction execution.
- *Writeback*: Instruction execution results are written back to registers.

In a typical VHDL specification, each pipeline stage consists of at least one process implementing the functionality of the particular stage. Additionally, each pipeline stage has its own internal data structures containing needed information – at least some kind of reference to the instruction that currently occupies the stage. The control structure of the pipeline is shown by the dashed edges in Figure 6.3. Each pipeline stage has its own output enable

Listing 6.2 – LEON2 SDRAM refresh counter snippets in VHDL

```
1 if (r.cfg.renable = '1') and (r.istate = finish) then
2   v.refresh := r.refresh - 1;
3   if (v.refresh(14) and not r.refresh(14))  = '1' then
4     v.refresh := r.cfg.refresh;
5     v.cfg.command := "10";
6   end if;
7 end if;
```

Figure 6.3 – Simple pipeline control structure



signal[6] which indicates whether the instruction can advance to the next pipeline stage. This means that processes implementing the functionality of a stage check these signals for incoming instructions. If an instruction is ready to advance to the next pipeline stage, the corresponding output enable control signal is active. The process of the successor stage detects this situation and updates its internal state accordingly by adding the new instruction.

Instruction retirement happens at the end of the *Writeback* stage, so the goal of the above-mentioned backward slicing is to identify the control statements checking for the output enable signals as well as the corresponding signal assignments. Ideally, only that control structure is needed for the timing model together with some sort of counter that represents how long an instruction remains within a pipeline stage. So, the slicing tool can be used for a better understanding and identification of the instruction flow through the processor pipeline.

As well as for the iterative model refinement, it depends on the concrete VHDL code how good the instruction flow can be separated from the more functional code snippets which make some VHDL designs more suitable for the purpose of extracting timing information than others (cf. Chapter 9).

6.5.2 Application of Model Abstractions

Up to this point, the size of the original VHDL model has been reduced without the introduction of abstractions except the needed steps for the removal of data paths, i.e., memory and address abstraction. Depending on the complexity of the hardware under analysis, this might already suffice to get a feasible timing analysis.

For modern processors used in embedded systems (even systems which are highly safety-critical), state abstractions are a must for the mentioned complexity reasons of the resulting pipeline analysis. The next sections list some commonly used abstractions that are also used in hand-crafted timing models of already existing timing analyzers provided by AbsInt GmbH.

[6] For simplicity, there is only one output enable signal shown. In a real specification, there are certainly more such signals.

Listing 6.3 – DLX effective address stage snippet in VHDL

```
1 -- Load-Store-Unit; Effective address stage
2 if LSU_EA_StageAvailable = '1' then
3     LSU_EA_AddrReg <= LSU_EA_AddrRegInput;
4     ...
5 end if;
```

Domain Abstractions

Domain abstractions are a common form of type transformations on the VHDL model. The above mentioned address abstraction (cf. Section 6.4.2) is a specific but the most prominent domain abstraction.

Listing 6.3 shows a VHDL code snippet of the superscalar DLX design [Hor97] representing the effective address stage within the load/store unit. Depending on the control signal `LSU_EA_StageAvailable` (in line 2), the address buffer of the effective address stage (`LSU_EA_AddrReg`) is filled according to the input signal `LSU_EA_AddrRegInput` (in line 3). Both signals are operating on concrete addresses and would therefore be replaced by intervals of abstract values containing the real addresses during the address abstraction. The resulting abstracted VHDL code can be seen in Listing 6.4 on page 138. Lines 2–8 first show an implementation of an assignment operator on address intervals which is then used to replace the original signal assignment. The type `AddressRange` realizing intervals of addresses is implemented as an array with two elements, where the first element (index 0) stores the lower interval bound and the second one (index 1) the upper bound.

In this case, only a simple assignment is needed but one can imagine that depending on the surrounding VHDL code, there might be more complex cases where even state splits would need to be incorporated. Then, it might be easier to express the abstraction directly in a custom simulation routine (cf. Section 6.5.2).

In addition to that, one might specify more type transformations on the VHDL model. An example candidate for a domain abstraction that is not an address abstraction can be seen in Listing 6.5 on the next page. Line 24 shows the usage of the signal `DataBus` that is declared in line 10 of the code listing. The declaration of the corresponding subtype `TypeBidirectionalDataBus` is shown in the first two lines of Listing 6.5. This 64bit wide vector can be

Listing 6.4 – Abstract DLX effective address stage snippet

```
1  -- Abstract signal assignment operator
2  procedure AddressRangeSignalAssign (
3        signal addrRange : out AddressRange;
4               value     : in  AddressRange) is
5  begin
6      addrRange (0) <= value(0);
7      addrRange (1) <= value(1);
8  end;
9
10 -- Load-Store-Unit; Effective address stage
11 if LSU_EA_StageAvaliable = '1' then
12     AddressRangeSignalAssign (LSU_EA_AddrReg,
13                               LSU_EA_AddrRegInput);
14     ...
15 end if;
```

Listing 6.5 – 60x bus read handling VHDL snippet

```
1  subtype TypeBidirectionalDataBus is
2      std_logic_vector(63 downto 0);
3
4  ...
5
6  entity Dlx is
7      port ( IncomingClock : in bit;
8             BusClock : out bit;
9             AddressBus : out TypeWord;
10            DataBus : inout TypeBidirectionalDataBus;
11            ByteEnable : out unsigned( 7 downto 0 );
12            TransferStart : out bit;
13            WriteEnable : out bit;
14            TransferError : in bit;
15            TransferAcknowledge : in bit;
16            InterruptRequest : in bichapters/t;
17            CacheInhibit : in bit;
18            Reset : in bit;
19            Halt : out bit );
20 end Dlx;
21
22 ...
23
24 BIU_IncomingData <= unsigned( To_bitvector( DataBus ));
25
26 \dots
```

Listing 6.6 – DLX register file access VHDL snippet

```
1     RF_DataA1 <=
2         X"0000_0000" when IF_InstrRegA(25 downto 21) = "00000"
3       else
4         RF_Reg(To_Integer(IF_InstrRegA(25 downto 21)));
5
6     RF_DataA2 <=
7         X"0000_0000" when IF_InstrRegA(20 downto 16) = "00000"
8       else
9         RF_Reg(To_Integer(IF_InstrRegA(20 downto 16)));
10
11    RF_DataB1 <=
12        X"0000_0000" when IF_InstrRegB(25 downto 21) = "00000"
13      else
14        RF_Reg(To_Integer(IF_InstrRegB(25 downto 21)));
15
16    RF_DataB2 <=
17        X"0000_0000" when IF_InstrRegB(20 downto 16) = "00000"
18      else
19        RF_Reg(To_Integer(IF_InstrRegB(20 downto 16)));
```

replaced by a Boolean value only indicating that the bus is driven with valid data. For a timing model, the concrete value on the bus is not important for in-order memory subsystems, i.e., where the data beats occur in the order of the corresponding bus requests. The reason for this is that the bus unit part of the model has precise knowledge about all pending bus transactions so that an incoming data beat can be unambiguously mapped to its originating memory request. The abstraction from the type `TypeBidirectionalDataBus` to the type `boolean` certainly implies lots of transformations on other signals/variables as well as the operators on these original domains. Such a work-flow is supported by the tool VhdlDomainAbstractor as described in Chapter 10.

Memory Abstraction

Listing 6.6 partly shows the specification of the register file access in the DLX [Hor97] design. The four signals `RF_Data` represent the operands of currently executed instructions. In lines 4, 9, 14 and 19, the register file data structure (`RF_Reg`) is accessed in order to retrieve the requested register contents. During data path removal, these accesses would be replaced

by procedure calls that actually would query the value analysis results. The declaration of the register file data structure is removed as well. Such query procedures would be implemented in custom simulation routines as presented in the next section.

Custom Simulation Routines

A prominent example for this is the cache abstraction. It is already employed in the aiT framework for all existing timing analyzers. The complete cache update is here replaced by a custom library with abstract data structures. This library implements the techniques introduced in [Fer97, FW98, FMWA99, FW99].

Selecting processes for replacement means that the essential timing behavior is reimplemented. Besides the cache abstraction, there is no "must-have" or typical process replacement that could be mentioned here. The concrete selection depends on the design under analysis.

Additionally, the removal of purely functional VHDL code can be realized by the invention of custom simulation routines as described in the next section.

Removing Functional Code

A VHDL model contains all details about the hardware component which is much more than the pure timing behavior. Depending on the structure of the VHDL code, the model preprocessing steps presented in Section 6.5.1 might not remove all statements which only refer to functional details. For a timing analysis, it is not necessary to know the functional design details of a unit. The timing information comprises the details about the instruction flow through the pipeline. So, it actually suffices to know how long each instruction stays in each stage of the processor pipeline.

For example, a simple arithmetic unit whose instructions have a fixed execution latency[7] may be abstractly represented by a reference to the currently executed instruction together with a simple counter which gets initialized with the particular instruction execution time on the unit. In each update cycle, the counter value is decremented so that the instruction leaves the unit if the counter value has expired. Imagining that a unit has one or

[7] Execution latencies may vary but within an interval of fixed bounds.

even more reservation stations. In this case, the model would just contain a corresponding number of additional instruction references. By this, the complete functional details of the arithmetic unit can be ignored. If there are instructions that have a variable execution time on the simple arithmetic unit, a specific state for each possible execution time has to be created (cf. the definition of state splits in Section 6.2.1).

In Listing 6.7 on the following page, some VHDL code snippets from the arithmetic-logical unit of the DLX [Hor97] design are shown. The actual functionality of this execution unit is implemented in the function `Alu` (cf. lines 3–21) whose returned result determines the output of the unit (cf. lines 25–27). Removing `Alu` and assuming a fixed value for the output of the execution unit then decreases the size of the model. The execution time of an instruction can then be represented by a counter as mentioned above.

Technically, such a removal of functional code segments can be achieved by replacing VHDL processes using the VhdlProcessReplacer. But the applicability strongly depends on the structure of the VHDL code. If the complete logic of a functional unit is coded within one single big process, the VHDL code must be rewritten first as it is not suitable for that kind of abstraction (cf. Chapter 9). Despite the lack of a representative survey on industrial hardware designs, at least the designs from the avionics and automotive domain do not exhibit such a coding style. This also reflects in the results of the experiments described in Chapter 11.

6.5.3 Derivation Step Categorization

From the user's perspective, a categorization of the presented model transformations according to their necessity is certainly of interest. The following paragraphs classify the model preprocessing steps and abstractions that have been described in Section 6.4.1 and Section 6.4.2 alongside the particularly needed model transformations. In general, creative parts that require a deep understanding of the hardware design have to be done manually where other more schematically parts can be done fully automatic.

Mandatory Derivation Steps The only fully mandatory step is the *removal of data paths* in the model preprocessing phase. Associated transformations are the VHDL memory abstraction (cf. Section 6.4.3) and the address abstraction (cf. Section 6.4.3). To apply these transformations, corresponding

Listing 6.7 – DLX arithmetic/logical unit snippet in VHDL

```
1 ...
2
3 function Alu( Source1, Source2 : TypeWord;
4               AluFunction: TypeAluFunction ) return unsigned
5 is
6     variable Result : TypeWord;
7     variable Error  : bit;
8 begin
9     case AluFunction is
10        when cAlu_add => Result := Source1 + Source2;
11        when cAlu_sub => Result := Source1 - Source2;
12
13        when cAlu_and => Result := Source1 and Source2;
14        when cAlu_or  => Result := Source1 or Source2;
15        when cAlu_xor => Result := Source1 xor Source2;
16
17        ...
18    end case;
19
20    return Error & Result;
21 end Alu;
22
23 ...
24
25 ALU_AluOutput <= Alu( ALU_SourceDataReg1,
26                       ALU_SourceDataReg2,
27                       ALU_DecoderInfoReg );
28 ...
```

code locations have to be identified. This is an engineering task, requires an understanding of the hardware design and therefore has to be done manually. Replacing existing code with an interface to the value analysis and the control-flow graph might be done automatically. That depends on the concrete VHDL code. For example, if address generation is encapsulated into a separate process, its replacement is simple and can be done by a custom simulation routine so that the manual work reduces to the implementation of the custom simulation routine.

Another mandatory action point results from the application of *domain abstractions*, namely the implementation of operators for the target domains. Here, detection of newly needed operators and their interface are automatically reported by the domain abstraction tool VhdlDomainAbstractor. In

contrast, their implementation has to be provided by the user.

In principle, no further steps are actually required to get a working timing analysis, i.e., the application of any model abstractions besides the data path removal is actually not needed. But the resulting analysis complexity in terms of their space consumption and runtime would not be satisfactory.

Optional Quality Increasing Derivation Steps The second category of derivation steps are optional and effectively increase the quality of the resulting timing analysis. Here, quality refers to the precision of the computed WCET bounds (a higher quality leads to more precise bounds with less overestimations) as well as the overall analysis complexity (reducing the size of the model decreases the complexity of the analysis).

The *environmental assumption-based model refinement* from the preprocessing phase is one of such optional steps. Realized by the application of the reset analysis, the assumption-based model refiner and the dead code eliminator, this step is automated except the invention of corresponding assumptions.

Another optional derivation step is the *timing dead code elimination*. Like the environmental assumption-based model refinement, this is a step from the model preprocessing and its goal is to restrict the timing model to those parts defining the instruction flow through the pipeline. The start, the identification of code locations where instruction retirement happens, is an engineering task and has to be done manually by the user. Backward slices from the identified locations are then automatically possible using Schlickling's Vhdl-Slicer [Sch13]. Based on these slices, further dead-code assumptions can be incorporated into the model. Where their invention is up to the user, their actual processing is automated as in the environmental assumption-based refinement.

In general, any kind of *model abstraction* represents an optional but quality-increasing derivation step. Their different types are described in Section 6.4.2.

6.6 Model Transformation Phase Coupling

The phase coupling problem as known from compiler design [Veg82] describes the phenomenon that different compiler optimization methods are

interdependent, i.e., decisions made by one optimization impose restrictions to another optimization. Finally, the order in which different optimizations are applied leads to different results regarding the quality of the generated code.

Looking at the different model transformations presented in Section 6.4.1 and abstractions in Section 6.4.2, the question arises whether they are phase-coupled or not. Although there is no exhaustive examination of that problem, it seems that there is no tight phase-coupling between the different model transformations as they are rather independent by definition. From a practical point of view, it might be good to first throw away as much of the model by timing dead code elimination as possible (cf. Section 6.4.3) before applying any state abstractions (cf. Section 6.4.2) as there is less code to be converted in the later step.

6.7 Summary

In one sentence: this chapter defines how to derive timing models suitable for WCET analysis from formal hardware descriptions in VHDL. To introduce this process, a description of timing models is presented to give the reader an impression of what is stored in such a model alongside some important properties like their nondeterministic processor update. Additionally, differences between a local and global worst-case computation are highlighted.

Afterwards, it is shown how a VHDL model is syntactically mapped to the chosen intermediate format CRL. In this context, the semantic level reduction is presented, a technique to express concurrent VHDL models as sequential programs in order to enable the application of abstract interpretation.

The derivation process of timing models from VHDL designs is described in a process-oriented way in Section 6.4.1 and Section 6.4.2. First, a VHDL model undergoes different model preprocessing steps that reduce its size and introduce necessary modifications required by the aiT framework. In a second phase, model abstractions are introduced to further compress the hardware representation. These defined derivation steps can be realized by concrete model transformations which are defined in Section 6.4.3.

To illustrate the timing model derivation, common working patterns together with some examples are shown. The description goes through the different derivation steps and explains their application.

In the end, a categorization follows that classifies the steps into mandatory and optional ones. It is pointed out whether associated transformations can be performed automatically or need manual contributions from the user.

7

Pipeline Analyzer Generation

"By three methods we may learn wisdom: First, by reflection which is noblest; Second, by imitation which is easiest; and third by experience which is the bitterest."

(Confuzius)

7.1 Overview

Where the previous chapter has detailed about how to extract information about the timing behavior of a system from its formal specification (in VHDL) into a so-called timing model, this chapter describes a method for the automatic generation of a static analyzer based on such a model. The analyzer can be used within the aiT framework as the pipeline analysis.

As mentioned in Section 4.3, this pipeline analysis performs an abstract simulation of a task's execution on the target system. This implies that the generated static analyzer needs to cope with an abstracted model of the hardware, which is nondeterministic in general. In other words, the simulation process might compute multiple successors for a given input system state leading to multiple possible execution paths partially with different costs in terms of execution time. For safety reasons, the generated pipeline analysis needs to follow all possibilities.

In order to describe the generation process, the next section first defines simulation semantics for a concrete VHDL model. Analogously, simulation semantics in the presence of employed abstractions are presented in the section thereafter with differences highlighted.

These abstract operational semantics represents the formal basis for the pipeline analyzer generation and the corresponding tool implementation (cf. Section 10.5) realizes this exactly.

7.2 Concrete Simulation

As mentioned in Chapter 4, it is not advisable to use the simulation of a concrete VHDL model for timing analysis of safety-critical real world applications because the needed computing resources are too high. Even though, in order to formalize the simulation process this section addresses the simulation of a concrete VHDL model, i.e., a formal hardware specification without any abstractions. This is needed for showing the differences to the simulation of an abstracted VHDL model as resulting from the timing model derivation process presented in the previous chapter.

7.2.1 Operational Semantics

In the following, the operational semantics for simulating a concrete VHDL model is described using inference rules as defined by Nielson [NN92]. This is a common technique for specifying the semantics of a programming language. The inference rules additionally define an interpreter for the language. The central point is a program state. Inference rules specify changes to this state as well as the conditions under which the state change takes place. By this, the semantic effect of the execution of each language construct, i.e., their effective changes to the system state, can be described.

The inference rules defined throughout this section have already been published by Maksoud [MPS09].

Sequential Process Execution

To define the execution of a VHDL process, a *context* has to be specified:

Definition 7.1 — Process Execution Context:
The *context* of a process p needed for its sequential execution is the tuple (Θ, ζ, Π) where

- Θ is called the *environment* mapping logical names to values. A *logical name* in VHDL can be either a variable v, a signal s, or a scheduled signal $\bar{s}$.
- ζ is the program counter. The function $next(\zeta)$ returns the address of the next statement of a program Π. This predicate has been taken from Hymans [Hym04]. If the current statement is the last one in a program, $next(\zeta)$ returns the special program counter ζ_{end} indicating that the process has completed its execution. The function $start(b)$ takes a list of statements b and returns the address of the first statement in b.
- Π is the list of statements of p, also called *program*.

The evaluation function *eval* embeds expression evaluation in VHDL. Corresponding operational semantics has been defined by C. Hymans [Hym04] by the specification of inference rules for all expression types and their operators.

The relation $\rightharpoonup_{seq}$ of sequential execution is defined by the following rules.

The rule *var* (7.1) defines the semantic effect of a variable assignment.

$$var \quad \frac{\Pi[\zeta] \Rightarrow v := expr; \qquad \Theta \vdash eval(expr) = u \qquad \Theta' = \lambda t. \begin{cases} u & \text{if } t = v \\ \Theta(t) & \text{otherwise} \end{cases}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta', next(\zeta), \Pi)} \tag{7.1}$$

If the statement to which the current program counter ζ points is a variable assignment ($v := expr;$) whose expression $expr$ evaluates to u under the current environment Θ, the context (Θ, ζ, Π) of the current process ρ is changed to the new context $(\Theta', next(\zeta), \Pi)$ where Θ' is the old environment Θ with the value of variable v changed to u.

Signal assignments are covered by rule *sig*:

$$sig \quad \frac{\Pi[\zeta] \Rightarrow s <= expr; \qquad \Theta \vdash eval(expr) = u \qquad \Theta' = \lambda t. \begin{cases} u & \text{if } t = \bar{s} \\ \Theta(t) & \text{otherwise} \end{cases}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta', next(\zeta), \Pi)} \tag{7.2}$$

If the statement to which the current program counter ζ points is a signal assignment ($s <= expr;$) whose expression $expr$ evaluates to u under the current environment Θ, the context (Θ, ζ, Π) of the current process ρ is changed to the new context $(\Theta', next(\zeta), \Pi)$ where Θ' is the old environment Θ with the scheduled value of signal s changed to u.

While in case of a variable assignment, the assigned value is directly visible in the environment, this is not the case for signal assignments. Here, the new value must be seen as the future (or scheduled) value (cf. Section 5.2.3). Note that during evaluation, *eval* always works on the current values of signals, never on scheduled ones. This exactly matches the VHDL semantics.

The next two rules specify the behavior of `if-then-else` statements.

$$tcond \quad \frac{\Pi[\zeta] \Rightarrow \texttt{if}\ expr\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{end if;} \qquad \Theta \vdash eval(expr) = true}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b_1), \Pi)} \tag{7.3}$$

If the statement to which the current program counter ζ points to is an `if-then-else` conditional statement whose expression $expr$ evaluates to *true*

under the current environment Θ, the context (Θ, ζ, Π) of the current process ρ is changed to the new context $(\Theta, start(b_1), \Pi)$, i.e., control is transferred to the first instruction of the subsequent block b_1. If the conditional expression *expr* evaluates to *false*, the control is transferred to the first instruction in the `else` block b_2 which is shown by the rule *fcond*:

$$\textit{fcond} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{if}\ expr\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{end if;} \\ \Theta \vdash eval(expr) = false\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b_2), \Pi)} \tag{7.4}$$

Obviously, `switch` statements can be easily transformed into `if-then-else` cascades, so the rules for these statements are left out. Actually, the translation from VHDL to the intermediate representation CRL performs this conversion as described in Chapter 10. Without loss of generality, `if` statements without `else` case are converted into `if-then-else` statements with empty `else` cases. The empty cases are handled by the rule *skip* (see below).

Similarly to conditional statements, the rules *tloop* and *floop* specify the behavior of `while` loops.

$$\textit{tloop} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{while}\ expr\ \texttt{loop}\ b\ \texttt{end loop;} \\ \Theta \vdash eval(expr) = true\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b), \Pi)} \tag{7.5}$$

If the statement to which the current program counter ζ points to is a `while` statement whose expression *expr* evaluates to *true* under the current environment Θ, the context (Θ, ζ, Π) of the current process ρ is changed to the new context $(\Theta, start(b), \Pi)$, i.e., control is transferred to the first instruction of loop body b. If the conditional expression *expr* evaluates to *false*, the control is transferred to the next statement after the whole loop statement which is shown in rule *floop*:

$$\textit{floop} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{while}\ expr\ \texttt{loop}\ b\ \texttt{end loop;} \\ \Theta \vdash eval(expr) = false\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, next(\zeta), \Pi)} \tag{7.6}$$

The semantic effect of `for` loops can be defined analogously and is left out here.

However, the above described rules do not fully specify all possible language constructs as there might be empty blocks, e.g., in `if-then-else` statements.

Those are covered by the following rule *skip*. The environment is not updated, control is simply transferred to the next statement.

$$skip \quad \frac{\Pi[\zeta] \Rightarrow ;}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, next(\zeta), \Pi)} \tag{7.7}$$

After the last statement in a process, the special program counter ζ_{end} is reached and process simulation has reached a final state at that point. The process then suspends, i.e., its program counter ζ is set to the special value ζ_{sus}. This state transition is covered by the rule *stop*.

$$stop \quad \frac{}{(\Theta, \zeta_{end}, \Pi) \rightharpoonup (\Theta, \zeta_{sus}, \Pi)} \tag{7.8}$$

Simulation Semantics

Process termination in the sequential execution rules is embedded in the definition of the function $next(\zeta)$. A process terminates, i.e., it suspends execution when reaching the special program counter ζ_{sus}. For this, the sequential execution rules defined in the previous section need to be applied iteratively which requires a global simulation context:

Definition 7.2 — Simulation Context:
The *simulation context* of a VHDL model is the tuple $(\Theta, \boldsymbol{\rho})$ where

- Θ is the environment as defined in Section 7.2.1 and
- $\boldsymbol{\rho}$ is a map from process labels $\mathbb{L}$ to processes.

A process $\boldsymbol{\rho}(l)$ is a tuple $(\zeta_l, \Pi_l, \omega_l)$ where ζ_l and Π_l are the program counter and the program respectively. And ω_l is a set of signal names representing the sensitivity list of $\boldsymbol{\rho}(l)$.

Remark:
Throughout this thesis, VHDL designs are assumed to be correct with respect to their process sensitivity list specifications, i.e., all signals read by a process p are listed in the sensitivity list of p.

Based on this simulation context, process execution, i.e., iterative application of the sequential execution rules can be defined.

$$\exists l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_l, \Pi_l, \omega_l) \wedge \zeta_l \neq \zeta_{sus} : (\Theta, \zeta_l, \Pi_l) \rightharpoonup_{seq}^{*} (\Theta', \zeta_{sus}, \Pi_l)$$

$$exec \quad \frac{\boldsymbol{\rho}' = \lambda(t \in \mathbb{L}).\begin{cases}(\zeta_{sus}, \Pi_l, \omega_l) & \text{if } t = l \\ \boldsymbol{\rho}(t) & \text{otherwise}\end{cases}}{(\Theta, \boldsymbol{\rho}) \rightharpoonup (\Theta', \boldsymbol{\rho}')} \tag{7.9}$$

The simulation context $(\Theta, \boldsymbol{\rho})$ is changed into a new context $(\Theta', \boldsymbol{\rho}')$ by rule *exec* if there is at least one process $\boldsymbol{\rho}(l)$ whose program counter value is not equal to ζ_{sus}. The process execution context of each process $\boldsymbol{\rho}(l)$ matching this condition is changed from (Θ, ζ_l, Π_l) to a new execution context $(\Theta', \zeta_{sus}, \Pi_l)$ by iteratively applying $(\rightharpoonup_{seq}^{*})$ the sequential execution rules (7.1)–(7.7). These rules define the relation $\rightharpoonup_{seq}$ of sequential process execution. By this, the rule *exec* advances the sequential execution of all active, i.e., not suspended, processes. Rule *exec* matches as long as there exists a non-suspended process. For multiple such processes, no specific order in which they are processed is defined here. The special semantics of VHDL (cf. Section 5.2.3) allows to choose an arbitrary execution order among the existing processes.

As mentioned in Chapter 5, the semantics of VHDL has two levels. After all processes have finished their execution and are suspended, all scheduled signal assignments for the current simulation time take effect. This might trigger a repeated execution of a process depending on its sensitivity list (cf. VHDL semantic description in Chapter 5) which is covered by the following rule *delta*:

$$\forall l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_{sus}, \Pi_l, \omega_l) \qquad \Theta' = \lambda t.\begin{cases}\Theta(\bar{s}) & \text{if } t = s \\ \Theta(\bar{s}) & \text{if } t = \bar{s} \\ \Theta(t) & \text{otherwise}\end{cases}$$

$$delta \quad \frac{\boldsymbol{\rho}' = \lambda(l \in \mathbb{L}).\begin{cases}(start(\Pi_l), \Pi_l, \omega_l) & \text{if } \exists s \in \omega_l : \Theta(s) \neq \Theta(\bar{s}) \\ (\zeta_{sus}, \Pi_l, \omega_l) & \text{otherwise}\end{cases}}{(\Theta, \boldsymbol{\rho}) \rightharpoonup (\Theta', \boldsymbol{\rho}')} \tag{7.10}$$

By this, the rule *delta* activates processes and applies delayed assignments, i.e., all scheduled signals become visible. For each process $\boldsymbol{\rho}(l)$ having at least one signal in its sensitivity list ω_l changed, the program counter ζ_l is set to the first instruction in the program $start(\Pi_l)$.

The inference rules presented in this section do not describe how simulation time advances. Simulation proceeds until no more updates can be done and the simulation time then must be updated by an external module. Since a process in synthesizable VHDL cannot be sensitive on a signal it drives and there is no possibility to wait for a timeout, there is no way to model the frequent change of a clock signal.

Using the operational semantics defined by the inference rules in the previous and this section, the effect of the execution of all language constructs of the synthesizable subset of VHDL is defined and can be used to generate a simulator given an input VHDL model. However, specifications of processors are too big and typically contain information having little or no impact on the timing behavior of the processor. Therefore, abstractions have to be applied to eliminate all unnecessary or expensive details.

7.2.2 Activation Sequences

Simulating the execution of a task with a given VHDL model would require to implement the two-level semantics as described in Chapter 5. In addition to the effect of a single process execution, process reactivation – VHDL delta cycles – need to be incorporated. In the operational semantics (cf. Section 7.2.1), the rule *delta* already covers this.

This means, additional simulation code has to be written implementing the *delta* rule that cannot be derived from the actual VHDL model. Fortunately, this code can be written once and is generic for all VHDL models because it follows the language definition.

Definition 7.3 — Activation Sequence:
The sequence of delta cycles between two consecutive advancements of simulation time is called an *activation sequence* as the underlying repeated process execution is needed to yield the next "steady" state, i.e., there are no scheduled signal assignments left for the current simulation time.

Each call to the cycle-wise update within the aiT framework (cf. Section 4.3) would expect to result in the final state of such an activation sequence. This property is independent from any employed model abstraction as it directly follows from the VHDL language specification.

7.2.3 Simulation Traces

The result of a simulation of a concrete VHDL model is a sequence of system states, also called trace (cf. its definition in Section 4.3.3). By selecting those states on a trace matching a given criterion, e.g., all states with an activated bus signal, this trace subset, which is computed by the timing analysis, can be compared to real hardware traces. Actually, this is an existing method for the validation of hand-crafted timing models [Sch05] and described in Section 8.2.

Even with a concrete VHDL model, it is not enough to simulate one single trace because the starting state of the system is not known in general. The aiT framework assumes that the analyzed code is run without interference from any other task that might have been executed previously as well as without any preemption, i.e., the processor pipeline is considered to be empty as well as the cache content[1]. Despite this, hardware traces have shown that the pipeline analysis needs to consider different instruction queue contents accommodating remaining instructions from a previously executed task. Those instructions actually are not executed anymore but still exist in the instruction queue occupying slots and therefore slow down the start of the execution a bit.

Another reason for the generation of more than one starting state are multiple clock domains. The main system bus is clocked with a fraction of the processor core clock frequency resulting in an offset between both clocks. As the starting offset between both clocks is not known, all possible displacements need to be taken into account.

In the end, simulation of a concrete VHDL model within a timing analysis would result in a set of simulation traces due to multiple starting states of the system. For an abstracted model, the situation is more complex as detailed in the next section.

7.3 Abstract Simulation

Abstract simulation, i.e., the simulation of an abstracted VHDL model (cf. Chapter 6) generally works like the concrete simulation as described in the

[1] Actually, the cache content can be configured among an initial empty or chaos state where chaos represents unknown contents.

previous section. Additionally, it has to cope with the nondeterminism of the underlying timing model as a result of the abstractions (cf. Section 6.2.1).

The next section first describes the effect of abstractions on the simulation result, namely the presence of state splits leading to simulation trees rather than single traces. Afterwards, Section 7.3.2 defines the abstract operational semantics for the simulation of abstract VHDL models.

7.3.1 Simulation Trees

Introducing abstractions to a program often implies a loss of information about the state of the program, e.g., values of variables or signals, etc. This information may be either missing or is not precisely known which is also caused by the fact that static analyses in principle work independently of any input. Due to the fact that program flow, i.e., the control flow, typically depends on the program state, i.e., the current environment, simulation of abstracted programs must cope with nondeterminism to cover all possible execution paths.

In a deterministic simulation, the program state is precisely known at all program points. The execution of a statement within a process updates the environment exactly in the way described in Section 7.2.1, i.e., there are no uncertainties.

By contrast, in a nondeterministic simulation of a VHDL model, a computation potentially yields more than one result leading to so-called *state splits*.

Definition 7.4 — System State Split:
A *system state split* happens if the state transition system leads to more than one possible successor state for one input state due to imprecise knowledge about the concrete state of a system. Multiple successor states are then generated which all differ in some part(s) of their state.

So, a simulation must proceed along all possible paths. Thus, simulating VHDL modules containing state splits proceeds along a *tree* rather than a trace.

Definition 7.5 — Simulation Tree:
In the presence of state splits where the state transition computation for

Listing 7.1 – Sample memory controller in VHDL

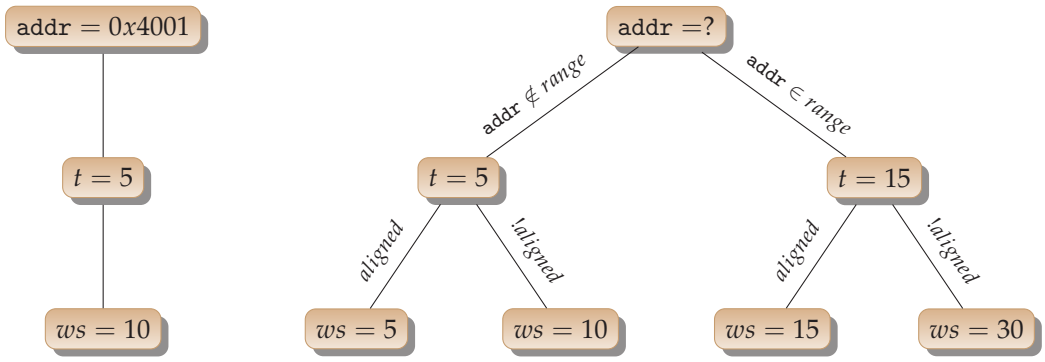
```
1  entity mem_ctrl is
2      port (addr : in integer; ws : out integer);
3  end entity;
4
5  architecture arch of mem_ctrl is
6      function access_time (a: integer) return integer is
7          begin
8              if a >= X"0000" and a < X"1000" then
9                --access to slow memory
10               return 15;
11             else
12               --access to fast memory
13               return 5;
14             end if;
15         end;
16 begin
17     P: process (addr)
18             variable t: integer;
19     begin
20         t := access_time(addr);
21         if addr mod 4 /= 0 then
22             --address not aligned
23             t := t * 2;
24         end if;
25         ws <= t;
26     end process;
27 end architecture;
```

one single input state results in multiple successor states, the simulation from one starting state spans a tree structure, the *simulation tree*.

Due to the presence of *timing anomalies* (cf. Section 6.2.2), it is generally not possible to ignore any path for safety reasons.

Listing 7.1 shows the VHDL description of a simplified memory controller. For a given address `addr`, the module computes the number of wait-states `ws` required for accessing the physical memory cells. Usually, the number of wait-states depends on the addressed memory type and the access type, e.g., aligned or non-aligned accesses. So, the decisive factors in this simplified memory controller are whether the access addresses the slower or the faster memory, i.e., `addr` $\in [0x0000, 0x1000)$ within the function `access_time`, and whether the given address `addr` is aligned or not.

Figure 7.1 – Simulation trace vs. simulation tree



(a) Trace (b) Simulation Tree

During a concrete execution of this module, `addr` is precisely known, e.g., when `addr` $= 0x4001$, hence the function `access_time` will return 5, and since `addr` is not aligned, the resulting number of wait states `ws` is 10.

Introducing a domain abstraction on `addr` from a concrete value to an interval, simulation becomes more difficult. The range check in function `access_time` may become ambiguous, i.e., it can be both, *true* and *false*, so the function might return two values, namely 5 and 10. This introduces nondeterminism into an abstract simulation, resulting in a state split. Furthermore, the check in process `P` also becomes ambiguous, resulting in an additional state split. A safe abstract simulation must result in four values of wait states, namely 5, 10, 15 and 30.

Figure 7.1 on page 158 outlines in part (a) the execution trace in the deterministic case, and the abstract simulation tree in the nondeterministic case in part (b) on the right. This example shows that a simulation coping with abstractions needs to keep track of all points where nondeterminism is encountered, i.e., all split points. Using the tree metaphor, a split point represents a vertex having more than one child. Every path through the tree leading to a leaf corresponds to an execution trace. Please note that due to abstractions there might exist a path in the tree that does not correspond to a real execution trace.

Analogously to the simulation of a concrete VHDL model, there must be multiple different starting states (cf. Section 7.2.3) so that the overall simulation result is a forest of simulation trees.

Listing 7.2 – Abstract simulation preprocessing example

```
1 --- Original code
2 if f1(i) + f2(j) * b > 1 then
3 ...
4
5 --- Preprocessed code
6 t1 = f2(j);
7 t2 = t1 * b;
8 t3 = f1(i);
9 t4 = t2 + t3;
10 t5 = t4 > 1;
11 if t5 then
12 ...
```

The next section describes the operational semantics of simulating abstracted sequential VHDL code.

7.3.2 Abstract Operational Semantics

Analogously to the definition of the operational semantics for the simulation of a concrete VHDL model (cf. Section 7.2.1), this section introduces inference rules [NN92] to specify the semantic effect of the simulation of an abstracted timing model. First, the sequential execution of processes is defined for all language constructs. Then the effect of the second semantic level of VHDL – the delta cycles – is defined.

An earlier version of the inference rules defined in this section has been published by Maksoud [MPS09].

Abstract Sequential Process Execution

Without loss of generality, the VHDL code is assumed to be preprocessed to avoid any nondeterminism in compound expressions, i.e., there shall be at most one operation or function call in assignment statements, and no operations at all in the conditions of `if` and `loop`-statements. The example code in Listing 7.2 (taken from [Mak07]) illustrates the preprocessing.

In the original code in line one, the expression $\texttt{f1(i)} + \texttt{f2(j)} * \texttt{b} > \texttt{1}$ is split up into its subexpressions using five temporary variables `t1,...,t5` (lines

6–10). By this, each assignment to one of the temporary variables contains at most one function call or operator usage. In the end, the conditional expression in the preprocessed `if` statement (line 11) is just a reference to the temporary variable `t5`.

The motivation behind this preprocessing is just to keep the abstract simulation of `if` and `loop` statements simple as state splits now can only happen in variable and signal assignments.

Analogously to the concrete sequential process execution, the abstract execution of a process needs a context, too, in order to define the operational semantics.

Definition 7.6 — Abstract Process Execution Context:
The *context* for sequential execution of an abstracted process p can be defined as the tuple (Θ, ζ, Π) where

- Θ is an environment as defined for the concrete process execution context in Section 7.2.1.
- ζ is the program counter as defined before. Furthermore, ζ_{end} is a special address indicating that the current simulation path is at its end, i.e., process execution stops here.
- Π is the program as defined in Section 7.2.1.

In contrast to non-abstracted VHDL, the evaluation function *eval* returns one *or more* values based on the current environment and the expression.

This definition of an abstract process execution context differs from the concrete simulation only in the employed *eval*-function.

Based on the above definition of an abstract process execution context, the inference rule for a variable assignment can be defined as:

$$var_{abstract} \quad \frac{\Pi[\zeta] \Rightarrow v := expr; \qquad \Theta \vdash eval(expr) \ni u \qquad \Theta' = \lambda t. \begin{cases} u & \text{if } t = v \\ \Theta(t) & \text{otherwise} \end{cases}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta', next(\zeta), \Pi)} \tag{7.11}$$

If the statement $\Pi[\zeta]$ at the current program point ζ is a variable assignment whose right hand side evaluates to a result u, the new environment Θ' is created based on the input environment Θ but with u as the new value for variable v. Furthermore, the program counter PC is advanced by $next(\zeta)$.

The new abstract process execution context then is denoted by (Θ', ζ', Π). The difference between the inference rule for a variable assignment in Section 7.2.1 here is the potential nondeterminism embodied in the selection of u as a result of the expression evaluation. For example, assuming that the expression $expr$ actually evaluates to three values u_1, u_2, u_3, rule *var* would match three times yielding three different abstract process execution contexts $(\Theta'_1, next(\zeta), \Pi)$, $(\Theta'_2, next(\zeta), \Pi)$ and $(\Theta'_3, next(\zeta), \Pi)$.

Analogously, signal assignments are then defined by the following rule:

$$sig_{abstract} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow s <= expr; \qquad \Theta \vdash eval(expr) \ni u \\ \Theta' = \lambda t. \begin{cases} u & \text{if } t = \bar{s} \\ \Theta(t) & \text{otherwise} \end{cases}\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta', next(\zeta), \Pi)} \tag{7.12}$$

Due to the VHDL preprocessing described above, there are only simple expressions left in the conditions for `if` statements which deterministically evaluate to *true* or *false*. State splits can therefore never occur at such points. The inference rule for an `if` statement whose condition expression evaluates to *true* can then be defined like this:

$$tcond_{abstract} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{if}\ expr\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{end if}; \\ \Theta \vdash eval(expr) = true\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b_1), \Pi)} \tag{7.13}$$

As there are no changes of the environment, control is just passed to the first statement of the `then` block b_1. Analogously, the rule for a condition expression evaluating to *false* is

$$fcond_{abstract} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{if}\ expr\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{end if}; \\ \Theta \vdash eval(expr) = false\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b_2), \Pi)} \tag{7.14}$$

Here, control is passed to the first statement of the `else` block, namely b_2. Because these two inference rules only alter the control-flow position, they are equal to the rules (7.3) and (7.4) of the concrete simulation described in Section 7.2.1.

The preprocessing of the VHDL code not only simplifies the abstract simulation of conditional statements. Loops can be handled similarly:

$$tloop_{abstract} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{while}\ expr\ \texttt{loop}\ b\ \texttt{end loop}; \\ \Theta \vdash eval(expr) = true\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b), \Pi)} \tag{7.15}$$

Rule $tloop_{abstract}$ shows the effect of the execution of a `while` loop whose condition evaluates to *true* where the next rule ($floop_{abstract}$) is for loop condition expressions that evaluate to *false*:

$$floop_{abstract} \quad \frac{\begin{array}{c}\Pi[\zeta] \Rightarrow \texttt{while}\ expr\ \texttt{loop}\ b\ \texttt{end loop;} \\ \Theta \vdash eval(expr) = false\end{array}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, next(\zeta), \Pi)} \tag{7.16}$$

In both cases, there are no changes to the process execution context except the value of the current program pointer ζ which is advanced to the next statement, either the first statement of the loop body (*true*-case) or to the next statement after the loop (*false*-case). As for the `if` statement rules, the rules $tloop_{abstract}$ and $floop_{abstract}$ are equal to their concrete simulation counterparts (cf. rules 7.5 and 7.6).

Analogously to the concrete sequential process execution semantics empty blocks have to be covered by a special rule:

$$skip_{abstract} \quad \frac{\Pi[\zeta] \Rightarrow ;}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, next(\zeta), \Pi)} \tag{7.17}$$

After the last statement in a process, the special program counter ζ_{end} is reached and process simulation has reached a final state at that point and the process suspends. This state transition is covered by the rule $stop_{abstract}$.

$$stop_{abstract} \quad \frac{}{(\Theta, \zeta_{end}, \Pi) \rightharpoonup (\Theta, \zeta_{sus}, \Pi)} \tag{7.18}$$

The program counter for the process is then updated to ζ_{sus}. This rule is equal to rule (7.8).

Abstract Simulation Semantics

As for the simulation of a concrete VHDL model, the simulation semantics, i.e., the iterative application of the abstract sequential process execution rules as well as the effect of VHDL delta cycles, need to be defined. For this, a global abstract simulation context is required.

Definition 7.7 — Abstract Simulation Context:
The *abstract simulation context* of a VHDL model is the tuple $(\Xi, \boldsymbol{\rho})$ where

- $\Xi = \{\Theta_1, \ldots, \Theta_n\}$ is a set of environments where each environment Θ_i is defined as in the definition of the abstract process execution context in Section 7.3.2 and

- $\boldsymbol{\rho}$ is a map from process labels $\mathbb{L}$ to processes.

  As for the definition of the simulation context in Section 7.2.1, a process $\boldsymbol{\rho}(l)$ is a tuple $(\zeta_l, \Pi_l, \omega_l)$ where ζ_l is the program counter and Π_l is the program, respectively. And ω_l is a set of signal names representing the sensitivity list of $\boldsymbol{\rho}(l)$.

This definition just differs from the simulation context for the concrete case in using a set of environments Ξ instead of a single environment Θ. This difference is implied by the introduction of abstractions that introduce state splits (cf. Section 6.2.1). The resulting simulation tree (cf. Section 7.3.1) has multiple end nodes with different environments. All these environments have to be collected which is done by the set of environments Ξ.

Based on the abstract simulation context, process execution, i.e., the iterative application of the abstract sequential execution rules can be defined:

$$\frac{\begin{array}{c}\exists l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_l, \Pi_l, \omega_l) \wedge \zeta_l \neq \zeta_{sus} : \\ \Xi' = \{\Theta' \mid \Theta \in \Xi, (\Theta, \zeta_l, \Pi_l) \rightharpoonup^*_{seq} (\Theta', \zeta_{sus}, \Pi)\} \\ \\ \boldsymbol{\rho}' = \lambda(t \in \mathbb{L}). \begin{cases} (\zeta_{sus}, \Pi_l, \omega_l) & \text{if } t = l \\ \boldsymbol{\rho}(t) & \text{otherwise} \end{cases}\end{array}}{(\Xi, \boldsymbol{\rho}) \rightharpoonup (\Xi', \boldsymbol{\rho}')} \; exec_{abstract} \tag{7.19}$$

The simulation context $(\Xi, \boldsymbol{\rho})$ is changed into a new context $(\Xi', \boldsymbol{\rho}')$ by rule *exec* if there is at least one process $\boldsymbol{\rho}(l)$ whose program counter value is not equal to ζ_{sus}. The process execution context of each process $\boldsymbol{\rho}(l)$ matching this condition is updated for each environment Θ_i in the set of input environments Ξ by iteratively applying $(\rightharpoonup^*_{seq})$ the abstract sequential execution rules (7.11)–(7.18). By this, the rule *exec* advances the sequential execution of all active, i.e., not suspended, processes. As defined in Section 7.3.2, the abstract sequential execution rules are embedded in $\rightharpoonup_{seq}$. In contrast to the *exec* rule for the simulation of a concrete VHDL model as defined in Section 7.2.1, the difference of the abstract simulation context from its concrete correspondent implies that all existing environments ($\Theta_i \in \Xi$) have to be used for process execution.

As mentioned in Chapter 5, the semantics of VHDL has two levels and as in the simulation of a concrete model, VHDL delta cycles need to be incorporated in the abstract case, as well. After all processes have finished their execution and are suspended, all scheduled signal assignments for the current simulation time take effect. This might trigger a repeated execution of a process depending on its sensitivity list (cf. VHDL semantic description in Chapter 5) which is covered by the following rule:

$$\forall l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_{sus}, \Pi_l, \omega_l)$$

$$\Xi = \{\Theta_1, \ldots, \Theta_n\}$$

$$\forall i \in \{1, \ldots, n\} : \Theta_i' = \lambda t. \begin{cases} \Theta_i(\bar{s}) & \text{if } t = s \\ \Theta_i(\bar{s}) & \text{if } t = \bar{s} \\ \Theta_i(t) & \text{otherwise} \end{cases}$$

$$\Xi' = \{\Theta_1', \ldots, \Theta_n'\}$$

$$delta_{abstract} \quad \frac{\boldsymbol{\rho}' = \lambda(l \in \mathbb{L}). \begin{cases} (start(\Pi_l), \Pi_l, \omega_l) & repeat(\omega_l) \\ (\zeta_{sus}, \Pi_l, \omega_l) & \text{otherwise} \end{cases}}{(\Xi, \boldsymbol{\rho}) \rightharpoonup (\Xi', \boldsymbol{\rho}')} \tag{7.20}$$

For each environment Θ_i in the input set of environments Ξ a new environment Θ_i' is computed representing the application of delayed assignments, i.e., all scheduled signals become visible. The new set of environments then consists of all new environments Θ_i'. Furthermore, for each process $\boldsymbol{\rho}(l)$ having at least one signal in its sensitivity list ω_l changed ($repeat(\omega_l)$), the program counter ζ_l is set to the first instruction in the program $start(\Pi_l)$ which effectively restarts its execution. The definition of the predicate *repeat* has been taken from the dissertation of Hymans [Hym04], Chapter 4, where repeated execution of abstract processes is defined. In contrast to the concrete semantics where *repeat* realizes an inequality of future and current value of a signal, the abstract variant is required to cope with a potentially introduced domain abstraction. In this case, a custom operator on the new domain has to be used that implements the *repeat* predicate. If there are no signal value changes for a process, i.e., *repeat* evaluates to *false*, the process remains in its suspended state. Except this predicate, the rule $delta_{abstract}$ differs from the concrete simulation rule *delta* as that *all* existing environments have to be updated concerning delayed signal assignments. Everything else is analog to the simulation of a concrete VHDL model. If a signal value has changed in at least one environment Θ_i, all affected processes are restarted with all available environments. An optimization would be to restart a process only

with those environments that actually have caused the signal value change. That optimization is left for the implementation described in Chapter 10.

Using the operational semantics defined by the inference rules in the previous and this section, the effect of the abstract execution of all language constructs of the synthesizable subset of VHDL is defined and can be used to generate a simulator given an input VHDL model. This simulator is called an *abstraction-aware* simulator because it can cope with employed abstractions to the VHDL model.

7.3.3 Correctness and Soundness

Throughout this section, an abstract operational simulation semantics for VHDL has been introduced which forms the theoretical basis for the generation of pipeline analyzers for the aiT framework. For that purpose, the resulting abstract simulation must be provably correct with respect to the following criteria:

- model transformations must not change the timing semantics of the original model and
- the abstract simulation semantics has to be sound.

The first category refers to all employed model transformations (introduced in Section 6.4.3) that lead to the resulting timing model. In parts, the results of the used static analyzers (cf. Section 6.3.3) influence the concrete transformations and therefore their abstract interpretation need to safely over-approximate any possible concrete result. This is called a sound abstraction and corresponding correctness proofs are given by [Sch13]. In addition, the transformations itself must not change the timing behavior of the hardware design, i.e., they have to be semantic-preserving. Chapter 8 deals with this topic and introduces a timing model validation technique that makes use of methods from the area of formal functional hardware verification. Interval property checking is proposed to show the semantic equivalence of the original hardware design and the derived timing model. By this, the semantic-preserving property of the employed transformations gets verified.

The derived timing model is the input of the pipeline analyzer generation phase. But besides the timing model itself, the generated simulation is based on the abstract operational semantics which have been defined in this chapter.

It remains to argue about the soundness of this abstract semantics with respect to its concrete counterpart. In other words, the abstract semantics has to be a safe over-approximation to the concrete semantics. A soundness proof is not given here, but follows from the correctness proof that has been done by Hymans [Hym04]. He defines a concrete and abstract operational semantics whose formal representation is close to the one given here. In some parts, for example the *wake* predicate (*repeat* in this thesis) and the evaluation of VHDL expressions (*eval* predicate), some foundations from his work are taken over here. Therefore, his soundness arguments can be applied to the abstract operational semantics defined in this chapter and conclude the correctness of the abstract simulation framework.

7.4 Summary

This chapter starts with the definition of an *operational simulation semantics* for non-abstracted VHDL models. For this, a *process execution context* is defined that stores the current state of a process during its simulated execution. Inference rules are introduced that specify how this process context changes when executing VHDL language constructs. The simulation of a complete VHDL design is then supported by the definition of a *simulation context* that manages the execution context of each process in the design. Additional inference rules then specify the effect of repeated process execution (VHDL delta cycles) and when simulation time advances.

Based on these semantics for a concrete VHDL model, an *abstract process execution context* is derived together with corresponding abstract variants of inference rules for execution simulation. Due to the abstractions, nondeterminism is introduced into the model, so that the abstract process execution semantics supports the simulation of all possible executions in the presence of uncertainties. The result is a so-called simulation tree. Analogously to the concrete variant, an *abstract simulation context* with corresponding simulation inference rules is defined.

These abstract simulation semantics represents the basis for the generation of aiT-compatible pipeline analyzers for abstracted timing models as they are derived by the process invented in the last chapter.

8

Timing Model Validation

"All truths are easy to understand once they are discovered; the point is to discover them."

(Galileo Galilei)

8.1 Overview

Safety-critical embedded systems have to be developed under strict requirements. Before any commercial launch, the system itself as well as its development process have to be certified against an appropriate safety-standard like the DO-178B [DO92] for avionics systems. If the system has to fulfill strict timing constraints, the determination of a safety-guarantee concerning its execution time in the worst-case is mandatory. Using the aiT framework (cf. Section 4.3) for this task implies that the tool itself is validated under the terms of the same safety-standard. Among other things, this implies the validation of the used timing model. As proposed by this thesis, deducing a timing model from a formal hardware specification of the system to be examined ensures that the timing analysis correctly represents its actual behavior because the specification exactly defines the semantics of the synthesized hardware. This assumes correctness of synthesis results, i.e., the behavior of generated hardware circuits fulfill their formal specification. Correctness proofs for this result from a formal verification of the synthesized circuits against the design specification.

From this point of view, the model is correct by construction and it remains to be proven that the employed abstractions do not change or only safely over-approximate the timing behavior of the input model so that the resulting timing analysis is safe and correct. This chapter provides insights into the currently employed validation techniques for timing models independently from their particular development method, i.e., whether they are developed in a hand-crafted manner or derived from VHDL. Beyond this, a new validation method is presented mainly intended to be applied to timing models that have been derived from formal hardware specifications as described in Chapter 6. Here, the main idea is to apply the same techniques as in formal functional verification of hardware components.

8.2 Legacy Validation Approaches

This section provides an overview of currently employed methods for timing validation within the aiT framework. There are two approaches: one for examining the timing model of a processor core and another one for the validation of the prediction of bus accesses.

8.2.1 Validation by Performance Counter Monitoring

Some processors like the Freescale PowerPC 755/7448 provide so-called *performance monitoring facilities*, i.e., they are able to monitor and count predefined events such as processor clock ticks, cache misses, types of instructions dispatched, or mispredicted branches. This section describes a validation technique which exploits these facilities to compare the observable hardware events against the predictions of aiT.

The process of the performance counter based validation starts with selecting a specific set of performance counter events, e.g., dispatched instructions, L1 cache hits, completed branches,..., to be observed. Then an appropriate test program with user code is run on an evaluation board. Each time an instruction completes, the values of the counters for the selected events are written to memory. The collected information about performance counter values is aggregated and displayed in a matrix illustrating the changes of counter values over time. Additionally, the pipeline analysis has to be augmented in order to predict the performance counter behavior. Augmenting here means that corresponding code has to be written if the pipeline analysis has been developed manually. Having used the semi-automatic approach proposed in this thesis (cf. Chapter 6), the engineer "only" has to make sure that the performance counter update is not removed from the resulting timing model.

After an analysis of the task execution with aiT, the information about the performance counter values and their changes over the execution time can be compared between the pipeline prediction and measured traces. Important for the comparison is that the test programs have a fixed behavior, i.e., they must have a deterministic starting state. Then, depending on the test programs and the underlying hardware, one can try to examine the behavior of the different functional units as separately as possible and special combinations of instruction sequences which are assumed to have a different behavior as documented in the hardware manuals. Schlickling/Pister [SP09] give further details on the concrete usage of this approach.

This technique complements the trace-based validation described in the next section since the performance counters gather information about the internal processor state that cannot be observed on the system bus.

8.2.2 Validation by Trace Matching

Trace matching is a method developed in order to match aiT results – the predicted timing behavior – against bus traces determined via measurements on the actual hardware.

The idea is to let the pipeline analysis write predictions of the visible bus transaction signals during the abstract simulation of the task's execution. In parallel, bus traces of the concrete execution are recorded and compared to aiTs prediction.

In practice, this technique has been established with good results [Sch05]. Its effectiveness results from the fact that most industrial applications are dominated by their memory accesses regarding the execution time. Therefore, most execution time is spent within accesses to the memory hierarchy which can be traced using the above mentioned bus traces rendering trace matching to be the most important technique for timing model validation so far. All measured end-to-end runtimes must be lower than or equal to the computed WCET bound, i.e., no single under-estimation must exist. Furthermore, the order of the visible bus signals in the measured trace has to be found in the prediction. In other words, the measured execution path must always be represented by one of the predicted paths. But it must not necessarily contribute to the worst-case path since measurements do not provide a full coverage of the potential execution paths in the analyzed executable and therefore do not trigger the worst-case path in general (cf. Section 4.2.2).

Currently, this is the only way to compare the aiT prediction for bus accesses with the actual hardware behavior. Although the idea sounds pretty simple, the main problem in this technique is the comparison of the measured hardware trace with the *execution tree* (cf. Section 7.3.1) in the timing prediction. This graph can be big for real world applications, consisting of millions of nodes. Although the comparison itself has been efficiently automated with tools, the needed human effort is still high, speaking of man months depending on the complexity of the examined hardware architecture. If the timing model has been built on documentation, such a trace validation exhibits differences between the model and the actual hardware's behavior. Needed reverse engineering by suitable test cases and drawing the right conclusions to correct the model is a difficult task. Additionally, the traces might have recorded asynchronous events like DMA accesses or dynamic memory refreshes. They are not incorporated in the timing model and therefore lead to

mismatches in the automatic trace comparison. These are examples for the high human effort.

8.3 Formal Functional Hardware Verification

Up to the nineties, the state-of-the-art in hardware verification has been a purely simulation-based approach, i.e., hardware simulators were used to obtain confidence about functional correctness of hardware circuits. This requires the corresponding simulation test benchmarks to achieve a good coverage of the hardware model which has been tried to achieve by stimulating potential inputs.

But the simulation models are running with much slower speed (factor of about one million) than the actual hardware. In consequence, only a small portion of the lifetime of a chip can be covered by this technique within an acceptable time span. So, a high probability remains that specification errors remain undetected because it is clear that not all corner cases of the functionality can be explored with this methodology. This problem is known as the *verification gap*.

Formal verification methods have been introduced in the early nineties in order to bridge this gap and are therefore often called *gap-free verification*. First commercially available methods were the so-called *assertion-based formal verification (ABFV)* [Jas11, Ave11, Syn11, Rea11] which are based on traditional model checking methods [CGJ+00]. The goal was to state assumptions on the behavior of a hardware design which should then be proven to be valid using a model checker. But traditionally, these approaches have complexity problems if applied within an industrial context, i.e., used for the verification of assertions on complex processor systems. And with the growing development rate of modern processors and systems, the situation got even worse. As stated by Bormann [Bor09], the development of these methods has not kept pace with "Moore's Law" [Moo65]. Often, hardware manufacturers are already satisfied with the formal verification of small components in their designs and neglect the interaction between different modules of a design.

Recent research of the last years has introduced new techniques for performing formal functional verification of hardware circuits against a formal specification that efficiently have overcome the performance problems with early ABFV approaches mentioned above. A recent candidate is described in

the next section because it can also be employed for the validation of timing models which were derived from formal specifications.

8.3.1 Interval Property Checking

Interval property checking (IPC) is a formal verification method that can be used for different aspects of functional verification of hardware circuits. The idea of the approach is to develop a "structure" of what has to be verified [Bor09]. And this structure can be derived from the operations of a hardware circuit, e.g., the hardware instruction set. The instruction semantics summarizes processes of the particular circuit over a defined range of time. An execution of an instruction triggers a dedicated effect (although dependent on the execution history, i.e., the current hardware state) within the processor so that the union over all these effects, i.e., all available instructions, then specifies the complete functionality of the hardware design. In order to define the semantic effect of instruction execution on the design, one can use temporal properties which can be derived either from the ISA (instruction set architecture) documentation as well as from timing diagrams which are usually available from the processor manufacturer.

A so-called *property checker* then tries to prove that the hardware implementation – a formal specification like VHDL, Verilog or a similar specification language – fulfills the given properties. The difference to bounded model checking (BMC) [BCC+03], a variant of model checking also used for formal verification of hardware designs, is that the notion of time is built into a property, i.e., a property can specify that certain state changes has to happen within an interval of time $[t, t + l]$. Property checkers then start simulation at time t with all possible input states. In contrast to that, BMC would start at time 0 and is only able to simulate up to a point $n = t + l$ in the simulation time. Simulation from the system start then introduces additional complexity.

As usual for model checking techniques, the computed output of a property checker is either the proof that the design satisfies the given properties or a counterexample where at least one property does not hold. This simplifies the traceability of the potential error. Potential here means that the checker analyzes the model independent from the input state, so there might be false negatives with that method. For a computed counterexample, it has to be checked whether the particular input state is possible for that situation. False

negatives could be avoided by intersecting them with the results of a reachability analysis that computes possible input states. The reachability analysis could automatically generate additional properties which then exclude unreachable input states, at least for simple reachability constraints. Where the automatic generation of such properties fails, the verification engineer has to manually exclude unreachable states by hand-crafted specification of additional constraints [Bor09]. Then, the final verification problem effectively has been reduced to a SAT-problem, for which powerful solvers exist.

8.3.2 Completeness

As the invention of IPC for formal functional hardware verification seems to render the verification of even complex circuit designs feasible, there is another problem to be addressed: *completeness*. The given set of properties represents another way of specifying the functionality of the design to be verified. So it has to be assured that the properties fully cover all hardware requirements. In other words: a property set for a specific hardware design is called *complete* if and only if the output signals of the design can be determined uniquely for all possible inputs.

To reach completeness, a property set has to be checked by a completeness checker as it has been developed by Bormann [Bor09, Cla07]. IPC combined with such a completeness checker is then called *Complete Interval Property Checking (C-IPC)* and a complete set of properties for a hardware design then forms an abstract specification of the circuits compared to its formal specification in VHDL or Verilog.

Complete interval property checking has shown its applicability on modern and large industrial hardware designs like the Infineon TriCore2 processor, Infineon's [Inf] next generation of microcontrollers which has been published by Bormann [BBM+07]. The theoretical foundations have been developed within the research projects *Verisoft XT* [Ver], Herkules [Her12], Valse and Valse XT. The technology has been embedded into a commercially available tool – *OneSpin 360MV* – which has been developed by *OneSpin Solutions GmbH* [One11].

Recent research has invented methods for the derivation of abstract models from a complete set of properties for a hardware design. With such models, global liveness and safety properties of the design can be proven and temporal

abstractions can be invented [USB+10]. Additionally, the theory has been adapted for the verification of weakly programmable IPs [LWS+10].

8.3.3 Example Property

To give an overview of the formulation of properties for hardware circuits, this section presents a rough description of an example taken from [Bor09]. A given SDRAM chip is first described and a sample property will be given afterwards.

The chip is connected to a specific memory controller which is located on the main system board. Data transfers between the processor and the memory controller are then driven on the system bus. As described in Section 3.3.4, the memory cells in SDRAM chips are arranged in matrices and due to this structure, the address of a single memory cell consists of a row address part and a column address part. Chip transactions between the memory controller and the SDRAM chip are then composed by different control signals:

- *Commands* encode the concrete action to be performed on the chip. Table 8.1 on page 176 shows the most important SDRAM commands.
- The *address bus* contains the address of the memory cell to be accessed.
- The *data write bus* contains data to be written in case of store operations.
- The *data read bus* contains read data delivered by the chip in case of load operations.

Often the control logic is more complex than described here and uses additional signals. They have been left out for simplicity and without loss of generality.

Analogously to the above described commands, the SDRAM manages a state which can be modeled by a finite state automaton. A simplified state machine is shown in Figure 8.1 on page 176. Starting with the state IDLE, it depends on the concrete address of an access. If the corresponding row is already open, the current state is changed to READ or WRITE for the particular type of the access. Otherwise, the row has to be activated first which is handled by the `precharge` and `row activate` commands in state PRE_ACTIVE. After having opened a row, the state machine enters state READ or WRITE. There, the actual access is handled by issuing the commands `read` or `write`, respectively. If the access is finished and does not cross any row boundary

(due to the particular access width), the control returns to the initial state IDLE. For a row-crossing access, the next row has to be opened and the current state is changed to PRE_ACTIVE, again.

The code in Listing 8.1 on page 177 shows a sample property for a read operation into a currently not activated row of the matrix. First, the input assumptions are defined in lines 3–7. They state that the input SDRAM controller state (at time t) is IDLE and that there is a read request whose row address part is not equal to the previously used row address. In lines 9–26, it is specified what the property checker should try to prove: First, the controller state at time $t+1$, $t+5$ and $t+9$ should be PRE_ACTIVE, READ and IDLE respectively. Additionally, the most recently accessed row address information should have been updated. Then, the different SDRAM commands are sent to the chip. This is indicated by the assignments to the signal `sd_ctrl` which have to be proven by the property checker. Furthermore, at time $t+3$ and $t+5$, the address transferred to the chip should be the row and the column address of the memory cell respectively. At time $t+8$, the requested data should have been transferred from the chip to the memory controller over the data read bus. This is modeled by the `rdata` signal. The `nop` commands used at time $t+2$, $t+4$ and during $[t+7, t+9]$ are needed because the concrete hardware sometimes requires some cycle delays between the transmission of subsequent SDRAM commands. Named signals and identifiers are directly coupled with their correspondents in VHDL.

This property language allows to express the functionality of the SDRAM interface including the timing behavior. In Listing 8.1 on page 177, the internal state of the SDRAM is specified for each clock cycle. The example presented in this section should only give an impression of the formulation of properties for hardware designs. For further details, please refer to the original example from Bormann [Bor09] that contains more details which were left out here.

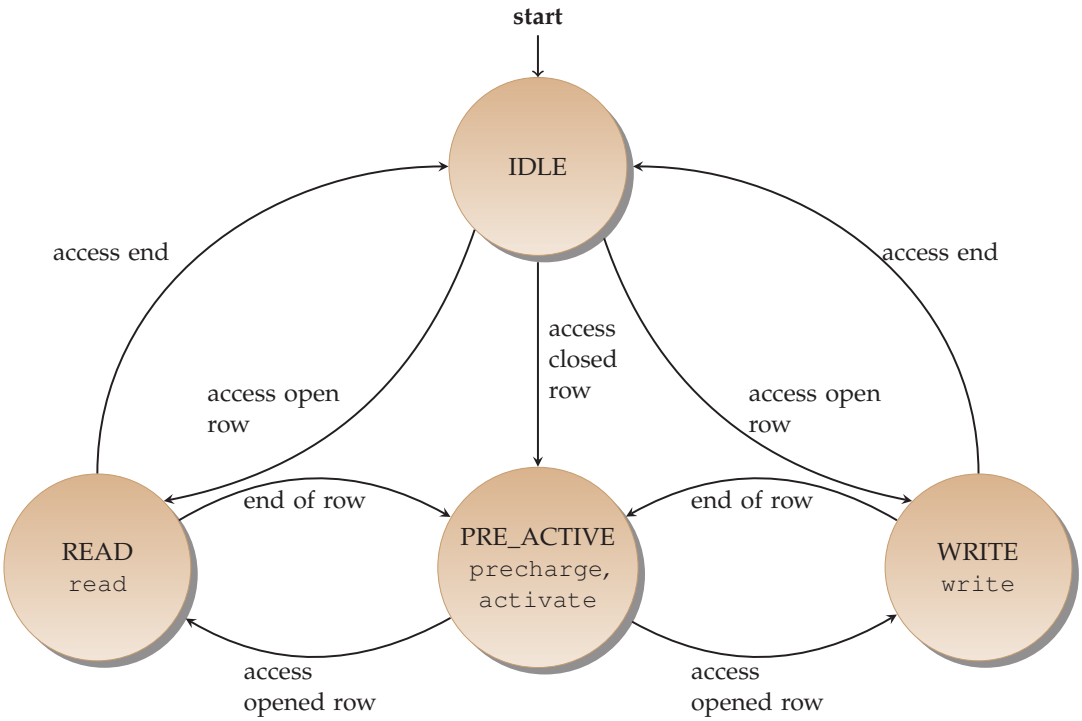
8.4 Property Checking Based Timing Validation

This section introduces a new method for the validation of timing models that have been derived from formal hardware specifications as described in Chapter 6 on page 109. Such models have been transformed with the goal to only represent the actual timing behavior of the hardware design.

Table 8.1 – SDRAM chip commands

Command	Description
`row activate`	Activates a line in the matrix for subsequent accesses to it.
`read`	Performs a read operation from the addressed memory cell within the currently active row.
`write`	Performs a write operation into the addressed memory cell within the currently active row.
`precharge`	Closes the currently active row.
`nop`	No operation. This command is needed to implement needed delays after the other commands.
...	

Figure 8.1 – SDRAM state machine



Listing 8.1 – SDRAM read sample property

```
1  property read_new_row is
2
3  assume:
4  at t:                  state = idle;
5  at t:                  request = '1';
6  at t:                  rw = '1';
7  at t:                  address /= last_row;
8
9  prove:
10 at t+1:                state = pre_active;
11 at t+5:                state = read;
12 at t+9:                state = idle;
13 at t+9:                last_row = prev(row(address));
14
15 during [t+1, t+7]:     ready = '0';
16 at t+1:                sd_ctrl = row_activate;
17 at t+2:                sd_ctrl = nop;
18 at t+3:                sd_addr = row(address);
19 at t+4:                sd_ctrl = nop;
20 at t+5:                sd_ctrl = read;
21 at t+5:                sd_addr = col(address);
22 at t+6:                sd_ctrl = stop;
23 during [t+7, t+9]:     sd_ctrl = nop;
24 at t+8:                rdata = prev(sd_rdata);
25 at t+8:                ready = '1';
26 at t+9:                ready = '0';
27
28 end property;
```

Therefore, it has to be shown that the transformed model still represents the same timing as the original model.

As the property checkers operate on the formal specification code (VHDL, Verilog or similar), the transformed model cannot be examined directly because the derivation process' internal representation of the model is the Control-Flow Representation Language (CRL) as described in Section 6.3.1. A conversion that reconstructs VHDL from the internal CRL representation has to be done first. Chapter 10 on page 203 describes the implementation of this reconstruction within the code generation phase. This way, a property checker could then compare both models (original and transformed one) against a specification (given as a set of properties). The property set created for the original design has to hold for the transformed model, too.

This approach still is in an early development process as a cooperation between Saarland University and the *Electronic Design Automation Group*, University of Kaiserslautern which have a rich experience and knowledge about formal functional verification in general and the application of interval property checking in the presence of abstractions specifically. In the next section, the current state of this cooperation is described and summarized with an outlook on potential future work in Section 8.6.

8.4.1 Current State

The above mentioned needed property set can be either specifically formulated for this purpose or taken from the properties used for a functional verification of the hardware design where the latter variant has the advantage of being complete (cf. Section 8.3.2). By this, the side product of a complete formal verification on the original hardware model additionally contains the timing behavior of the design although this information usually might be formulated implicitly. A problem arising in this approach are the transformations performed to derive the timing model from the original one. Depending on the concrete transformations (cf. Section 6.4.3), a lot of the properties will not hold in the transformed model. For example, the removal of pure functional code (cf. Section 6.5.2 on page 140) like the internal behavior of an arithmetic-logical execution unit (ALU) will most probably break those properties that check the ALU's semantic. In other words, functional attributes of the model might get lost during the transformation and abstractions done during the timing model derivation procedures. Properties then have to be adjusted or possibly be removed so that only the timing behavior still is reflected by them.

Assuming a performed domain abstraction (cf. Section 6.5.2), needed implementations of new operators for the target domain should be specified directly in VHDL in order to better support the usage of a property checker. If this is not possible or intended, additional properties have to be formulated. They need to specify the particular effect of the newly introduced operators on the global state of the design because the property checker needs to know the concrete semantics.

Another assumption which has to hold for the whole derivation process is that there is no change in the input signals between the original model and the transformed one. If this does not hold, a mapping between old and new signals has to be provided.

Two proof-of-concept equivalence checks exist so far. They are described in the following.

Proof-of-concept: Address Abstraction

The goal here was to show that an address abstraction as described in Section 6.4.2 can be proven not to change the timing behavior of the system in general. It has been manually performed for parts of the load/store unit of the pipelined DLX design (described in Chapter 11). To recall the effect of an address abstraction: concrete addresses a in the design get replaced by the usage of intervals of addresses of the form $[lb, ub]$ where lb (ub) represents a lower (upper) bound on the potential addresses for a program point in the specification code.

To show the equivalence of original and transformed model, each address a in the original model has to be within the bounds of lb_a and ub_a in the abstracted model, i.e.,

$$\forall a : lb_a <= a <= ub_a.$$

This equivalence has been proven using the OneSpin equivalence checker [One11].

Proof-of-concept: Reset Analysis

Here, the goal was to validate the results of the reset analysis (described in Section 6.5.1) because they build the basis for further analysis like the assumption-based model refinement (cf. Section 6.4.1) which end up in code removals. The general idea is to formulate the analysis results as a property and feed it to the OneSpin property checker. Formulated assumptions are that the signals involved in the hardware reset are appropriately activated.

By this, the correctness of the reset analysis could be shown for the pipelined DLX design.

8.5 Summary

This chapter deals with the validation of timing models employed in the aiT framework, i.e., the presentation of approaches that can demonstrate

the correctness of such models. Two existing legacy methods are presented where both base on the comparison of execution time predictions with event sequences that have been recorded during a real execution of the analyzed task. The first is called *performance counter validation* and focuses on the processor core's performance counters (if supported). Events like instruction retirement, cache hits/misses, number of instruction dispatches, etc. are traced and compared to the corresponding prediction of the model.

This technique complements the second presented legacy method, *trace validation*, which is concerned with visible bus transaction signals. They are recorded using a logic analyzer so that the produced execution trace can be compared to the corresponding prediction from the timing model. As the execution time of most safety-critical embedded control software is memory-bounded, the bus transactions can be used to validate the most important part of the timing model: the memory management subsystem.

The formal verification technique *interval property checking* can be used to prove temporal attributes of a hardware design and can therefore be applied as a validation technique for safety and correctness properties of timing models derived from formal hardware specifications in VHDL or Verilog. Therefore, the second part of this chapter presents how to transfer this technique appropriately to the validation of timing models. Employed model transformations as presented in Chapter 6 are shown not to change the timing semantics of the model in an unsafe manner, i.e., not to introduce any under-estimation. Despite the fact that the presented methods are in an early phase of development, the existing equivalence proves which have been sketched, show their applicability for a timing validation in general. Proof-of-concept equivalence checks exist for the output of the reset analysis as well as the hand-crafted address abstraction.

8.6 Future Work

In addition to the description of the current state of development, there are still open issues regarding the application of interval property checking for the timing validation of semi-automatically derived models:

- The correctness of the *assumption-based model refinement* procedure (cf. Section 6.4.1 on page 122) has to be proven. Analogously to the validation of the reset analysis, the correctness can be shown by reformulating

any analysis result as properties which are then feed to the OneSpin property checker. Corresponding assumptions given as input for the refinement analysis are incorporated into the property (as property assumptions) so that an equivalence result of the checker is directly related to them. For a better user convenience, the conversion from refinement analysis results to corresponding IPC properties should be done automatically.

- *Correctness* of all *types of transformations* (cf. Section 6.4.3) has to be shown. The idea is to specify the effect of a transformation within a property. Removal of functional code from an arithmetic-logical unit (ALU) in a processor pipeline might be shown "correct" if the whole unit is treated as a black box. Its effect within the processor pipeline is described by a property which additionally contains the specification how long the ALU would be occupied by an executed instruction. Similar equivalence checks have to be performed for any other transformation applied to a model. An interesting point then might be actual over-approximations introduced by single transformations and abstractions. Used properties then not only have to show that the timing semantics of the original model still hold. It must be proven that each approximation is *safe*, i.e., all abstract values computed by the transformed model cover any possible concrete value.

The overall goal of showing that the timing behavior of the design is not changed by any transformation as described in the validation strategy, can then be achieved by showing that all properties of a complete set are still satisfied by the transformed model modulo the breakage of properties which check internal not timing relevant behavior. But this has not been formulated in any way so far. Additionally, the equivalence checks are only valid for the specific hardware design for which they have been done.

A more "off-topic" idea is to examine the potential extraction of timing relevant information from a given complete property set for a given hardware design. As both techniques have been designed and used for the determination of safety guarantees, it would be interesting to see whether such properties could complement the derivation of timing models from the hardware specification.

9

Timing Predictability

“You see things; and you say: "Why?" But I dream things that never were; and I say: "Why not?"”

(George Bernard Shaw)

9.1 Overview

Computer systems have dramatically changed during the last decades, especially for embedded systems as mentioned in Chapter 3. Despite innovations like virtual memory, high-level programming languages or memory management, execution times only played a role for marketing issues like "this new system performs much better on the SPEC benchmarks. ...". Unfortunately, the timing behavior of an embedded system strongly affects its functional behavior.

Nowadays, more and more people are becoming aware of this problem [Lee09]: it is crucial for a correct behavior of an embedded system to be able to precisely predict the execution time of its tasks. But the developments of the last years were so-called "ill-suited" for real-time systems [GRW11]. Architectures have become more and more complex, neglecting the resulting complexity of their timing-behavior analysis, and thereby introduced a lot of sources of uncertainties. These sources are situations where a static timing analysis cannot precisely predict a hardware decision. As described by Grund [GRW11], two lines of research are currently visible:

- improve existing timing analysis techniques to better cope with recent architecture developments or
- influence future system design for a better predictability especially regarding the timing behavior.

Between both worlds, there is no agreed formal definition of the term "timing predictability". So far, it is rather used subjectively based on the encountered difficulties when performing a timing analysis. As there are a variety of variables influencing architectural predictability, Grund et al. [GRW11] proposes a metric for comparing the predictability of systems based on a so-called *predictability template* instead of a fixed definition. The template is based on the property to predict (for example the execution time), the sources of uncertainty and the quality measure.

Intuitively summarized, timing predictability can be defined like this:

Definition 9.1 — Timing Predictability:
In order to compare the *timing predictability* of two architectures for a given program, the ratios between the best-case and worst-case execution

time (BCET and WCET[1]) over all possible starting system states and inputs have to be determined.

Remark:
Intuitively, the gap between the result of a WCET analysis and the concrete WCET of the analyzed task would represent a more precise definition of the hardware's timing predictability. But as the concrete WCET of a task cannot be computed in general (cf. Section 4.1), such a metric is not applicable in practice.

The gap between the BCET and WCET results from the above mentioned uncertainties of timing effects that might not even occur in the concrete execution but cannot be excluded from the static analysis point of view [RGBW07]. These uncertainties can be caused by hardware (*state-induced*) or by software (*input-induced*) and can be categorized according to [WFC+09] into:

- *sequential control flow*:
  Different input might lead to different control flow.
- *interleavings of concurrent control flow*:
  Component concurrency lead to multiple possible interleavings to shared resources like multiple bus masters.
- *architecture flow*:
  Due to the two points above, there are different input system states for the abstract simulation leading to a large number of paths that have to be analyzed. That effectively results in a large state space.

Additionally, the gap between the actual WCET of a task and the computed timing bound is an important factor for the predictability. Again, it is affected by both the timing predictability of the underlying hardware and its features as well as precision of the employed timing analysis.

The goal of this chapter is to give an overview of the predictability of commonly used hardware features with respect to the timing analysis employed by the aiT framework (cf. Section 4.3) which is described in the next section. Additionally, there is an evaluation in Section 9.3 about the impact on VHDL language constructs on the precision of a timing analysis which has been derived from such a model as described in Chapter 6.

[1] cf. Chapter 4

9.2 Timing Predictability of Hardware Features

As mentioned in Chapter 3, architecture development in the last decades has lead to more and more parallelism in the system due to the invention of complex processor pipelines, caches (of different levels), buffers and queues for memory accesses and suchlike features. Consequently, the execution time of such complex architectures can no longer be directly deduced from the documented instruction latencies. Therefore, only counting and accumulating these timings is no longer feasible [KWH+08]. Actual instruction latency is only computable by examination of the execution history. Larger queues or more buffers then lead to longer lasting influences of past events to the time of the currently executed instruction. As detailed below, this introduces uncertainties increasing the state space to be explored by the timing analysis.

Furthermore, more complex architectures exhibit timing anomalies (cf. Section 6.2.2) which affects the predictability, as well. Regarding this topic, architectures can be classified into three categories (taken from [WGR+09]):

- *Fully timing compositional architectures*:
  The (abstract model of) an architecture does not exhibit timing anomalies. Hence, the analysis can safely follow local worst-case paths only. One example for this class is the ARM7 reference architecture [ARM00]. Actually, the ARM7 allows for an even simpler timing analysis. On a timing accident all components of the pipeline are stalled until the accident is resolved. Hence, one could perform analyses for different aspects, e.g., cache, bus occupancy, separately and simply add all timing penalties to the best case execution time.

- *Compositional architectures with constant-bounded effects*:
  These exhibit timing anomalies but no domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant number of cycles to the local worst-case path. The Infineon TriCore TC1797 [Inf07] is assumed, but not formally proven, to belong to this class.

- *Non-compositional architectures*:
  These architectures, e.g., the PowerPC 755 exhibit domino effects and timing anomalies. For such architectures, timing analysis always have

to follow all paths since a local effect may influence the future execution arbitrarily.

Nowadays not only the complexity of modern processor pipelines imposes difficulties for timing analysis in general. Recent trends are going towards the introduction of multi-core systems into safety-critical embedded systems. Most of such systems on the market have been originally developed for desktop or server systems. Their design uses big second and third level caches which are often shared among the processor cores. Up to now, the problem of performing a precise timing analysis for multi-core systems actually is not solved, but the research community is already addressing it [WFC+09, CFG+10].

In the following sections the timing predictability of certain hardware features is described. These features have been developed for increasing the average case performance of the system ignoring its effects on the worst-case behavior. If not mentioned otherwise, all statements below apply both to single as well as multi-core architectures.

9.2.1 Processor Pipelines

Processor pipelines have become more and more deep and complex in the last years. On the one hand, deep execution pipelines (cf. Section 3.4) have been introduced and some execution units got its own "sub-pipeline", i.e., the unit has its own pipeline in order to overlap the execution of consecutive instructions. This was done mainly for units executing long-lasting instructions like a floating-point or a load/store unit. Additionally, performance improving features have been developed as for example prefetching, branch folding, branch prediction, delay slots, fast data forwarding and shortcuts, superscalarity, out-of-order execution, speculative execution and store gathering among others (cf. Section 3.4.3).

Assuming the availability of detailed documentation or specification about the behavior of all these features, they all are good predictable, i.e., its timing effect in principle is well understood and can be modeled by a static analysis. But such features typically increase the potential parallelism within the processor pipeline. At a first glance, this "only" induces a higher space consumption of a single system state. But often, interferences between combined features arise that can lead to a significant increase of the state space to explore in case of uncertainties. For example, the IBM PowerPC

750 [IBM06] has a lot of buffers and queues within its load/store unit which allow a bunch of parallel pending memory requests. One application is: if there is a store and a load operation to the same memory location, the data to be written by the store operation can be fast forwarded to the requesting load operation increasing the overall system performance. Assuming that there are imprecise value analysis results for these load or store operations, the timing model cannot actually precisely decide whether the fast data forward happens or not. So, both possibilities have to be examined so that the timing model is getting more vulnerable to imprecise information from the value analysis because the analysis cannot statically exclude some of the possibilities. Combining such effects lead to an exponential increase (in the number of such combined features) of the possible execution paths.

That is just one example how complex pipeline control structures might increase the computational complexity of the pipeline analysis within the aiT framework. The result is twofold: on the one hand the analysis time increases because of a larger system state. And on the other hand, the computed time bound might get more pessimistic because of uncertainties rendering the whole architecture to be less predictable as best-case and worst-case diverge.

9.2.2 Caches

The predictability of caches (cf. Chapter 3) is dominated by the predictability of the employed replacement policy [RGBW07] where least-recently-used (LRU) and its pseudo variant (PLRU) are the best predictable ones. A sophisticated comparison of the predictability of caches can be found in publications by Grund and Reineke [Rei08, GR10]. Certainly, the worst predictable caches are the ones with a random replacement policy which are just impossible to predict.

For caches smaller than 32KB, the LRU policy can be seen frequently in embedded system architectures. But because LRU is costly, caches with greater sizes usually are supplied with PLRU replacement. Second level caches which are often much bigger, e.g., 1 MB for the Freescale PowerPC 7448 [Fre05a] and IBM PowerPC 750 [IBM06], are then often implemented with some random replacement strategies.

Another factor deciding the predictability of caches is the write policy which configures the concrete effect on stores. There are two write policies available:

write-through where a store is directly written to the next level in the memory hierarchy and *write-back* where the data is written into the next hierarchy level if the concrete memory cell is evicted from the cache. Where the write-through policy can be predicted precisely, this is not the case for write-back because it then has to be known when exactly a memory cell is evicted from the cache. But this information is not always precisely available as it depends on the general precision of the cache analysis [RGBW07].

In general a worse predictable cache means that the corresponding cache analysis could not precisely predict all cache hits and misses. The pipeline analysis then needs to follow both parts which on the one hand leads to an increase of computational complexity and on the other hand results in more pessimistic time bounds for memory accesses which definitely hit the cache in the concrete execution.

9.2.3 Buses

In general, buses are clocked with a lower frequency than the processor core. The main system bus connecting the core with the chipset[2] usually runs at ratios between 1:2 and 1:8 compared to the CPU clock. A static analysis needs to incorporate this by assuming all possible clock offsets between core clock and system bus clock for the starting system states because the concrete offset is statically not known.

The number of possible displacements of phase between CPU- and bus clock signal is bounded, i.e., at the start of a CPU cycle the bus cycle can only be in a finite number of states. For example, if the processor core operates at $f_{CPU} = 100$ MHz and the bus at $f_{BUS} = 25$ MHz, there are 4 different states. In general, the number of states is determined by:

$$\text{bus-clock-states} := \frac{f_{CPU}}{\gcd(f_{CPU}, f_{BUS})}$$

The smaller the number of bus-clock-states the more efficient is the micro-architectural analysis. Note that for integral ratios of CPU- to bus-frequency the formula simplifies to f_{CPU}/f_{BUS}. It might be beneficial to use integral ratios; even if a close-by non-integral ratio would have a higher average-case performance.

[2] usually called Northbridge

Buses can also be classified as parallel, e.g., *SCSI*, or bit-serial, e.g., *USB*, buses. Parallel buses carry data words in parallel on multiple wires, bit-serial buses carry data in serial form. Because of the separation of addresses and data on parallel buses, the execution of consecutive memory accesses can be overlapped, i.e., for two accesses, the address phase of the second access can be overlapped with the data phase of the first access. This is called *bus pipelining*. Although pipelined buses are tractable for static timing analysis in principle, they increase the state space with rising pipeline depths. However, the pipeline depth depends on the potential parallelism in the processor core regarding memory accesses. Both, the bus architecture and core must match here, i.e., it does not make much sense to use a bus architecture that can process more requests in parallel than the connected processor core is able to start.

Instances that can request accesses to the bus, are called *bus masters*. On simple systems there is only one bus master since there is typically one CPU that requests the bus. This scenario can be modeled because the timing behavior is deterministic. The more masters a bus has the more difficult it is to analyze the traffic on the bus and the less precise will be the bounds on latencies that can be guaranteed. There are multiple methods to handle the arbitration between multiple bus masters [WGR+09].

9.2.4 Main Memory

The predictability of memory accesses mainly depend on the accessed type of memory (neglecting the load/store unit part in the processor core). In safety-critical embedded systems, mainly the following memory types are used.

Static Random-Access-Memory (SRAM)

SRAM features fixed access latencies independent to the execution history and is therefore perfectly predictable. Actually SRAM often is the type of memory which is used within level one or two caches. Because of the rather high manufacturing costs, SRAM is unfortunately not used for bigger main memories.

Dynamic Random-Access-Memory (DRAM)

In contrast to the static ram, DRAM is more difficult to predict mainly because the underlying memory cells need to be refreshed from time to time[3]. These so-called *DRAM refreshes* are asynchronous events which cannot be predicted precisely as it is not known when exactly they happen. An approach for this is to analyze the computation time ignoring refreshes at all together with adding a safety margin to the computed WCET. The margin then can be determined based on the refresh frequency and the worst-case duration of one single refresh[4] so that the DRAM refresh costs are amortized over time. This is possible because such a refresh has no side-effect to the state of the system with one exception: it closes all memory pages such that a subsequent access to a previously open page then results in a page miss that would have hit otherwise. Therefore, the accounted refresh costs need to include the time to open a page. This represents an overestimation for the case that a subsequent access would not hit a previously opened page.

Such an argumentation is not valid for cache misses. Depending on the hit/miss classification, a memory request is differently routed through the processor pipeline, so that cache miss penalties cannot be incorporated after the analysis. Instead, they must be taken into account locally, i.e., when the access happens during the simulation. And if the access cannot be classified precisely, both possibilities have to be taken into account.

Another asynchronous event is a so-called *direct memory access (DMA)* where input/output performance is increased by a factor of about 10 without stalling the processor core until the access is finished. If the frequency of such DMA accesses is known, their time costs can be approximated in a similar way as described for the refreshes.

Due to the design of DRAM chips (segmentation into different memory banks, pages, ...) the corresponding memory controller's design can get complex. Internal queues and buffers with a complex update logic increase the state space for the pipeline analysis in a similar way as the above mentioned complex pipeline structures of the processor core. Of course, an exhaustive documentation about the internal behavior of the memory controller is a necessity for a successful static timing analysis.

[3] The actual interval depends on the concrete memory chip.
[4] Both parameters are known from the chip manufacturer.

9.2.5 Peripheral Devices

Peripheral devices are connected to the system board via external buses like PCI [PCI98], CAN [CAN03], etc. As such buses generally have their own clock domain, such accesses are processed by a memory controller which is divided into two parts. One part that is clocked with the frequency of the system bus clock and another part that is clocked with the frequency of the external bus. In contrast to the displacement between the processor core clock and the system bus clock (cf. Section 9.2.3) where the system bus clock is generated from the core clock signal, these external clocks often are not produced from the same clock generator and therefore not synchronized to the core clock.

The static analysis has to incorporate this when predicting the access latency of a memory access to such a peripheral device. All possible displacements between a best case starting condition where communication on the external bus is directly possible, and a worst-case starting condition where the memory controller needs to wait until communication on the external bus is possible. In addition to that the system designer has to provide minimal and maximal access latencies for the actual time spent on the device. This approach can be used for a safe time prediction but can also be costly in terms of computation time and space consumption. The number of system state splits (cf. Section 6.2.1) here depends on both the difference of the system and external bus clock frequencies as well as the given access latency interval. The computational complexity might grow rather fast if the difference between the interval bounds is large. Usually, this difference should be around 10 at most although this strongly depends on the remaining complexity of the whole architecture.

9.2.6 System Configuration

Modern systems are highly configurable, i.e., certain provided hardware features can be adjusted in their behavior or even disabled. Therefore the chosen hardware configuration strongly affects the timing predictability of the whole system. In the following, design hints and advices for possible values of system configuration parameters that increase the system's timing predictability are given. The list is not meant to be exhaustive and has been based on experiences with timing analyses of industrial code. For a more

thorough treatment, see the detailed discussion by Cullmann and Gebhard [CFG+10].

- *Unified caches*:
  If configurable, unified caches (cf. Section 3.3.3) should be avoided as much as possible because their timing analysis is more difficult. For example, some PowerPC processors with unified caches can be configured to use them for instructions or data exclusively or their ways can be separately reserved for instructions or data.

- *Cache locking*:
  Cache locking effectively disables the eviction of any cache content, i.e., no new tags (cf. Section 3.3.3) in the cache will be allocated by memory accesses and can only be manipulated by special cache instructions. Read accesses to data that has been filled into the cache will result in a cache hit. Otherwise, they are handled by the next memory hierarchy level like the L2 cache or main memory. Store operations modify the local copy in the cache and are then forwarded to the next memory hierarchy level if the store either misses the cache or the write policy is configured to write through. Regarding their access behavior, locked caches can be used like scratchpad memories.

  Such a usage enable to work around unpredictable cache replacement policies like random replacement (cf. Section 9.2.2). Frequently accessed data can be manually filled into the cache and locking is enabled afterwards.

  Another potential application of cache locking is in the presence of write-back memory areas (cf. Section 9.2.3). In its locked mode, the cache does not need to perform any write-back operation since there is never any eviction from the cache that normally causes a write-back of the replaced cache line into the next cache level or main memory. As a write-back policy gives a better performance in the average case, developers tend to decide to use that and thereby making the timing analysis of their task much more difficult in terms of result precision and computational complexity. From the analysis point of view, the locked ways of the cache are considered as scratchpad memory (cf. Section 3.3.4) and the actual cache to be analyzed is smaller enhancing the analysis precision.

- *Branch prediction*:
  As described in Section 3.4.3, branch prediction can be categorized into

a static and a dynamic variant. The dynamic variant uses big tables for caching directions (taken or not taken) of processed branches during execution. Such tables have to be explicitly modeled within the static timing analysis thus increasing its memory consumption dramatically. It is therefore desirable to deactivate dynamic branch prediction and only use its static variant.

- *Clock ratios*:
  Section 9.2.3 details that the ratio between the processor core clock and the system bus clock has to be taken into account by inserting multiple different system states at the start of the pipeline analysis (cf. Section 4.3.3). As there are other reasons for state splits during the abstract simulation, less starting system states potentially save a lot of state splits and therefore computation time and space. This goal could be supported by choosing smaller ratios between core and bus clock. Furthermore, it is desirable to avoid odd ratios like 1:4.5.

- *Multiple bus masters*:
  Multiple bus masters render a timing analysis more difficult as explained in [WGR+09], so if possible, there should be only one bus master in the system, the processor core.

9.2.7 Summary

The timing predictability of the hardware features presented throughout this section can be categorized into three "classes" of predictability:

- features that are actually well predictable if examined in isolation,
- features like write-back cache policies which are predictable in principle but only with pessimistic timing bounds and
- unpredictable features like random replacement caches (although there are workarounds for them as described above).

Most of the features fall into the first class, i.e., they are well understood.

In principle, even the combination of features is not the main problem in state-of-the-art static timing analysis. But some of them interfere in a way that their combination results in a high analysis complexity due to a significant increase of possible execution paths in the presence of (even small) uncertainties.

For example, the memory management units in processors like the IBM PowerPC 750 [IBM06] or Freescale PowerPC 7448 [Fre05a] have numerous queues for buffering memory accesses and there is a sophisticated clash checking for fast data forwarding between consecutive accesses, i.e., it is checked if any requested data has been already retrieved by a previous access. A cache miss results in loading the corresponding cache line which often is more data than it is requested. Following accesses might result in a cache hit therefore. If the target address of a memory access is not precisely known, the outcome of these clash checks could not be predicted by the analysis so that all possible decisions then must be examined. This raises the number of possible execution paths and represents an increasing spiral of complexity because the different types of state splits (cf. Table 6.1 on page 114) mutually influence each other.

Statically, such systems are only analyzable with the major advantage of abstract interpretation as used in the aiT framework (cf. Section 4.3) which is its inherent feature of domain abstraction, i.e., combining sets of concrete system states into one single abstract state. This improves analysis performance rendering such static timing analysis of complex systems feasible at all. Badly designed embedded software might then trigger the bad case for the timing analysis, i.e., the occurrence of multiple different uncertainties at the same analysis time. The resulting state space to be explored by the analysis can be dramatically large increasing the computation time and space consumption. Therefore besides all hardware predictability, it is equally important that the software to be analyzed has been designed with respect to timing analysis as proposed by Gebhard [GCH11].

9.3 Timing Impact of VHDL Constructs

Section 4.3 shows that the basis for the timing analysis performed by the aiT framework is a timing model describing the timing behavior of the examined hardware components in the system to be analyzed. When deriving such a timing model from the VHDL specification, the quality (in terms of WCET bound precision) of the resulting analysis may depend on the coding style used in the VHDL description.

Where the previous section gives an overview of the timing predictability of certain hardware features, this section is about the impact of the VHDL code structure as well as the usage of certain language features on the

precision of the resulting timing model. Due to the language flexibility of VHDL functionally equivalent hardware components can be constructed in different ways. But not all of them are equivalent regarding the applicability of abstract interpretation. A VHDL design therefore is better predictable if the derived timing model enables a higher precision regarding the computed time bounds.

The following results have been achieved during the development of the *VHDL Timing Model Derivation Toolset* (cf. Chapter 10) as the tools have been applied to different VHDL specifications of different coding styles. Details on the experimental results are shown in Chapter 11). The insights of this evaluation of VHDL design decisions have been obtained by comparing different VHDL models. Unfortunately, some of these models are not open and the intellectual property of different hardware manufacturers so that they are confidential. Because of that, it is not possible to give concrete examples for the design decisions in order to prevent any legal problems.

In the following, two classes of VHDL design decisions are presented that distinguish between predictability enhancing and degrading decisions.

9.3.1 Predictability Enhancing Design Decisions

The design decisions described in this section have an enhancing effect on the design's predictability, i.e., the quality of the resulting timing model using the methods described in Chapter 6 and Chapter 7 is better in terms of WCET bound precision.

Complex Data Types

From a structural point of view, hardware is rather simply arranged in zeroes and ones. This should be reflected in the used VHDL data types. If the information carried by a variable or signal is simple, i.e., Boolean, its data type should be similarly simple, i.e., a Boolean data type.

Developers tend to use standard integer data types for simplicity reasons so that the Boolean characteristic is not expressed explicitly. This situation might be even worsened when the same logical information is encoded with different data types so that implicit casts occur. For example, a Boolean flag might be encoded with different integer data types (unsigned, signed,

different bit sizes). Other typical examples for such scenarios are control signals like output enables or bus transaction signals.

Using over-complex data types for variables and signals, the complexity of applied static analyzers might grow because they need to store and maintain this information during analysis time. For VHDL signals, the information is stored even three times to maintain the current, last and scheduled signal values, respectively. In addition to the increased analysis complexity, the identification of the instruction flow through the pipeline is more difficult. For the case of different data type usage for the same logical information, the application of domain abstractions is more complex because consecutive transformations are needed, one for each data type.

Process Separation

Process separation means to separate logically or structurally different things into different processes. For example, the stages of a processor pipeline might be implemented within their own VHDL processes and communicate over signals that control the data paths between them. Similar separations might be done at other levels, e.g., to separate the specification of functional units. The instruction flow of the processor pipeline is then represented more explicitly within the VHDL design.

Positive effects of such a process separation besides a better maintainability and readability of the VHDL code would a better applicability of purely functional code removal as described in Section 6.5.2. If the complete hardware behavior is implemented within one big process, the identification of purely functional parts of the design by static analysis gets more difficult.

Control Signals

Section 6.5.1 describes that the flow of instructions through the processor pipeline might be controlled by a set of signals which are called control signals. This means for example that each stage of the pipeline has some sort of “output enable” signal indicating that the current contents of the stage can be advanced to the next stage.

The advice is to introduce such control signals for explicit modeling of the pipeline control structure instead of using variables/signals containing actual data like addresses. Furthermore this would enhance the readability of the

VHDL code for the user because the separation between control structure and actual functionality, e.g., adding values, is much clearer then. The instruction flow can then be better detected by the methods mentioned in Section 6.5.1.

Functionality Separation

Where the introduction of explicit control signals is good for a separation between control structure and functionality, the functionality itself might be separated further. Error and exception handling has to be processed in each hardware model. The corresponding checks and actual error and exception code are often ignored for a timing analysis. Because of that, such code should be separated from the actual functionality in order to simplify its removal during iterative model refinement (cf. Section 6.5.1).

Code Discipline

VHDL is a feature-rich and complex language offering different ways to express the desired semantic effect of a hardware component. Because of that it is mandatory for the programmer to code in a disciplined way. This mainly means to use the same data type for the same kind of information. Often it can be seen that the same kind of information is stored in variables/signals of different types in different processes. This decreases the readability and simplicity of the whole model as well as making life harder for domain abstractions because they rely on the fact that the same kind of information is represented by the same data types. An example for that might be that an address can be represented by different custom data types of different bit sizes.

9.3.2 Predictability Degrading Design Choices

The following design decisions have a degrading effect on the design's predictability, i.e., the quality of the resulting timing model. At least it is more difficult to derive the timing model.

Interfaces

The interfaces of processes are their sensitivity lists, i.e., the list of signals for which a value change triggers a repetition of the particular process execution. For VHDL procedures and functions, the interface is their particular list of parameters.

From a static analysis point of view, every element in such an interface imposes a control-dependency. Therefore it is important to minimize the number of elements within these interfaces to those which are really needed. In any case, it should be avoided to include variables or signals which are not necessary at all for the semantic effect of the process, function or procedure.

The consequence of actually not needed elements within interfaces is that the results of static analysis are too pessimistic. For example, lets assume that a system has separated instruction and data caches and two update processes, one for each cache. If the sensitivity list of the instruction cache updating process contains a reference to the complete state of the data cache and vice versa, a static analysis has to assume that any change in the data cache triggers a repeated execution of the instruction cache update process although both updates could be isolated.

Combinatorial versus Sequential Logic Design

As defined by Ashenden [Ash08], a combinatorial design is given if the output values are determined solely from the input values. In other words, the design does not maintain any internal state. In contrast to that, a sequential design has an internal state so that the output values depend on the input values *and* the internal state, i.e., the history of execution affects the successor state computation.

Sequential logic designs are more similar to the semantics of traditional imperative programming languages. For that reason they are more suitable to be analyzed using program analysis techniques, for example abstract interpretation. So the advice is to favor a sequential logic design over the other possibilities in order to achieve better results regarding the derivation of timing models as described in Chapter 6. Furthermore, a sequential logic design of a hardware component usually can be mapped better to its corresponding description in the particular user/reference manual which supports the readability of the code.

Instruction Retirement

In order to identify the instruction flow through the processor pipeline, so-called retirement points need to be identified (cf. Section 6.5.1). These are points where an instruction leaves the pipeline so that the instruction semantic has taken effect to the system state.

Depending on the overall structure of the VHDL code, it might become difficult to identify those points precisely. Reasons for that might be confusing variable naming, naming conventions in general, the process structure of the whole design, etc. As stated in Chapter 6, to derive timing models from formal hardware specifications, a good understanding of the whole model is needed. And especially if the person trying to derive a timing model does not belong to the original development team of the analyzed VHDL model, a clear code structure is needed to simplify model understanding.

One cannot give precise definitions for simple and clear code structure as this is a subjective topic. However, if looking at some open-source implementations of hardware components like the LEON2 [Gai05] or LEON3 [GHC05], this subject certainly has to be addressed.

Intra-Process Control Flow

Similar to the above mentioned clear code structure, the control flow within a process should be examined, as well. Creating unnecessary complexity does not only degrade, in a negative way, the readability of the whole model, it might impact results of a static analysis due to abstraction losses at control-flow joins (cf. reasons for uncertainties in Section 9.1). With a clear logical distribution to different processes, the complexity within one process can be bounded.

9.4 Summary

This chapter has started with an intuitive description of the term *timing predictability* of an embedded system in Section 9.1. Based on that terminology, the predictability of existing hardware features are presented and categorized into three classes:

- features that are actually well predictable if examined in isolation,

- features like write-back cache policies which are predictable in principle but only with pessimistic timing bounds and
- unpredictable features like random replacement caches (although there are workarounds for them as described above).

The majority of processor features fall into the first class. Combinations of them can be problematic from the analysis complexity point of view in the presence of (even small) uncertainties. An example of the memory subsystem of the IBM PowerPC 750 [IBM06] and Freescale PowerPC 7448 [Fre05a] depicts the complexity problems arising from the side effects of employed buffers and queues.

The precision of computed WCET bounds depends on how precise the analysis' timing model represents the behavior of the underlying hardware and thereby defines kind of a quality measure for the timing model. In the context of a timing model that has been derived from a VHDL model (cf. Chapter 6), the second part of this chapter identifies and evaluates the impact of some VHDL design choices on the derived timing model's quality. The results are design advices that support the timing model derivation process. Designs suitable for a timing model derivation should feature minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design.

10

VHDL Derivation Tool Set Implementation

"640K software is all the memory anybody would ever need on a computer."

(Bill Gates)

Table 10.1 – VHDL Derivation Tool Set collection

Tool Category	Tool
Parsers	Vhdl2Crl2
Static Analyzers	VhdlResetAnalyzer
	VhdlAssumptionBasedModelRefiner
	VhdlSlicer
Transformation Tools	VhdlTimingDeadCodeEliminator
	VhdlDomainAbstractor
	VhdlProcessReplacer
Generators	PipelineAnalyzerGenerator
	AbstractVhdlGenerator

10.1 Structure of the VHDL Derivation Tool Set

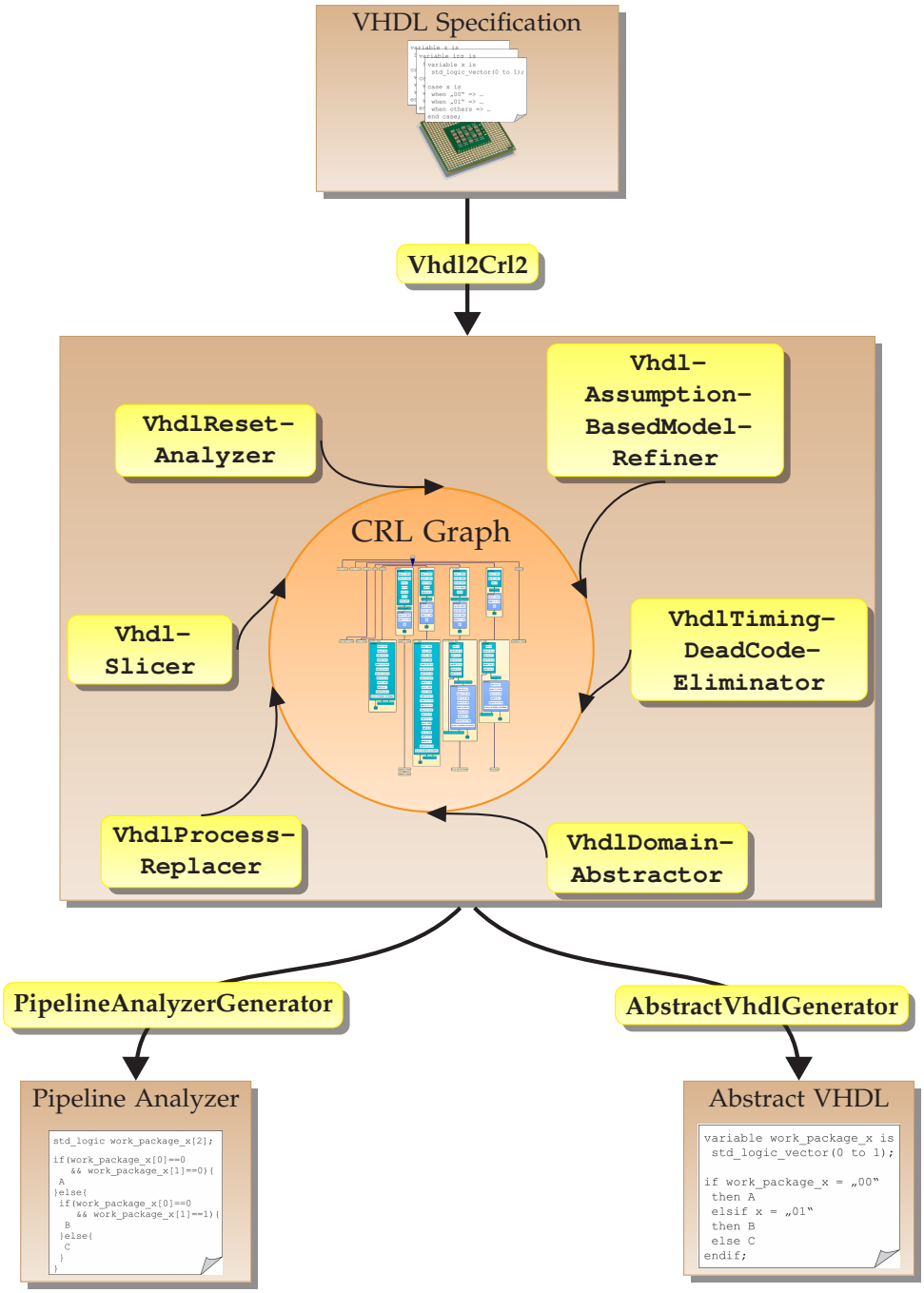
This chapter describes the implementation of the *VHDL Derivation Tool Set*, a collection of tools to support the semi-automatic derivation of timing models from formal hardware specifications in VHDL. The theoretical background has been described in Chapter 6.

The tool set consists of parser, analysis, transformation and code generation tools which are listed in Table 10.1. All of them make use of a support library, libvhdlanu, that realizes the evaluation of VHDL expressions and a prototypical type inference system. This functionality has been factored out into a separate library because it is required by all involved tools.

The VHDL compiler and libvhdlanu have been designed and implemented by my colleague Marc Schlickling and myself in cooperation. In contrast, the static analyzers have been developed solely by Schlickling [Sch13] and I am responsible for the transformation and generation tools. Some early versions of the pipeline analyzer and abstract VHDL generators have been developed by Mohamed Abdel Maksoud [Mak07].

Their structural arrangement is illustrated in Figure 10.1 on the next page. With the help of Vhdl2Crl2, a VHDL model can be parsed and transformed into a CRL graph structure. All static analyzers and transformation tools

Figure 10.1 – VHDL Derivation Tool Set – Structure



operate on this data structure where the results of the analyzers are used to incorporate appropriate changes to the model using the transformation tools.

Having finished the model analysis and transformation cycle, there are two generators available, namely the PipelineAnalyzerGenerator and the AbstractVhdlGenerator. The first is used for the generation of an aiT-compatible pipeline analyzer to be used in its timing analysis tool chain where the latter is able to reconstruct VHDL from the intermediate representation. Due to employed abstractions, the reconstructed VHDL is possibly no longer syntactically valid, so the output in general is called abstract VHDL.

This chapter describes the details of all above mentioned tools in the following sections. After that there are some VHDL language restrictions of the tool set given in Section 10.7.

10.2 VHDL Compiler

As mentioned in the last section, the tool Vhdl2Crl2 is responsible for converting a VHDL design into the internal CRL graph format. This conversion is a result of consecutive processes whose flow is sketched by Figure 10.2 on the facing page. The phases are now described one by one.

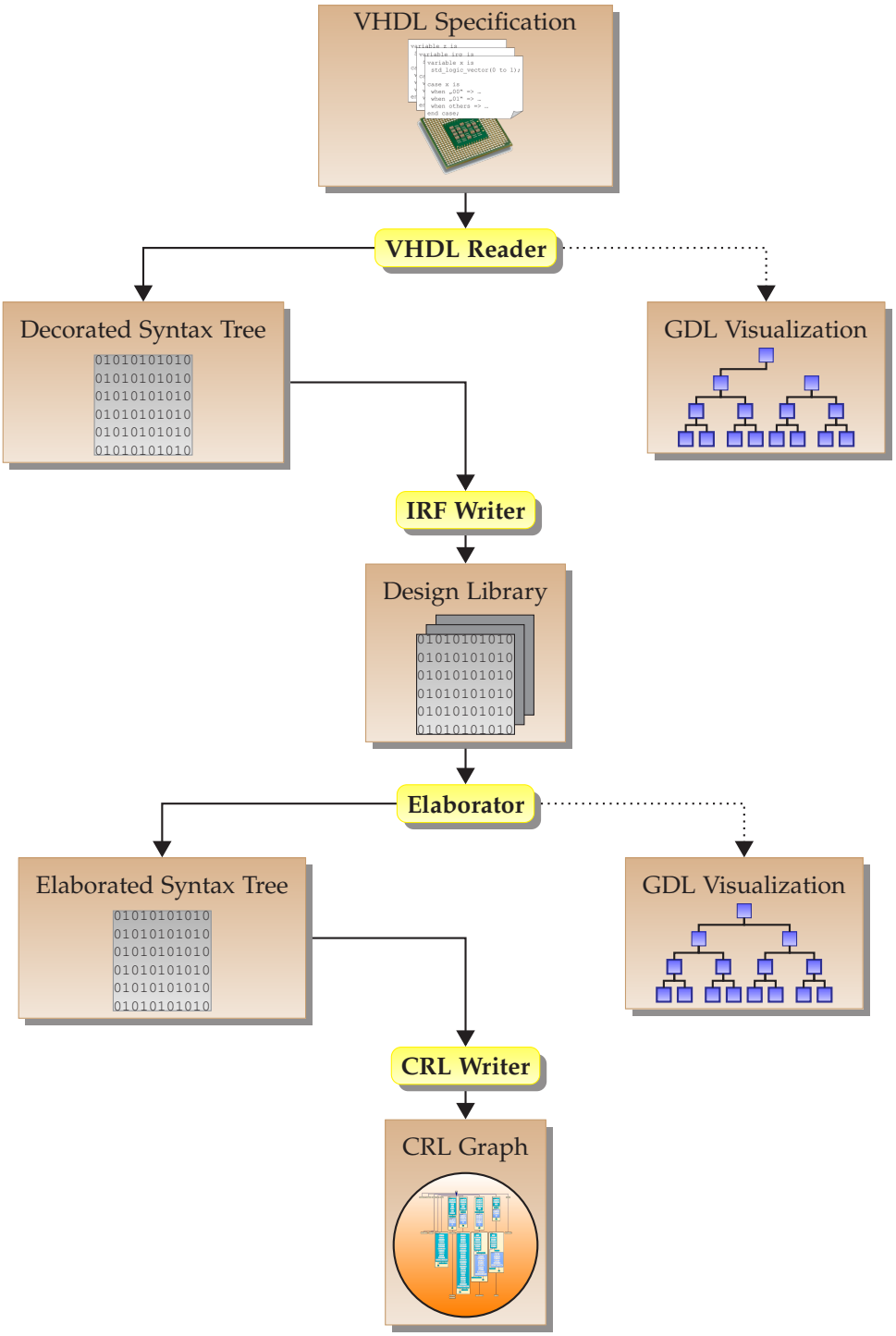
10.2.1 Analysis

First, the VHDL files are fed to the *analysis phase*. During the analysis phase, an input VHDL file is read into memory by performing lexicographic, syntactic and semantic analysis as known from compiler design [WM95].

Besides these "classical" tasks of a compiler, VHDL `case` statements are converted into a nested hierarchy of `if-then-else` statements in order to simplify the CRL generation and static analysis because only this case has to be supported there. The conversion can be easily done as it does not introduce any semantic change to the VHDL design.

The result of the analysis phase in Vhdl2Crl2 is a decorated syntax tree as shown in Figure 10.2 on the next page. To be precise, for each found entity/package in the design, a separate syntax tree is created so that the overall result is a forest of syntax trees depending on the input VHDL.

Figure 10.2 – VHDL Compiler (Vhdl2Crl2) – Structure



Mainly for debugging purposes, an optional visualization of the forest can be generated. Similarly to all visualizations in the aiT framework (cf. Section 4.3), the output format is GDL [EB09] so that aiSee [Abs11] can be used to view and explore the visualization. Figure 10.3 on page 210 shows such a sample visualization. Each rectangle represents a node in the syntax tree with some important properties of the corresponding data structure. These are among others an unique identifier, the specific node type and some references to the original source code location in the input VHDL specification. Figure 10.3 on page 210 for example shows the start of the assignment statement in line 24 of Listing 5.1 on page 98 – the 3-bit VHDL counter example. The node with the id 605 represents the topmost sequential statement node and its left child indicates that this specific statement is an assignment statement. In the left child sub-tree, the left hand side of the assignment is encoded where the right sub-tree analogously represents the right hand side of the assignment. As this is an expression with a high depth, only the topmost parts of the corresponding syntax tree are shown in this example. The source code documentation contains a detailed description of the syntax tree specification. It explains the intent of all node types and the way in which nodes of different types can be connected to each other to build a valid syntax tree.

10.2.2 IRF Writer

Each syntax tree generated by the analysis described in the previous section is stored in a special textual format called *intermediate representation format (IRF)*. To illustrate the format, a sample snippet of an IRF file is shown in Listing 10.1 where the syntax tree from Figure 10.3 on page 210[1] is represented.

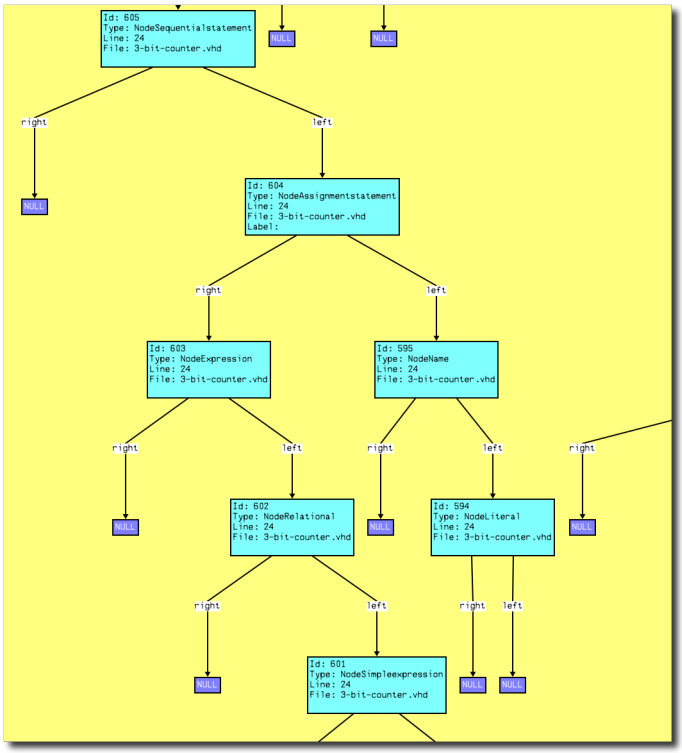
The syntax of the IRF file format is in its structure similar to XML where each line in a tag body consists of an assignment of the form $key = value$. Each used *key* then refers to a special member of the corresponding class data structure. In line 3 of the listing, a syntax tree node specification is started by opening the tag "`NODE`". Then, in lines 5–8, some common properties are specified. Lines 12–35 then define the left sub-tree of the left child node illustrating the recursive structure of the IRF grammar. In other words, the left child of a node is again a node with left and right children specifications.

[1] Graphical representation of left and right sub-trees have been swapped by aiSee's layout algorithm.

Listing 10.1 – Sample IRF file extract

```
...

<NODE>
# Common node properties inherited from ParseNode
ID       ="605";
TYPE     ="NodeSequentialstatement";
LINENO   ="24";
FILENAME ="3-bit-counter.vhd";

  # Definition of the left child
  <LCHILD>
    <NODE>
    # Common node properties inherited from ParseNode
    ID       ="604";
    TYPE     ="NodeAssignmentstatement";
    LINENO   ="24";
    FILENAME ="3-bit-counter.vhd";
    # Common node properties inherited from StatementNode
    LABEL="";

    # Specific properties
    ASSIGNMENT_TYPE="SIGNALASSIGNMENT";

    # Definition of the left child
    <LCHILD>
    ...
    </LCHILD>

    # Definition of the right child
    <RCHILD>
    ...
    </RCHILD>
  </LCHILD>

  # Definition of the right child
  <RCHILD>
  ...
  </RCHILD>
</NODE>

...
```

Figure 10.3 – Sample Vhdl2Crl2 generated syntax tree



Such an intermediate format is needed because a VHDL design might be distributed over multiple source code files with dependencies among them due to data structure references. In that case, each file has to be analyzed (as described in the last section) separately in the correct order. In other words, the analysis of a VHDL code module *a* which depends on data structure declarations provided by code module *b*, requires *b* to be analyzed first so that needed declarations can be loaded from existing IRF files. The collection of all IRF files is called the *design library* for the VHDL specification that is iteratively filled with parser information from the analysis phase.

The order, in which the files are given to Vhdl2Crl2, has to be determined manually by the user itself. Other existing VHDL parsers like GHDL [Gin] handle this in the same way although there are some commercially available parsers (like the one used in Mentor's ModelSim [Men08]) which are using heuristics to determine the correct file order automatically. Basically, they use a trial-and-error approach that permutes the order smartly until all unresolved reference errors are eliminated. For Vhdl2Crl2, such an enhancement

could be implemented, as well (cf. Section 12.2).

In the end, the whole VHDL design to be processed is available in the design library so that the elaboration can be started.

10.2.3 Elaboration

After all source files have been analyzed and added to the design library as described above, the VHDL standard [IEE99a] requires a specific transformation process called *elaboration* to be applied. This process has been described and introduced in Section 5.2.4.

Vhdl2Crl2 implements this elaboration process with respect to a given entity name which represents the topmost component in the given hardware design. In addition to the particular transformations described in Section 5.2.4, the following changes to the VHDL code are realized:

- *Type name conversion*:
  For example, a type name t declared in a package p of a library l may be referenced by $l.p.t$. Depending on how the library has been made visible (cf. semantic of the `use` clause), the type is additionally visible under its base name t. All used type names are renamed by their fully prefixed name in order to have a consistent view on identifiers.

- *Early constant propagation*:
  Constant values are propagated during the elaboration process for one level, i.e., all usages of constant values are replaced by their particular constant value. Only one level is propagated because this can be done in a single-pass and without further analysis which is performed by the static analyzers (cf. Section 10.3) anyway.

- *Removal of instantiated component declarations*:
  Component declarations which have been instantiated are removed in order to improve readability of the code. Due to the instantiation, such declarations are never referenced in the design.

- *Garbage Collection*:
  The elaboration steps described above have transformed the forest of syntax trees generated by the analysis phase (cf. Section 10.2.1) into one single big tree containing all needed data structures for the hardware design. It is the syntax tree of the topmost entity specified for elaboration. If there is still any syntax tree separated from this big tree,

it is definitely unused in the design and for the given topmost entity. A further optimization step, removes any such "orphaned" trees.

- *Expansion of function calls*:
  All function calls used within compound statements are factored out into assignments to dynamically created temporary helper variables or signals. This step's purpose is to restrict locations of nondeterminism. As described in Section 7.3.2, the goal is to only have at most one operation or function call in assignment statements in order to simplify the abstract simulation semantics.

All these listed transformation steps are not explicitly required by the VHDL standard and represent model optimizations which support a later static analysis.

10.2.4 CRL Writer

After the VHDL model has been analyzed and the resulting syntax tree is elaborated, the corresponding CRL formulation can be generated. The implementation makes use of the AbsInt (`http://www.absint.com`) support library libcrl2.

As described in Section 4.3.2, CRL has been designed to represent the control-flow of a program. Therefore, the structure is hierarchically composed and consists of routines and basic blocks with instructions. The latter even might be build of multiple micro operations per instruction to reflect assembly programs for VLIW architectures (cf. Section 3.3). In the context of representing VHDL designs as sequential programs, micro operations are not needed so that a VHDL statement is assigned to a CRL instruction. More details about CRL can be found in Section 4.3.2.

Analysis Framework

To support static analysis on such VHDL models (cf. Section 10.3), the following additional constructs have to be generated:

- Routine **`simul`**:
  This routine consists of a loop whose body contains routine calls where each called routine represents one of the former VHDL processes as converted from the input VHDL specification. All calls are guarded,

i.e., there are conditionals whose expressions decide whether the call is actually executed or not. They are needed to model delta cycles due to value changes in signals contained in the particular sensitivity list of a VHDL process (cf. Section 5.2.3).

- Routine **`vhdl_clock`**:
  The clock cycles are represented by this routine containing a loop whose body simulates a full clock cycle, i.e., the routine `simul` is called twice: once for the simulation of a rising core clock and once again for the simulation of a falling clock edge. The mentioned loop is modeled as a self-recursive routine for a better static analysis support [MAWF98].

- Routine **`environment`**:
  Environmental signal assignments are represented by a call to this special routine where external signal assignments can be modeled. This routine is called from the simulation loop in routine `simul`.

- Routine **`analysis_start`**:
  The current implementation of the program analyzer generator (PAG), which is used to specify the static analyzers (cf. Section 10.3), does not allow the start of any analysis to be a self-recursive routine so that the routine `analysis_start` is generated as a wrapper. It calls the self-recursive routine `vhdl_clock`.

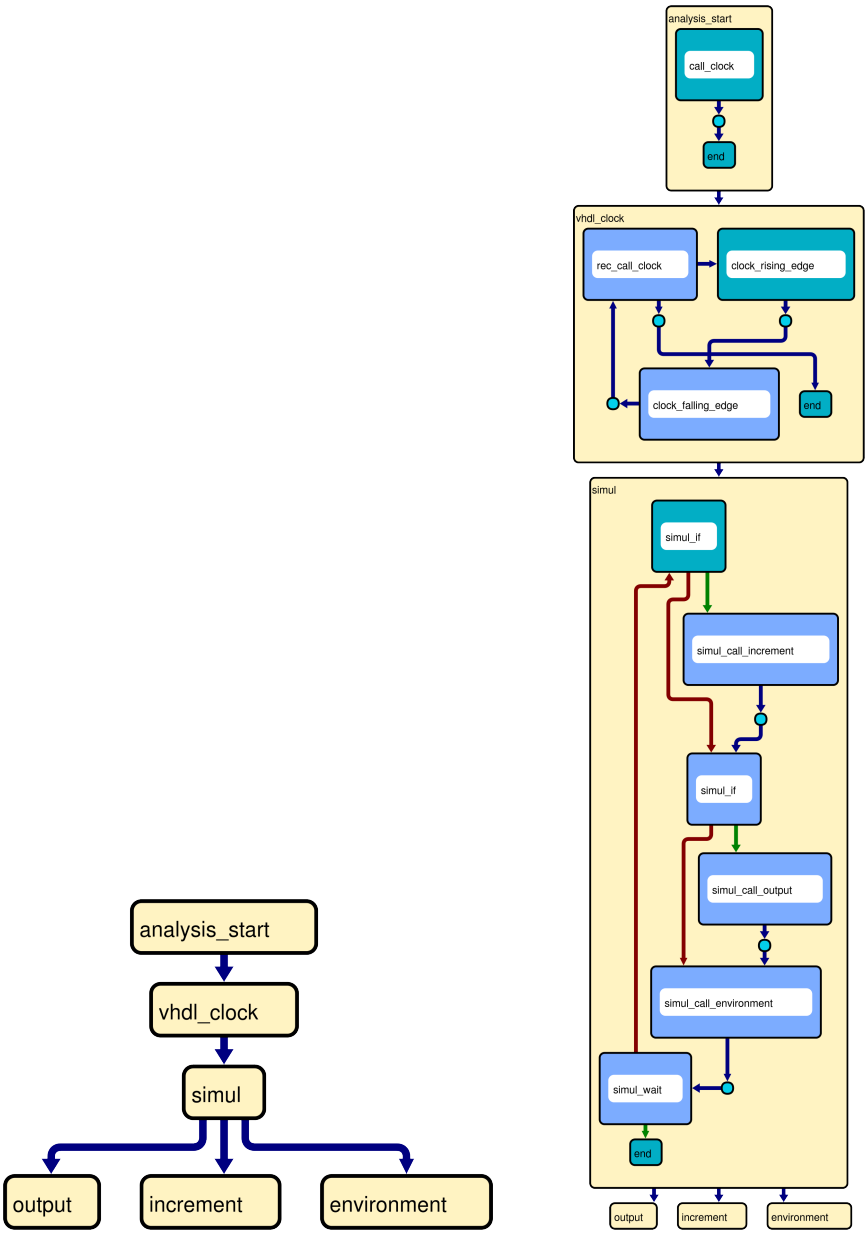
These constructs build up an analysis framework needed for the simulation of the VHDL semantics.

Figure 10.4 shows a GDL [EB09] visualization of the CRL graph generated from the example of a 3-bit counter in Listing 5.1. Sub-figure (a) shows the graph in its folded view so that only routines are visible, in order to get an overview. In this small example, only the routines `output` and `increment` originate from the input VHDL model. All other routines implement the simulation and analysis framework as described above. In sub-figure (b), the routines `analysis_start`, `vhdl_clock`, and `simul` are shown in a boxed view so that their control-flow is visible. The starting basic blocks of a routine are always filled with the same color as the corresponding `end` block.

Meta Information

Similar to compiler-generated debug information in binary executables, CRL has the capabilities to store arbitrary data in so-called *meta* sections. They

Figure 10.4 – Sample Vhdl2Crl2 generated CRL graph



(a) Folded view (b) Boxed view

are special language constructs which can store lists, vectors, maps, etc., of values like strings or numbers. Vhdl2Crl2 uses these meta sections to store the following kind of information:

- *Hierarchy Information*:
  The elaboration process (cf. Section 10.2.3) eliminates any hierarchy in the used data structures, e.g., by collapsing record variables/signals into its elements recursively. But for the VHDL reconstruction, the hierarchy information has to be preserved and is therefore stored in a special meta section of the generated CRL graph. For each identifier in the original VHDL code, a string encoding its location within other data structures is stored, i.e., a list of other identifier names in which it is contained.

- *Global Value Map:*
  During the CRL graph generation, Vhdl2Crl2 stores all encountered constant definitions together with their values in a specific meta section which is called the global value map. These values are later used by the static analyzers (cf. Section 10.3).

- *Global Type Map:*
  Actual types of variables and signals (except function/procedure parameters, see below) are stored in a specific meta section of the CRL graph which is called global type map. The data from the internal type checking system are for example needed for the VhdlDomainAbstractor tool (cf. Section 10.4.2).

- *Parameter Type Map:*
  Types of parameters of functions and procedures are treated separately from all other identifiers. Their visibility is restricted to the particular function/procedure, so they are stored in a special meta section and are flagged with their particular routine name.

The global and parameter type maps are similar to symbol tables as known from the compiler construction [WM95]. Actually, their content is filled with information from the symbol table within Vhdl2Crl2 after having parsed the whole VHDL design.

10.2.5 Usage

Synopsis

Vhdl2Crl2 [-i | --irf] [-g | --gdl]
[-c | --crl
[[-r|--rising-clock-edge] [-f | --falling-clock-edge]]]
[-e|--elab <name>] [-t | --trace-file] [-q|--quiet <int>]
[-o|--output <file>] -d|--design-library <path>
[-p|--library-prefix <name>] [name]

-i | --irf
Generate IRF file from parsed VHDL file.

-g | --gdl
Activate GDL visualization.

-c | --crl
Activate CRL generation.

-e | --elab <name>
Activate elaboration of design library. `<name>` is the top level entity name.

-r | --rising-clock-edge
Only generate rising clock edge triggered model. Effective in conjunction with --crl.

-f | --falling-clock-edge
Only generate falling clock edge triggered model. Effective in conjunction with --crl.

-q | --quiet <int>
Quiet level within the range [-1,6]. Default is -1. A higher quiet level results in less emitted messages.

-o <file>
Optional name of the generated CRL file. Effective in conjunction with --crl. If omitted, output is emitted on standard out.

-d | --design-library <path>
Path to the design library for storing and reading.

Listing 10.2 – Vhdl2Crl2 sample usage

```
1 ## Analysis of a VHDL file
2 Vhdl2Crl2 --design-library . --irf  3-bit-counter.vhd
3
4 ## Elaboration of the design library and CRL file generation
5 Vhdl2Crl2 --design-library . --elab Counter --crl -o Counter.crl
```

-p | --library-prefix <name>
Specify prefix attached to the front of each entity name in any generated IRF file. Effective in conjunction with --irf.

The <name> parameter specifies the input VHDL specification which should be parsed. If <name> is omitted, the input is read from standard input (except when using the --elab switch, cf. Section 10.2.3).

Example

Listing 10.2 shows a sample usage of Vhdl2Crl2 for processing the 3-bit counter example in Listing 5.1 on page 98 (file 3-bit-counter.vhd). First, the VHDL file is analyzed and added to the design library by converting it into IRF format. Then, the design library is elaborated and the corresponding CRL formulation is generated into the file Counter.crl.

10.2.6 Complexity

As shown in Figure 10.2 on page 207, Vhdl2Crl2 consists of the four processes *VHDL Reader*, *IRF Writer*, *Elaborator* and *CRL Writer*. The parser used within the VHDL reader is generated using the open-source tool bison [Fre11], so its runtime complexity is the one of a LALR [WM95] parser. All other processes are iterating over the syntax forest (cf. Section 10.2.1), so their asymptotic runtime complexity is linear in the number of nodes in all syntax trees. IRF and CRL generation are single-pass processes, i.e., they only need to iterate once over each syntax tree. In contrast, the elaboration process is more complex because it is composed of numerous different transformation steps which are executed consecutively (details about the elaboration can be found in cf. Section 10.2.3). Therefore, this step is the most complex one concerning the runtime performance.

VHDL specifications tend to be large concerning the lines of code as described in Section 11.2. This property propagates to the corresponding syntax trees and crl representations so that they can grow up to multiple gigabytes.

More statistics about runtime performance and space consumption issues can be found in Chapter 11.

10.3 Static Analyzers

The following list of static analyzers belong to the VHDL derivation tool set but have been implemented by Marc Schlickling [Sch13] solely and are only listed for the sake of completeness:

- *VhdlResetAnalyzer* computes initial signal and variable assignments of a VHDL design.
- *VhdlAssumptionBasedModelRefiner* computes parts of a VHDL design that are so-called timing-dead (cf. Section 6.4.1).
- *VhdlSlicer* computes backward slices on a VHDL design.

All of these analyzers operate on the intermediate representation (CRL) of the VHDL model as illustrated by Figure 10.1 on page 205. The computed results are stored in special attributes of the CRL graph so that subsequent tools analyzing or transforming the graph can access them.

10.4 Model Transformers

Besides the static analyzers mentioned in the previous section, some model transformation tools have been developed representing source-to-source translations on the CRL graph. Based on the results of the analyzers from the last section, these tools are used to actually shrink the size of the timing model. Furthermore, the introduction of state abstractions is supported.

Debugging, understanding and exploration purposes are supported by visualization capabilities of the CRL graph using the graph viewer software aiSee [Abs11]. For this, one of the tools from the aiT tool chain – `crl2gdl` – can be used to convert any CRL file output of the presented transformation tools into GDL representation. As all transformation tools operate on CRL, their

structural changes to the graphs can be illustrated by such visualizations. Sample aiSee visualizations of control-flow graphs are shown in Figure 10.4 on page 214.

10.4.1 Timing Dead Code Eliminator

VhdlTimingDeadCodeEliminator removes parts of the VHDL model which have been marked as "timing-dead", so the tool is based on the results of the VhdlAssumptionBasedModelRefiner (cf. Section 10.3). Marking a construct as timing-dead is realized by the CRL attribute `vhdl_timing_dead` which is interpreted as a Boolean value pointing to such dead constructs.

Structure

The implementation operates on the CRL representation of a VHDL design as illustrated in Figure 10.1 on page 205. First, the input CRL file is read into memory. Then, a depth-first search (DFS) on the graph structure is performed that checks for the above mentioned timing-dead markers and performs the removal. Removal here means that the particular dead component is unlinked from the graph structure. After the DFS walk, some post-processing steps are done which are detailed below. In the end, the shrunken CRL description is written to the specified output file. All input and output processes are realized using the AbsInt libcrl2 library.

CRL Graph Iteration

The DFS walk over the graph structure checks the timing-dead attribute for all CRL constructs that are annotated by Schlickling's static analyzer [Sch13], namely routines, instructions and edges. Therefore, dead routines and instructions can be identified directly by checking the attribute. Each such routine or instruction is stored separately for deletion in the post-processing steps and all their incoming edges are marked as timing dead. A basic block b is identified as dead if all of b's incoming edges have been marked as dead by the analyzer. Consequently, all outgoing edges of b are marked as dead and b is marked for the post-processing steps (see below). The storage of pointers to dead constructs and their post-pass deletion is an implementation details and implied by the libcrl2 library. Direct deletion from the graph

is not possible because the iteration would otherwise fail with a runtime error.

Additionally, constructs which potentially can be simplified due to the removal are stored for post-processing, as well (see below for a description of the simplification process).

Post-Processing

After the DFS walk, some post-processing steps take place. They have to be done afterwards in order to technically simplify the DFS walk. For example, there might be different paths leading to the same dead block. If the block would be directly deleted during its first occurrence on the DFS walk, the iteration over the graph needs to take care about deleted constructs.

- *Deletion of stored dead blocks/routines/instructions*:
  The CRL constructs stored during the DFS walk (see above) are actually deleted from the heap memory in this step.
- *Code simplification*:
  Due to the code removal, basic blocks surrounding deleted blocks potentially can be merged to one new block. If a block b has only one alive, i.e., not timing-dead, outgoing edge, the successor block can be merged with the block b. Another situation where a simplification is applicable is if a call edge, i.e., an edge from a basic block to another block in a different routine, is marked as timing-dead, e.g., due to having marked the whole target routine dead. Then, the return block, i.e., the successor block of the block containing the call, can be merged with the caller block.

Timing-Dead Propagation

The property of being timing-dead might be propagated from one basic block to another one by the removal. For example, an originally "timing-alive" basic block could get timing-dead because all incoming edges have been marked dead as a consequence of the deadness of its predecessor blocks. This propagation is supported by VhdlTimingDeadCodeEliminator and can happen for basic blocks and routines. Typically, this occurs in the presence of concurrent signal assignments where the destination signal has been assumed to be of constant value. In the control-flow graph, such assignments

Listing 10.3 – VhdlTimingDeadCodeEliminator statistics

```
1 Progress: [09:53:03]: Parsing input Crl file 'else-case.crl'...
2 Progress: [09:53:03]: Computing transformation statistics...
3 Info: Eliminated 0 (0 empty) of 3 routines ( 0 percent ).
4 Info: Eliminated 3 (2 empty) of 17 blocks ( 18 percent ).
5 Info: Eliminated 0 of 8 instructions ( 0 percent ).
6 Info: Eliminated 3 of 17 edges ( 18 percent ).
7 Progress: [09:53:03]: Writing result Crl file...
```

are represented by one instruction that is solely contained within one basic block which itself is contained in a single routine. Marking this instruction as timing-dead results in the removal of the surrounding basic block which is then empty and therefore results in the removal of the whole routine due to the timing-dead propagation.

Statistics

At the end of the transformation process, a statistic is emitted showing a percentage and absolute number of removed CRL constructs. The output is categorized by the construct type, i.e., subdivided into the number of removed blocks, routines or instructions.

Listing 10.3 shows a sample output of VhdlTimingDeadCodeEliminator where the statistic can be read in lines 3–6. They show the number of eliminated routines, basic blocks, instructions and edges respectively. If some of the eliminated elements have been removed because of a timing-dead propagation (cf. above), their portion of the total number of removed elements is shown in parentheses and called *empty*. For example in line four, one can see that two of the three eliminated blocks have been removed due to such a propagation. Section 11.4 interprets such propagation rates in a quantitative description of possible VHDL model reductions.

Complexity

The asymptotic runtime of VhdlTimingDeadCodeEliminator is linear in the sum of the number of nodes and edges in the graph as the DFS walk is the dominating process concerning the runtime. Consumed main memory

is linear in the number of CRL constructs in the input graph as the corresponding data structures are held in the heap memory of the process. This dominating role of the CRL graph iteration arises in the runtime and memory consumption experiments which are described in Section 11.3 and show a linear complexity dependency from runtime and consumed main memory to CRL graph size.

Usage

VhdlTimingDeadCodeEliminator [-o | --output <file name>]
[-v | --verbose] <file name>

`-o | --output <file name>`
Transformed CRL file is written into the file specified by <*file name*>.

`-v | --verbose`
Activate more progress and information messages.

The `<file name>` parameter specifies the input CRL file.

10.4.2 Domain Abstractor

The VhdlDomainAbstractor supports automatic domain abstractions as described in Section 6.4.2, i.e., the conversion of identifiers from a source to a destination type where both can be specified on the command line of the tool.

Structure

As illustrated in Figure 10.1 on page 205, the VhdlDomainAbstractor operates on the CRL graph structure which has been read into memory from a given file. Analogously to the VhdlTimingDeadCodeEliminator (cf. Section 10.4.1, the parsed CRL graph is traversed by a DFS walk searching for identifiers matching the specified source domain. Having finished the graph traversal, results are printed:

- transformed identifiers,
- needed operator implementations due to the transformation and optionally

- further transformation proposals if the search scope has been restricted (see below).

CRL Graph Iteration

For each CRL instruction (aka VHDL statement) which is encountered during the DFS walk over the graph, the corresponding identifier uses and definitions are determined. Then, their types are extracted from the global and parameter type map (cf. Section 10.2.4) sections stored in the CRL graph. All identifiers whose type matches the specified source domain are converted to the specified destination domain by adjusting the corresponding entry in the type map meta sections of the graph. There is a so-called *check-only* operation mode where the computed transformations are not applied. The actual computation is not changed, but a written model is the identity of the input model. This can be used to determine needed operators under a given domain abstraction without actually modifying the input.

VhdlDomainAbstractor features so-called *transformation scopes*, i.e., the user might want to restrict the type transformation to a certain set of identifiers. Then only these identifiers are processed which is useful for partial conversions. For example, one might only want to focus on a specific part of the model in order to postpone the examination of all other parts. By this, the domain abstraction can be used iteratively which simplifies the processing of larger VHDL designs.

Due to the type conversion, new operators working on the destination domain might be needed. Therefore, the graph traversal additionally checks all used operators after the type conversion. For example, lets assume there is an expression $a + b$ where a and b both are addresses (e.g. `ieee.unsigned (31 downto 0)`). If a and b are converted to an abstract domain *AddressRange* where a and b both represent an interval of addresses, an operator $+_{interval}$ is needed which can work on the new type. After having finished the type conversion, VhdlDomainAbstractor prints out such collected needed operators that have to be provided by the user. Already implemented operators can be specified via a file containing their signatures (see Section 10.4.2). These are then not reported by the tool.

Another helpful feature of the domain abstraction tool are transformation proposals if the transformation scope has been restricted to a set of identifiers.

If the conversion of an identifier type is only prevented by this scope restriction and the transformation proposals are enabled, this identifier is stored internally. After the transformation process, all collected (not converted) identifiers are printed.

Complexity

The asymptotic runtime of VhdlDomainAbstractor is linear in the sum of the number of nodes and edges in the graph as the DFS walk is the dominating process concerning the runtime. Consumed main memory is linear to the number of CRL constructs in the input graph as the corresponding data structures are held in the heap memory of the process.

Usage

VhdlDomainAbstractor [-o | --output <file name>]
[-v | --verbose]
-s | --source-domain <domain>
-d | --destination-domain <domain>
[-c | --check-only]
[-r | --restrict <identifier>]
[-p | --enable-proposals]
[--signature-file <file name>]
<crl file name>

-o | --output <file name>
Transformed CRL file is written into the file specified by <*file name*>.

-v | --verbose
Activate more progress and information messages.

-s | --source-domain <domain>
The source domain is specified by <*domain*>.

-d | --destination-domain <domain>
The destination domain is specified by <*domain*>.

-c | --check-only
Activates the *check-only* mode.

-r | --restrict <identifier>
Transform only the given identifier. This option can be used multiple times to restrict the transformation to specific identifiers.

-p | --enable-proposals
Activate transformation proposals. This option is only meaningful in conjunction with the `restrict` option.

--signature-file <file name>
Already implemented new operators can be specified via the file whose name is given by <*file name*>.

The `<crl file name>` parameter specifies the input CRL file.

10.4.3 Process Replacer

Compared to VhdlTimingDeadCodeEliminator and VhdlDomainAbstractor, the VhdlProcessReplacers implementation is simpler. It automates the replacement of VHDL processes by custom simulation routines which are provided by the user.

First, a specification file (given on the command line) is read into memory. It contains pairs of CRL routines and URLs where the first one identifies the routine to be replaced and the latter points to the file containing the corresponding custom implementation of the process. Analogously to the VhdlTimingDeadCodeEliminator and VhdlDomainAbstractor, the CRL graph structure is read into memory, as well.

Then, the algorithm iterates over all existing routines in the CRL graph. Any routine whose name matches one of the names in the specification file mentioned above, is flagged with the following attributes:

- `vhdl_custom_simulation_routine`:
  This attribute is of type Boolean and indicates that the routine has been reimplemented by a custom simulation routine.

- `vhdl_custom_simulation_routine_url`:
  This attribute contains the source code location of the new implementation for the routine.

All calls to such a routine are flagged with the same attributes which is needed for other tools like the PipelineAnalyzerGenerator (cf. Section 10.5.1).

Moreover, a special meta section (`process_replacement_map`) is added to the graph representing the mapping from the specification file. The abstract VHDL generator makes use of this meta section. It only integrates empty VHDL processes into the reconstructed design because the custom simulation code is mainly written in C and not VHDL.

Ideally, these routines would not only be flagged with attributes to identify them as being reimplemented by a custom simulation routine, but actually replaced by the new implementation. This is not done because the new code must not necessarily be written in VHDL. As the main goal is to generate an aiT-compatible pipeline analyzer from the model, such custom simulation routines actually are written in C/C++.

10.4.4 Complexity

The asymptotic runtime of VhdlProcessReplacer is linear in the sum over the number of nodes and edges contained in the input CRL file as the parsing and rewriting of the CRL file is the dominant runtime influencing factor. Fortunately, the used AbsInt implementation of the access library for CRL directly offers access to the routines of a CRL graph so that this only consumes constant asymptotic runtime. As the number of entries in the specification file cannot exceed the number of routines in the CRL graph, this reading process can be neglected in the runtime examination.

The consumed main memory is linear to the number of CRL constructs in the input graph as the corresponding data structures are held in the heap memory of the process. For the same reason as in the runtime examination, the number of entries in the specification file only is a negligent factor.

10.4.5 Usage

VhdlProcessReplacer [-o | --output <file name>]
[-v | --verbose]
-s | --specification <file name>
<crl file name>

-o | --output <file name>
Transformed CRL file is written into the file specified by <*file name*>.

-v | --verbose
Activate more progress and information messages.

-s | --specification <file name>
The name of the specification file is given by <*file name*>.

The `<crl file name>` parameter specifies the input CRL file.

10.5 Generators

As illustrated by Figure 10.1 on page 205, two possible outputs can be generated from the CRL graph: an aiT-compatible pipeline analyzer and a (partially) abstract VHDL representation. Both are described in the following sections.

10.5.1 Pipeline Analyzer Generator

The PipelineAnalyzerGenerator produces ANSI-C code implementing the functionality of a cache/pipeline analysis as presented in Section 4.3.3. Basis for the generation algorithm are the inference rules for abstract simulation of VHDL designs which have been defined in Section 7.3. By this, the rules additionally serve as a formal description of the implementation.

Each routine in the CRL graph is printed by the tool in its ANSI-C representation. This task is straightforward and is mainly concerned with syntactical transformations except for signal assignments. They have to be treated specially due to their "delayed" semantics, i.e., their assignments only take effect after all routines have been executed (cf. Section 5.2.3). A global storage construct has to be generated which temporarily stores future values for signals collected by the execution of their assignments. Therefore the code for signal assignments is a bit special because the need for the global storage update. Additionally, simulation code is generated in order to correctly simulate repeated process execution due to VHDL delta cycles (cf. Section 5.2.3).

Generated Analyzer Structure

An aiT-compatible pipeline analyzer consists of two logical parts: one representing the actual timing model of the underlying hardware architecture and another generic part which connects the architectural model with the control-flow graph of the executable to be analyzed.

Due to its generic property, the latter part does not need to be generated. It is already offered as a generic implementation (*pipeline analyzer framework*) where the architectural part can be connected with by a specific interface. This interface is a main function – `single_step` – which computes the evolution of an input system state by a single processor core clock update. One of its parameters is an input system state and it performs a processor core clock cycle update, i.e., all possible successor states (according to the semantics of the underlying model) are computed. Due to the employed state abstractions there might be multiple successor states for one input state (cf. Section 6.2.1). The analysis framework iteratively feeds appropriate input system states to the generated `single_step` function and collects all computed successor states until the task simulation is completed.

As mentioned above, each CRL routine is converted to an ANSI-C function in the generated pipeline analyzer. The main update function `single_step` then consists of consecutive calls to all those functions which represent processes in the original model. Corresponding information has been preserved in CRL graph attributes.

There is a small difference between the inference rules for VHDL delta cycles and the implementation in the generated pipeline analyzer: if the value of a signal has changed, all VHDL processes need to be executed again according to the abstract simulation semantics (cf. Section 7.3.2). Actually, this is an over-approximation to the concrete VHDL semantics as only those processes that are sensitive to that signal, need to be restarted. This optimization is realized by the pipeline analyzer.

The runtime complexity of PipelineAnalyzerGenerator is asymptotically linear to the sum over the number of nodes and edges in the CRL graph as the generation process can be done in a single pass over the graph structure. Similarly, the memory footprint is dominated by the in-memory CRL graph.

The work on this aiT-compatible pipeline analyzer has already been published by Maksoud et al. [Mak07, MPS09].

10.5.2 Abstract VHDL Generator

Although the main purpose of the VHDL derivation tool set is to generate an aiT-compatible pipeline analyzer from the timing model in CRL presentation, another tool called AbstractVhdlGenerator has been developed which is able to regenerate VHDL code from a CRL representation. The motivation is that the VHDL code generated this way can be used as input for a validation of the timing model using interval property checking as described in Chapter 8. Another advantage of the VHDL regeneration is the better readability of the VHDL in contrast to the pipeline analyzer code. For people familiar with the original design, the regenerated code is looking familiar and recognizable which additionally supports debugging processes.

As mentioned in Section 5.2.4, the VHDL elaboration process eliminates higher-ordered language constructs like records by atomic values. The VHDL restoration process reverts this by restoring hierarchy information stored in the `hierarchy` meta section (cf. Section 10.2.4) of the CRL file.

Most of the CRL components can be easily converted to valid VHDL statements. As processes and subprograms both correspond to routines in the CRL representation, special attributes at such constructs are used to distinguish between them in order to restore the correct construct. Edges in the CRL graph could originate from two different VHDL language constructs: loops or `if-then-else` statements. Because the loop transformation (cf. Section 4.3.2) has converted loops into self-recursive routines, edges between basic blocks uniquely identify `if-then-else`-statements. Analogously to VHDL processes and subprograms, CRL routines which have been generated by the loop transformation, are flagged with a special attribute for their identification. Besides that, instructions inside basic blocks of the CRL graph directly map to VHDL statements. The runtime complexity and memory consumption of AbstractVhdlGenerator are asymptotically the same as for the pipeline generator.

Maksoud [Mak07] gives details about the reconstruction algorithm.

10.6 Implementation Complexity

The implementation of the different tools has been a technically ambitious project because the goal always was to accept the whole synthesizable sub-

Table 10.2 – VHDL Derivation Tool Set: lines of code

Tool/Library	Lines of Code			
	Blank	Comments	Code	Total
libvhdlanu	656	403	2 740	3 799
Vhdl2Crl2	6 962	8 452	28 385	43 799
VhdlTimingDeadCodeEliminator	417	568	866	1 851
VhdlDomainAbstractor	403	500	1 022	1 925
VhdlProcessReplacer	69	88	170	327
AbstractVhdlGenerator				
PipelineAnalyzerGenerator	2 183	2 143	8 558	12 884

standard of VHDL. At the start of the project, it has been decided to develop a new parser tool, Vhdl2Crl2, from scratch because existing tools either do not accept the fully synthesizable VHDL subset or do not provide access (in the form of a programming interface) to the generated intermediate representation. The design and development of such a new parser for a complex language like VHDL together with the static analyzers, transformation and generation tools has been a complex task which is reflected in the following statistical data.

Table 10.2 outlines the size of the tool implementations in terms of code lines, where blank, comment, and actual code lines are listed in separate columns alongside a total sum in the rightmost column. Each row represents a tool or library. The two generator tools AbstractVhdlGenerator and PipelineAnalyzerGenerator share the CRL file reading functions and internal data structures. Therefore, they are represented by one single entry in the table because both are implemented in the same executable which supports two different operation modes. All in all, the presented implementation consists of 64 697 lines of code (without the contribution of the static analyzers). Vhdl2Crl2 and the combined generator tool are the largest software pieces in the bundle with 43 799 lines and 12 996 lines, respectively. But the 3 799 lines of code for libvhdlanu indicate the complexity arising from VHDL expressions and types. With an average of 1 367 lines, the transformation tools are more compact. Alongside the representation of VHDL models as control-flow graphs using CRL, their examination and modification is supported by the

Table 10.3 – VHDL Derivation Tool Set C++ classes and files

Tool	Files	C++ Classes
libvhdlanu	10	2
Vhdl2Crl2	197	97
VhdlTimingDeadCodeEliminator	7	2
VhdlDomainAbstractor	7	3
VhdlProcessReplacer	3	1
AbstractVhdlGenerator		
PipelineAnalyzerGenerator	15	44

libcrl2 (cf. Section 10.2.4). The code size of the combined generator tool (12 884 lines) lies between the VHDL compiler's and transformation tools' sizes. Code sizes of the static analyzers are not given here as they have been solely developed by Schlickling. He lists corresponding information about them [Sch13].

A more structural metric to depict the size of the implementation is the distribution of the source code over files and C++ classes. Analogously to the code line comparison, Vhdl2Crl2 is the by far largest software in the derivation tool set with 197 files and 97 class declarations. The majority of these files and classes, 178 and 93 respectively, is used for the syntax tree representation of the parser (cf. Section 10.2.1). In general, the distribution of files and classes to tools/libraries exhibits the same trends as the above described code line metric.

10.7 Implementation Restrictions

The implementations of the tools presented throughout this chapter still have a proof-of-concept status, i.e., they should proof the feasibility of the proposed methods but do not claim to be already industrially usable. In consequence, there are some restrictions to the input VHDL code which are listed in the following.

First of all, *only the synthesizable subset of VHDL* [IEE99b] is supported by

Vhdl2Crl2 although this actually is not a hard restriction. As the overall goal is to enhance existing methods for timing analysis on realistic hardware, designs that are not synthesizable will never be examined. One exception remains: some synthesis tools might support language constructs which are not synthesizable even though this contradicts the purpose of the synthesizable VHDL standard.

Besides this, there are other restrictions to the analyzed design which are listed in the following:

- Sensitivity lists of processes must not contain references to single array elements. The current development state of the VHDL parser does not yet accept those constructs.

- Calls to functions that return a record type must only occur on the right hand side of an assignment. Currently, this would break the elaboration step within Vhdl2Crl2 (cf. Figure 10.1 on page 205).

- Calls must not contain actual parameters that are complex expressions in which records or record elements are involved. This is not supported by the implementation of the elaboration step.

- Function symbols equal to the operator symbols provided by the IEEE library, e.g., '+' or '−', cannot be used for function overloading. Usually, the operator symbols from the IEEE library are not used as function calls (prefix notation), so the time needed to resolve such overloading can be economized.

- Constants of type record can only be initialized using ordered named association. No unnamed or unordered association is allowed.

- Arrays of records are not supported. The current state of the elaboration phase within the VHDL compiler currently does not support the collapsing of record structures that are elements of arrays.

- Data types using `up/downto` slices where the bounds are computed using complex expressions are not supported by VhdlDomainAbstractor. Complex in this context means that no nested expressions are used.

All restrictions from this list are not impossible to fix up in general and could be overcome with more or less effort.

10.8 Summary

This chapter describes the implementation of the VHDL Derivation Tool Set that consists of the following tools:

- *VHDL compiler*:
  The tool Vhdl2Crl2 translates designs into the intermediate representation format (CRL) to enable the processing with analysis, transformation and generation tools.
- *Static analyzers*:
  Schlickling [Sch13] provides three abstract interpretation-based static analyzers: VhdlResetAnalyzer, VhdlAssumptionBasedModelRefiner and VhdlSlicer. They are used to explore the VHDL model and to incorporate assumptions (as fixed signal assignments).
- *Transformation tools*:
  Three model transformation tools are presented: VhdlTimingDeadCodeEliminator effectively removes code that has been marked dead by the assumption-based model refiner. Implied code simplifications like empty basic block removal and block merging are supported. Domain abstractions can be invented by source-to-source type changes using the VhdlDomainAbstractor. The tool requests needed operators from the user for the particular target domain and supports different application scopes, i.e., the transformation effect can be restricted explicitly to a given set of identifiers. Finally, the replacement of VHDL processes by custom simulation routines is supported by the VhdlProcessReplacer.
- *Generator tools*:
  Two different generator tools are available. PipelineAnalyzerGenerator is able to convert a timing model in CRL format to an aiT-compatible pipeline analyzer. The nondeterministic property of the input timing model is explicitly supported. AbstractVhdlGenerator reconstructs an abstract VHDL variant of the timing model that is employed in its validation using interval property checking (cf. Chapter 8).

For each presented tool, its structure and underlying algorithmic details are described. All tools have been implemented in C/C++.

Afterwards, some statistical data about the implementation are given in the form of lines of code and number of code modules, files, class declarations, etc.

Although the fully synthesizable VHDL standard is supported by the tools, in general, there are some restrictions which are listed in Section 10.7. All restrictions from this list are not impossible to fix up and could be overcome with more or less effort.

11

Experimental Results

"Not everything that can be counted counts, and not everything that counts can be counted."

(Albert Einstein)

11.1 Overview

This chapter describes experiments with the VHDL derivation tool set implementation as described in Chapter 10. Corresponding results are discussed and evaluated. They have been obtained for the following VHDL designs:

- a superscalar DLX variant similar to a PowerPC 603e,
- the SPARC V8 architecture based LEON2 processor,
- a memory controller used within modern avionics systems and
- two representative automotive processors.

As described in Chapter 10, the developed tools have a proof-of-concept status, i.e., showing the general applicability of the whole derivation approach. Therefore, some of the designs make use of VHDL language features which are currently not supported by the transformation and generator tools (cf. Section 10.7). Particular restrictions are mentioned accordingly throughout this chapter.

This chapter is structured into three parts. First, the architecture of the *superscalar DLX* is described because the runtime and space consumption experiments which are detailed in the subsequent Section 11.3 have been done on the VHDL model of this architecture. After that, Section 11.4 discusses achievable space compaction rates for the resulting timing models compared to the original input code. The precision of the derived models concerning computed WCET bounds is presented in Section 11.5. Finally, Section 11.6 contains an evaluation of the industrial applicability of the derivation approach together with a general summary of the experimental results.

11.2 Complexity of VHDL Specifications

VHDL is a complex language because of the expressiveness of this language, i.e., hardware circuits may be described in different domains. Within one domain, there are different levels of abstractions (cf. Section 5.2.1). To have a concise formal specification of hardware circuits, the register-transfer (RTL) level would suffice because the hardware's behavior can be specified on the granularity of a processor clock cycle. But even on the RTL level, VHDL offers

flexibility on the syntactic level. By this, there are language constructs that are difficult to distinguish for a VHDL parser. For example, it is not clear whether two redundant operators for array accesses, namely $a[i]$ and $a(i)$, are needed. The latter variant cannot be distinguished from a function/procedure call without a context-sensitive lexicographic analysis. Figure 11.1 on the next page shows the graphical representation of the generated parser automaton of Vhdl2Crl2. Due to the graph's scaling factor, it is not meant to present more details about the implemented grammar. Instead, the visualization underlines the complexity of the VHDL syntax because the large number of edges in the graph overlie the nodes that represent parser states.

In the following, a superscalar DLX architecture is presented which is used in the experiments described in the subsequent sections. After that, the sizes of different VHDL models which are available to the author, are discussed.

11.2.1 Superscalar DLX

The superscalar DLX machine is an implementation from the Technical University of Darmstadt [Hor97]. It is based on the DLX presented by Hennessy [HPG06] and conceptually similar to a Freescale PowerPC 603e [Fre02]. Figure 11.2 on page 239 illustrates the functional units of the DLX and data flow between them.

Starting with the instruction fetch handling, the superscalar DLX design has a direct-mapped instruction cache with a size of 64 byte. This is a small cache capacity and not realistic for such a processor. For example, the L1 instruction cache of the Freescale PowerPC 603e is much bigger with a size of 16 kilobyte. Although this has no direct impact on the runtime and memory consumption performance of the derivation tools, it is of interest for the precision of the resulting timing model. The impact of inaccuracies in the pipeline model to the computed WCET bound is smaller because the processor core has to wait for data accesses more frequently due to the small cache size.

Additionally, there is a four-entry branch-target-buffer (cf. Section 3.4) for the prediction of branch targets. Instructions can be fetched by the instruction fetch unit from both of these two sources with a throughput of up to two instructions per clock cycle. Fetched instructions are then stored in two buffers (cf. *Instruction A* and *Instruction B* in Figure 11.2 on page 239). In order to increase the overall processor performance, prefetching over computed branches whose target is not directly known to the fetch unit triggers one

Figure 11.1 – Bison generated state automata of Vhdl2Crl2

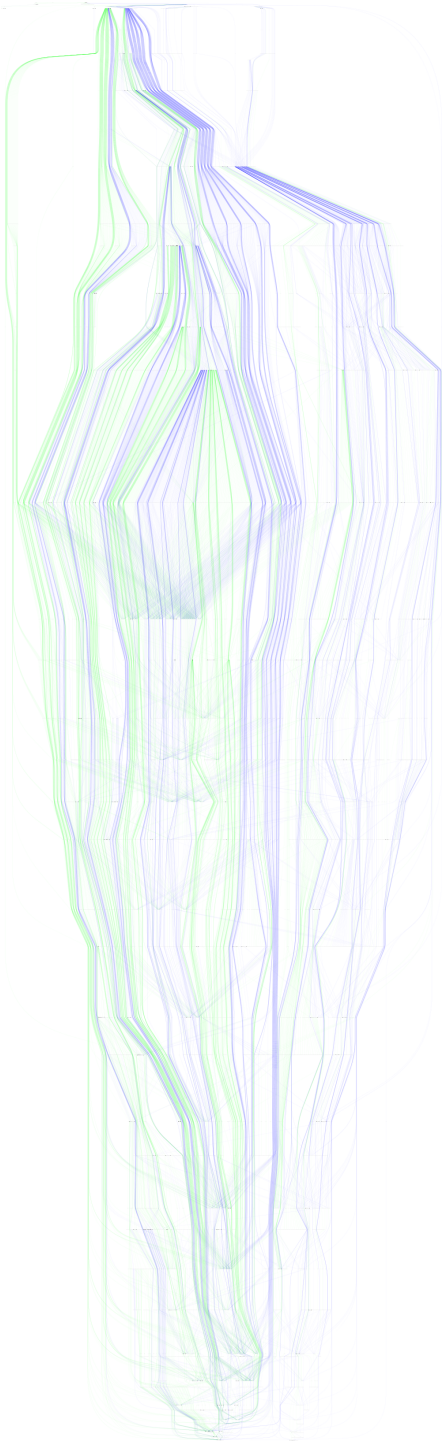
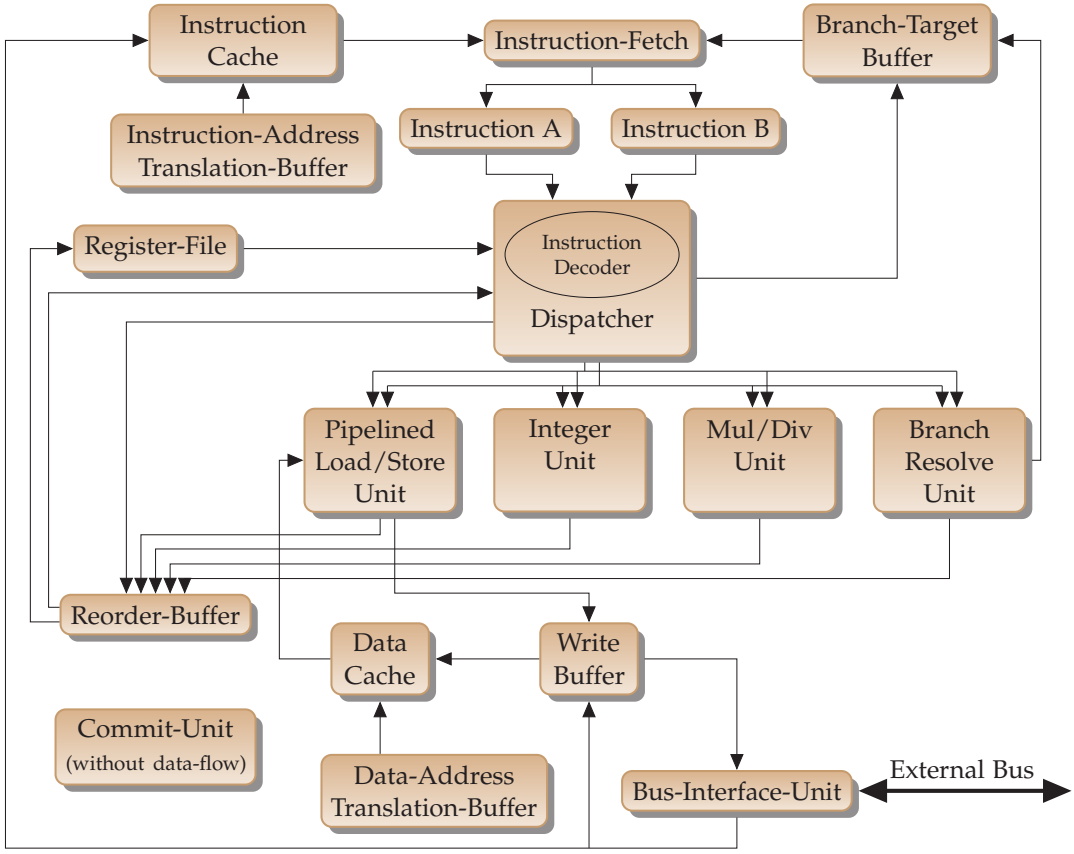


Figure 11.2 – Superscalar DLX architecture – Data flow



level of speculative execution. In contrast, unconditional branches are directly folded, i.e., removed from the instruction stream and fetching is redirected appropriately.

The dispatch unit then decodes fetched instructions and determines the particular responsible execution unit. Additionally, current values of the instruction's source operands are retrieved and rename registers are allocated. The unit is able to dispatch two instructions that are thereby appended to the internal FIFO queue of the reorder buffer, per clock cycle to the execution units. Depending on the used addressing mode, the 32 registers (with 32 bit width each) or the memory cells serve as operands for the instructions.

After dispatch, the four execution units actually implement the instruction semantics. For memory accesses, there is a load/store unit mainly served by the direct-mapped L1 data cache which has the same size as the L1 instruction

cache (64 byte). Store operation performance is optimized by the existence of a write buffer that can process one write so that only a second write would stall the pipeline. Arithmetic/logical and multiplication/division integer operations are executed on the integer and multiplication/division unit respectively. Furthermore, there is a branch resolve unit that manages the resolution of the branch prediction which has been triggered by the prefetching mechanism. Mispredicted branches cause a complete flush of all instructions executed under speculation in the pipeline. Otherwise, in the correct prediction case, all speculatively executed instructions are allowed to write back their changes to registers/memory.

By design, the DLX machine supports out-of-order execution because the four execution units work independently from each other. Nevertheless, the program semantics has to be preserved, so instructions have to retire in-order. This program induced order on the instruction stream is enforced by the reorder buffer as instruction retirement of up to two instructions per clock cycle is performed there from a FIFO queue.

Memory accesses that miss the L1 data cache are forwarded to the bus interface unit which then schedules corresponding transactions on the external bus. Incoming data beats (instruction fetches or reads) are forwarded by this unit to the instruction and data cache. For data reads, incoming data bytes from the bus interface unit are received by the write buffer and bypassed to the data cache because a clash checking against pending writes is done here. Besides this, the superscalar DLX design supports virtual memory by two address translation buffers (one for each cache), precise exceptions, and interrupt handling.

11.2.2 Code Size Comparison

The VHDL specification of the *Superscalar DLX* design is smaller than other complete processor specifications that provide the same architectural features within their pipeline. It consists of two files: `DlxPackage.vhd` and `Dlx.vhd` where the first declares the internal data structures and the latter actually implements the design. The total number of code lines is around 5 000 with a high amount of comments: around 1 100 lines which is about 22 % of the total number of lines. As presented by Table 11.1 on page 242, industrially used processors are much bigger with respect to the specification size in lines.

For example, the *Avionics MCU* represents a memory controller unit (MCU) developed for avionics systems. Its concrete name is hidden due to a nondisclosure agreement with the manufacturer. With 18 986 lines of code in total, this MCU consists of around 3.7 times more lines than the superscalar DLX. The documentation contributes with 18.4 % to the code lines.

The *LEON2* processor, an open-source implementation of a SPARC V8 from Gaisler Research [Gai05], consists of about 14 times more lines (69 144) than the DLX design. Here, only about 10 % of the code is documentation.

The listed *Automotive CPU 1* is a modern coprocessor of a widely used automotive system whose concrete name cannot be stated due to nondisclosure agreements with the manufacturers. It is implemented within 31 981 lines of code in total with about 17 % of comments.

Automotive CPU 2 represents a modern processor for automotive systems. Its VHDL specification consists of 164 476 lines (about 33 times bigger than the DLX design) in total with a documentation ratio of rounded 16.7 %. Within this comparison of different processors, it is the most complex and powerful one regarding automotive applications. Throughout the experiments, only parts of the memory subsystem of this processor (providing dummy declarations for missing module dependencies) have been processed because some third party libraries used in the design have not been provided by the manufacturer. The memory subsystem is a suitable part of the design that could be processed separately. This part (*Automotive MMU*) consists of around 8 400 lines of code in total with 1 285 comment lines and 1 067 blank lines.

So, VHDL specifications of modern and complex hardware components tend to be large as it is underlined by this small statistic in Table 11.1 on the following page. The total number of lines is additionally sketched in Figure 11.3 on the next page.

11.2.3 Structural Size Comparison

The structure and style among VHDL designs varies greatly due to the flexibility of the language itself. Although the superscalar DLX is similar to the Freescale PowerPC 603e [Fre02], its specification only consists of one entity and one package, i.e., different functional units are not separated into different design entities. Additionally, the superscalar DLX specification often makes use of combinatorial logic (cf. Section 9.3.2) which allows a more compact

Table 11.1 – VHDL design size comparison

VHDL Design	Blank in lines	Comment in lines	Code in lines	Total in lines
Superscalar DLX	718	1 108	3 196	5 022
Automotive MMU	1 067	1 285	6 048	8 400
Avionics MCU	3 783	3 500	11 703	18 986
Automotive CPU 1	4 058	5 637	22 286	31 981
LEON2	5 793	7 253	56 098	69 144
Automotive CPU 2	20 719	27 326	116 431	164 476

Figure 11.3 – VHDL design size comparison diagram

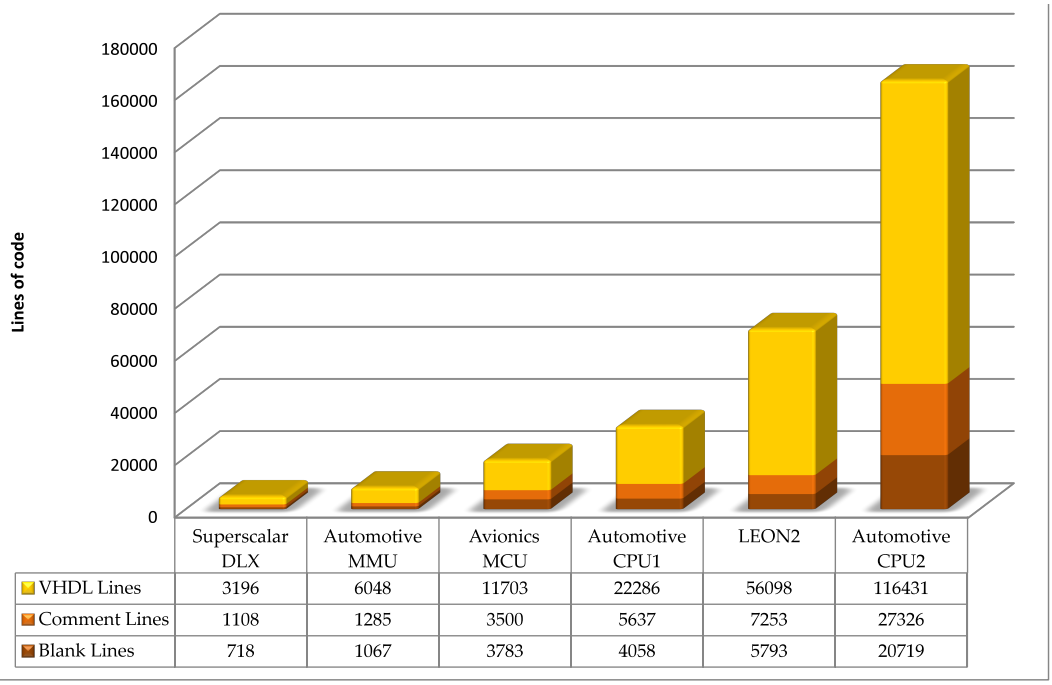


Table 11.2 – Structural VHDL design size comparison

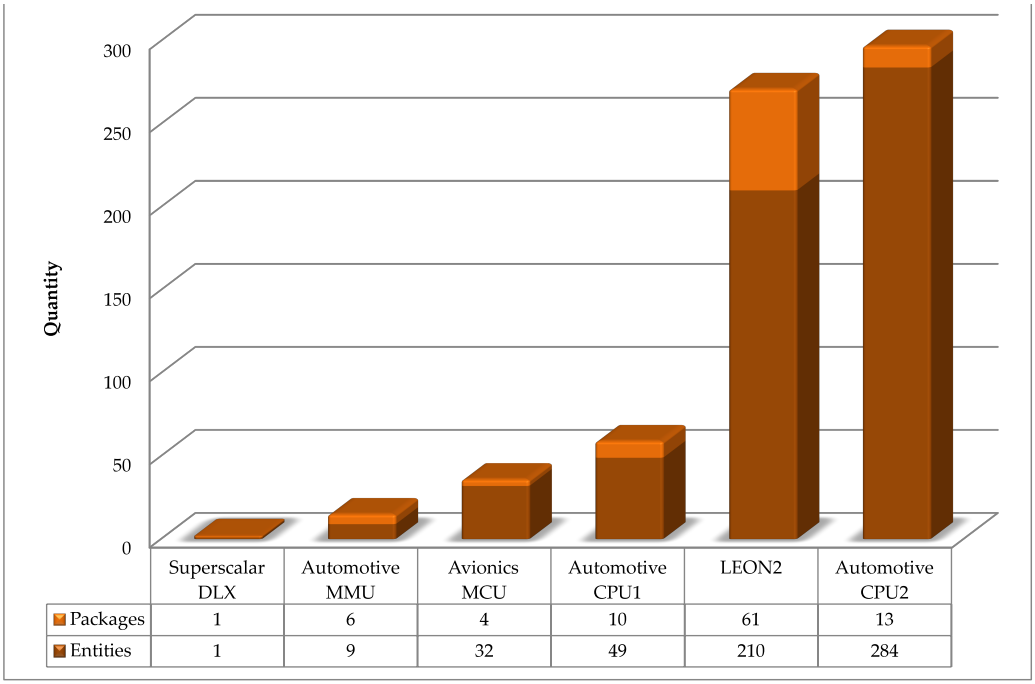
VHDL Design	Entities	Packages	Total Units
Superscalar DLX	1	1	2
Automotive MMU	9	6	15
Avionics MCU	32	4	36
Automotive CPU 1	49	10	59
LEON2	284	13	297
Automotive CPU 2	210	61	271

but less readable design. In contrast, even the avionics memory controller and automotive CPU 1 already consist of 32 entities with four packages and 49 entities with 10 packages respectively. The design of the LEON2 is split into 284 entities and 13 packages. By this, it provides more entities than the automotive CPU 2 (210 entities and 61 packages) despite being smaller in terms of code size (lines of code). The MMU part of automotive CPU 2 consists of 9 entities and 6 packages. But the number of packages in the LEON2 specification is much smaller than in automotive CPU 2, namely 13 compared to 61. Table 11.2 again summarizes the described structural size characteristics of the examined specifications. The bar diagram in Figure 11.4 on the next page gives another view on the comparison explained in this section.

11.3 Derivation Process Complexity

This section shows the results of runtime and space consumption experiments with the tools in the VHDL derivation tool set. The goal is to show their applicability on whole processor specifications even in the state of a proof-of-concept implementation. The developed static analyzers, i.e., VhdlResetAnalyzer, VhdlAssumptionBasedModelRefiner as well as the VhdlSlicer, are not evaluated in the scope of this thesis. Schlickling [Sch13] conducts similar experiments regarding these tools concerning runtime and space consumption. Throughout the next section, runtime performance and memory consumption measurements of Vhdl2Crl2, VhdlTimingDeadCodeEliminator,

Figure 11.4 – Structural VHDL design size comparison diagram



	Superscalar DLX	Automotive MMU	Avionics MCU	Automotive CPU1	LEON2	Automotive CPU2
Packages	1	6	4	10	61	13
Entities	1	9	32	49	210	284

VhdlDomainAbstractor, and VhdlProcessReplacer are described and evaluated. These experiments have been conducted on a Linux workstation equipped with an Intel Core2 Duo (2.66 GHz) and 8 Gigabyte of DDR main memory.

11.3.1 Tool Execution Time Experiments

Vhdl2Crl2

The execution time of Vhdl2Crl2 has been measured on the VHDL specifications described in Section 11.2.2: the superscalar DLX, the avionics memory controller, automotive CPU 1 and MMU as well as the LEON2 processor. Besides the total execution time, the contribution of the parsing and elaboration phase is given. Depending on the input model, the runtime of Vhdl2Crl2 varies between 0.5 min (for the DLX) and 16 min (for the LEON2). The remaining designs could be processed within 2.7 s (avionics MCU), 6.5 s (automotive CPU 1) and 2.5 s (automotive MMU). Concluding from the original size of the model, a higher execution time would have been expected due to the

Table 11.3 – Vhdl2Crl2 runtime distribution

VHDL Design	Parsing	Elaboration	Total Runtime
	in s	in s	in s
Superscalar DLX	11.02	21.26	32.28
Automotive MMU	69.47	82.68	152.15
Avionics MCU	82.97	78.85	161.83
Automotive CPU 1	198.61	191.16	389.77
LEON2	517.55	497.82	1 015.37

runtime performance scaling of Vhdl2Crl2 as described in Section 11.2. All results of the runtime measurements are presented in Table 11.3. They have been obtained using a little Linux runtime and memory space consumption profiling tool called `proc-time` provided by `http://lilypond.org/`. For illustration, the measured runtimes are additionally shown in a bar diagram of Figure 11.5 on the next page.

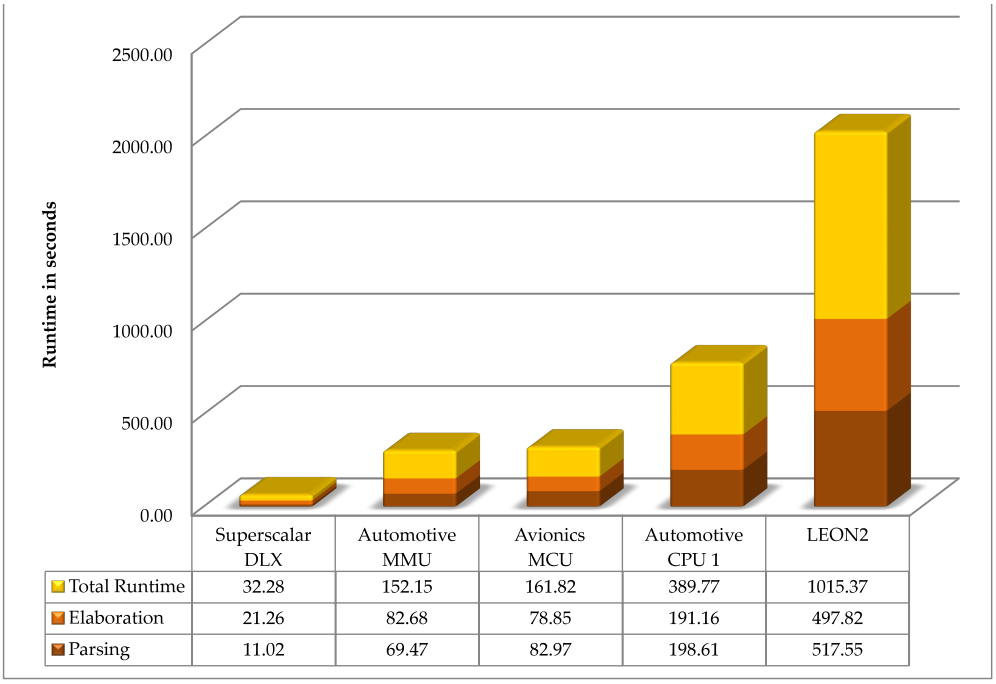
Figure 11.6 on the following page highlights a good scaling factor of Vhdl2Crl2's runtime within the experiments. Depending on the size of the input VHDL model, an almost linear scaling factor can be observed when comparing the different runtimes. Overall, the runtime results show a good performance of Vhdl2Crl2 compared to the complexity of VHDL, both syntactically and semantically. It is remarkable that the elaboration needs nearly as much time as the parsing phase. The main reason is the number of transformations required by the VHDL standard and their particular complexity (cf. Section 10.2.3). Each transformation requires at least one separate pass over the syntax forest.

Despite this, there still is optimization potential. For example, the structure of the syntax tree is close to the actual grammar. Its complexity can be reduced without loosing any important information but increasing the runtime performance.

Transformation Tools

The execution times for the three transformation tools (cf. Section 10.4) have been measured on the VHDL design of the superscalar DLX, the avionics

Figure 11.5 – Vhdl2Crl2 runtime distribution diagram



	Superscalar DLX	Automotive MMU	Avionics MCU	Automotive CPU 1	LEON2
Total Runtime	32.28	152.15	161.82	389.77	1015.37
Elaboration	21.26	82.68	78.85	191.16	497.82
Parsing	11.02	69.47	82.97	198.61	517.55

Figure 11.6 – Vhdl2Crl2 runtime performance scaling function

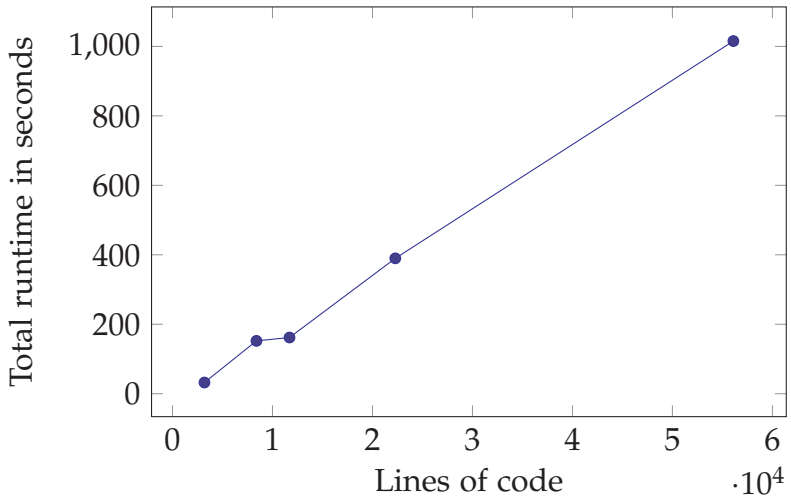


Table 11.4 – Transformation tools runtime distribution

VHDL Design	VTDCE Runtime in s	VDA Runtime in s	VPR Runtime in s
Superscalar DLX	1.38	4.09	1.47
Automotive MMU	1.64	6.63	1.76
Avionics MCU	1.92	6.90	1.83

memory controller and automotive MMU. LEON2 and automotive CPU 1 have not been processed as they make use of `downto/upto` slices where the lower and/or upper bound is not determined directly via constants but more complex expressions. Those are currently not supported by the tools (cf. Section 10.7).

As listed in Table 11.4, the superscalar DLX requires a computation time of 1.38 s to VhdlTimingDeadCodeEliminator (VTDCE). For the model of the automotive MMU and avionics memory controller, the transformation took 1.64 s and 1.92 s respectively. As the runtime is dominated by the iteration over the CRL graph, the concrete inputs can be neglected. Nevertheless, Section 11.4 at least gives the concrete model assumptions for the run of VhdlTimingDeadCodeEliminator on the DLX. For the other two models, the assumptions unfortunately cannot be given without disclosing details about the confidential specifications.

Processing time of VhdlDomainAbstractor (VDA) on the same models took 4.09 s, 6.63 s and 6.90 s for the DLX, automotive MMU and avionics MCU in that order. Input for the tool has been the particular source domains representing a concrete address in the design and a custom defined type `AddressRange` which represents an interval of addresses. But the concrete source and destination domain are not that important for runtime measurements because the complexity of the graph transformations are not directly related to them.

VhdlProcessReplacer (VPR), the tool for replacing VHDL processes by custom simulation routines, needs 1.47 s to process the superscalar DLX design, 1.76 s for the automotive MMU and 1.83 s for the avionics memory controller. The setup of that experiment has been to replace three arbitrary processes by corresponding references to custom implementations. As this tool “only” replaces

processes by references to its custom implementations, the graph iteration is the most dominant factor in the runtime as described in Section 10.4.3.

For illustration purposes, all results are additionally sketched in the bar diagram of Figure 11.7 on the facing page.

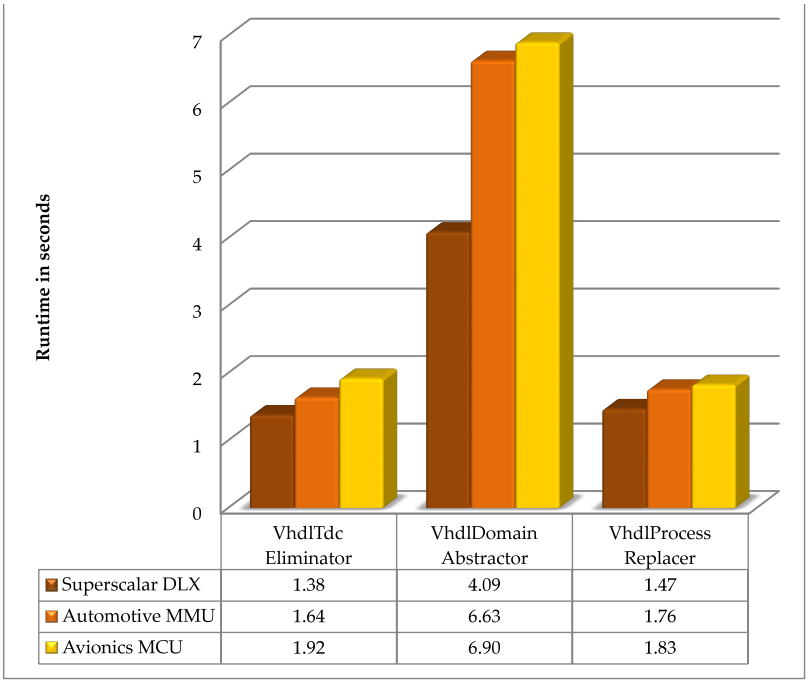
The low execution times, compared to the parsing and elaboration processes, can be explained by a lower complexity of the computations as they operate on the control-flow representation. Moreover, the runtime of the transformation tools actually is not directly influenced by the concrete structure and style of the VHDL design. Mainly the size of the control-flow graph decides about the execution time of the transformation tools because the runtime is dominated by the iteration over that graph which is implemented using a specific AbsInt access library (libcrl2). This library is highly optimized for just that purpose and therefore runtime and space efficient.

The runtime of VhdlDomainAbstractor is slightly higher than the runtime of VhdlTimingDeadCodeEliminator and VhdlProcessReplacer. Although all three tools share the graph iteration (and therefore its runtime costs), the domain abstractor performs more operations on the CRL attributes when querying and changing type information. In contrast to that, the timing dead code eliminator mainly works on the structure of the graph itself, i.e., it changes edges which is a bit cheaper transformation than changing attributes (or their values). The same holds for the process replacer which has to identify matching processes to be replaced and marks them with specific attributes that are read by the pipeline analyzer generator.

Interpreting Figure 11.8 on the next page, the execution times of VhdlTiming-DeadCodeEliminator (VTDCE) and VhdlProcessReplacer (VPR) scale nearly linear with increasing size of the input VHDL model. Based on only three measurement points, it is difficult to deduce a scaling function. Nevertheless, a code review reveals a major dependency of the total runtime to the CRL graph iteration. This fact underlines the scaling assumption.

Because the times for these two tools are below two seconds throughout the experiments, i.e., the runtime difference is small, and therefore the slope of the corresponding plot curve is low. Similarly, the curve for VhdlDomain-Abstractor shows a nearly linear behavior, as well. Due to its higher complexity (see above), the effect of a larger input model is more visible than for the other two transformation tools.

Figure 11.7 – Transformation tools runtime distribution diagram



	VhdlTdc Eliminator	VhdlDomain Abstractor	VhdlProcess Replacer
Superscalar DLX	1.38	4.09	1.47
Automotive MMU	1.64	6.63	1.76
Avionics MCU	1.92	6.90	1.83

Figure 11.8 – Transformation tools runtime performance scaling function

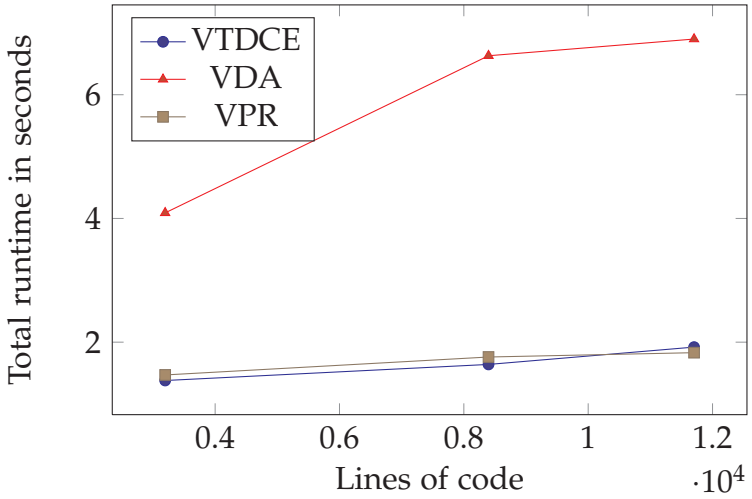


Table 11.5 – Generator tools runtime distribution

VHDL Design	PAG Runtime in s	AVG Runtime in s
Superscalar DLX	7.79	0.60
Automotive MMU	13.51	2.94
Avionics MCU	28.63	7.44

Generators

As described in Section 10.5, the VHDL derivation tool set contains two different generators: one for the creation of aiT-compatible pipeline analyzers (PipelineAnalyzerGenerator) and one for the reconstruction of (abstract) VHDL from a CRL representation (AbstractVhdlGenerator). To demonstrate the runtime behavior of both tools, the transformed timing model (in CRL representation) of the superscalar DLX, the one of the avionics MCU and automotive MMU have been fed to them. For applicability and availability reasons of a timing model which are explained in Section 11.6, both generators have not been tested with the designs of LEON2 and automotive CPU 1. The results of the runtime measurements are shown in Table 11.5 and additionally illustrated in Figure 11.9 on page 252.

With a runtime of 7.79 s for the DLX, 8.51 s for automotive MMU and 31.52 s for the avionics memory controller, the pipeline analyzer generator's (PAG) runtime performance is good. This has been expected because the code generation process is straight forward. On the one hand, all instructions (former VHDL statements) within CRL routines have to be converted to ANSI-C functions. Here, the semantics of this sequential statement execution is the same as in C. Surrounding these functions, a main processor cycle update function, called `single_step` (cf. Section 10.5.1), has to be generated which actually implements the second level of VHDL semantics (cf. Section 5.2.3). The remaining challenge is the realization of the backtracking rules presented in Section 7.3.2 in order to express nondeterminism induced by abstractions of the concrete hardware model. But this generation is computationally not as hard as an abstract interpretation based static analysis on the model or the parsing process. Therefore, the runtime of the pipeline analyzer generator is rather short.

Table 11.6 – Vhdl2Crl2 memory consumption distribution

VHDL Design	Parsing in MB	Elaboration in MB
Superscalar DLX	111	576
Automotive MMU	191	989
Avionics MCU	432	1 680
Automotive CPU 1	756	3 043
LEON2	1 929	6 960

AbstractVhdlGenerator (AVG) is even faster than the pipeline analyzer generator (cf. Table 11.5 on the facing page) because this process is highly supported by information already computed by Vhdl2Crl2. As the task is to regenerate VHDL from the CRL representation including the hierarchy restoration of the used VHDL data structures, the VHDL compiler computes and preserves meta information about that hierarchy (cf. Section 10.2.4). Those information can then be used by the AbstractVhdlGenerator.

Figure 11.10 on the next page shows the runtime performance scaling of the generator tools. Although a scaling function cannot be deduced from only three measurement points, at least this assumption can be stated because the algorithmic implementation mainly depends on the size of the input CRL graph.

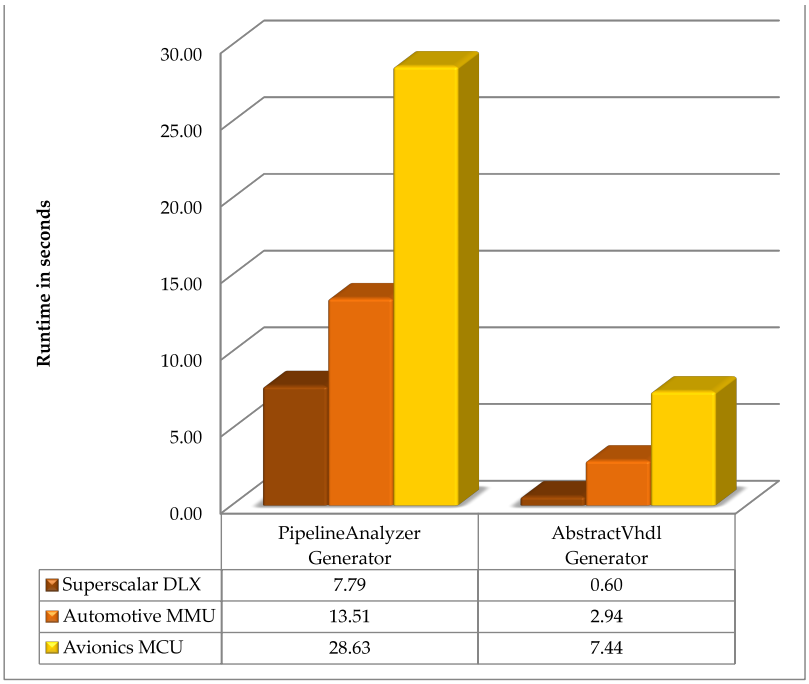
11.3.2 Tool Memory Consumption Experiments

Vhdl2Crl2

During the execution time measurements described in Section 11.3.1, the memory consumption of Vhdl2Crl2 has been recorded. Therefore, the results have been obtained by the same tool as in the runtime measurements, namely `proc-time` provided by `http://lilypond.org/`.

As listed in Table 11.6, the needed main memory for the parsing varies between 111 MB for the superscalar DLX and 1 929 MB for the LEON2. Concerning the elaboration process, the consumed memory ranges between

Figure 11.9 – Generator tools runtime distribution diagram



	PipelineAnalyzer Generator	AbstractVhdl Generator
Superscalar DLX	7.79	0.60
Automotive MMU	13.51	2.94
Avionics MCU	28.63	7.44

Figure 11.10 – Generator tools runtime performance scaling function

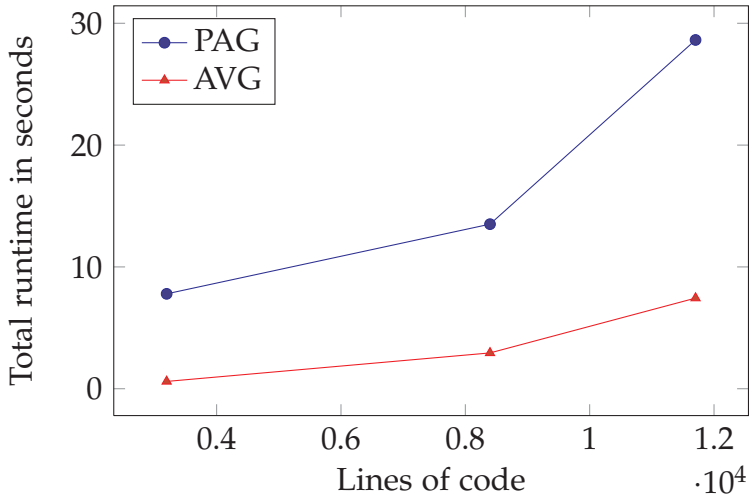
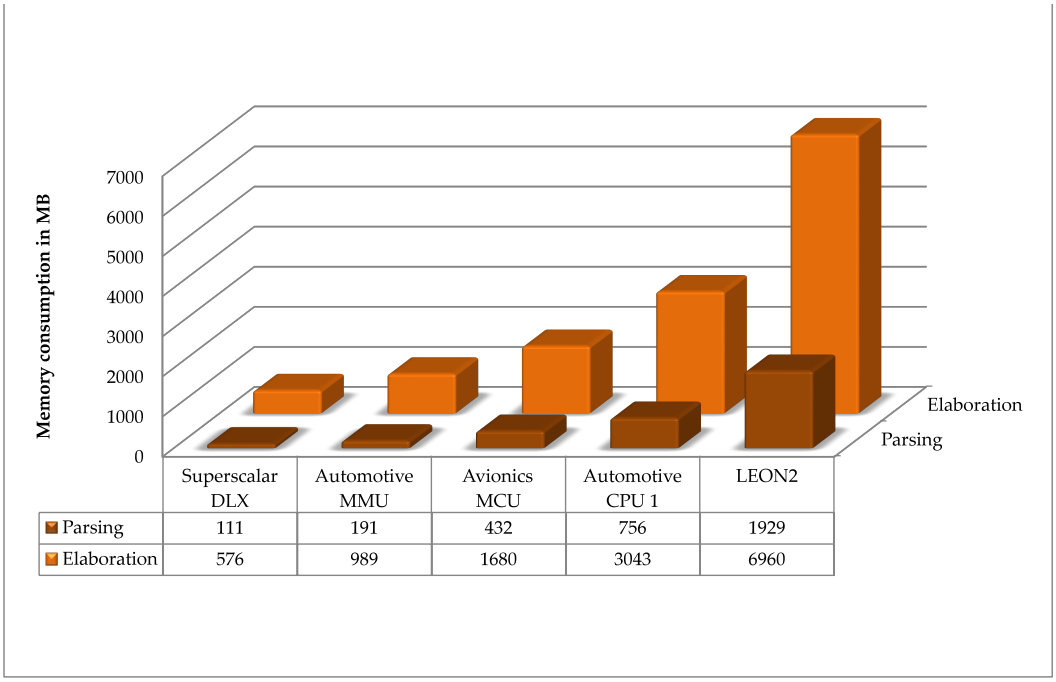


Figure 11.11 – Vhdl2Crl2 memory consumption distribution diagram



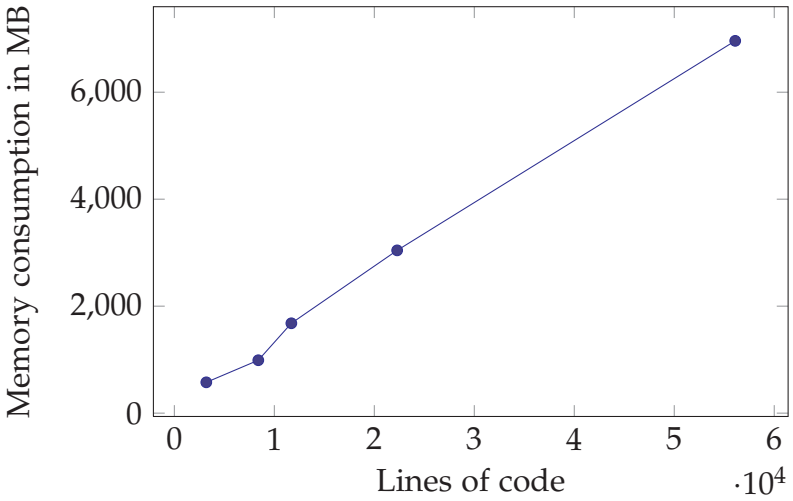
	Superscalar DLX	Automotive MMU	Avionics MCU	Automotive CPU 1	LEON2
Parsing	111	191	432	756	1929
Elaboration	576	989	1680	3043	6960

576 MB and 6 960 MB respectively. Figure 11.11 additionally illustrates these memory consumptions.

As illustrated by Figure 11.12 on the following page, a nearly-linear scaling can be observed for the memory consumption of Vhdl2Crl2 in relation to the size of the input VHDL design. These results correspond to the measured runtime performance scaling.

Summarizing and interpreting the memory experiment results, the parsing and elaboration of processor descriptions can be memory-consuming looking at the more than six Gigabyte needed to process the LEON2 model. Additionally, it can be observed that the memory consumption of the elaboration phase is much higher than the needed memory for the parsing phase. The reason mainly is the embedding of components and generate statements during the elaboration as required by the VHDL standard. Still, these results show weaknesses in the implementation which has space optimization potential especially in the representation of the annotated syntax tree. Nevertheless, it is demonstrated that even whole processor designs can be handled by this proof-of-concept implementation on a workstation with state-of-the-art main memory equipment.

Figure 11.12 – Vhdl2Crl2 memory consumption scaling function



Another interesting comparison is the size of the input VHDL model (in Kilobytes) with the size of the resulting CRL file. For example, the LEON2 model has a size of around 3 MB where the generated CRL representation is larger with a size of 5.3 MB. This increase can be explained by the unrolling of generate statements which are embedded into the elaborated entity `leon`.

Transformation Tools

The memory consumption of the three transformation tools has been measured together with the described runtime experiments (cf. Section 11.3.1), i.e., experiments have been done with the superscalar DLX, the avionics memory controller and the automotive MMU designs. The required memory profiles for them are listed in Table 11.7 on the next page and sketched in Figure 11.13 on the facing page.

As all of the three transformation tools operate on the CRL description of the VHDL model, their memory consumption is dominated by the size of the input model which has to be loaded and remains in memory during runtime. This explains that all tools have the same maximal memory consumption for the same inputs. Overall, the transformation tools have a cheap memory profile thanks to the efficiency of the CRL access library (libcrl2).

Comparing the memory consumption of the tools with the original size of

Table 11.7 – Transformation tools memory consumption distribution

VHDL Design	VTDCE in MB	VDA in MB	VPR in MB
Superscalar DLX	4	4	4
Automotive MMU	11	11	11
Avionics MCU	24	24	24

Figure 11.13 – Transformation tools memory distribution diagram

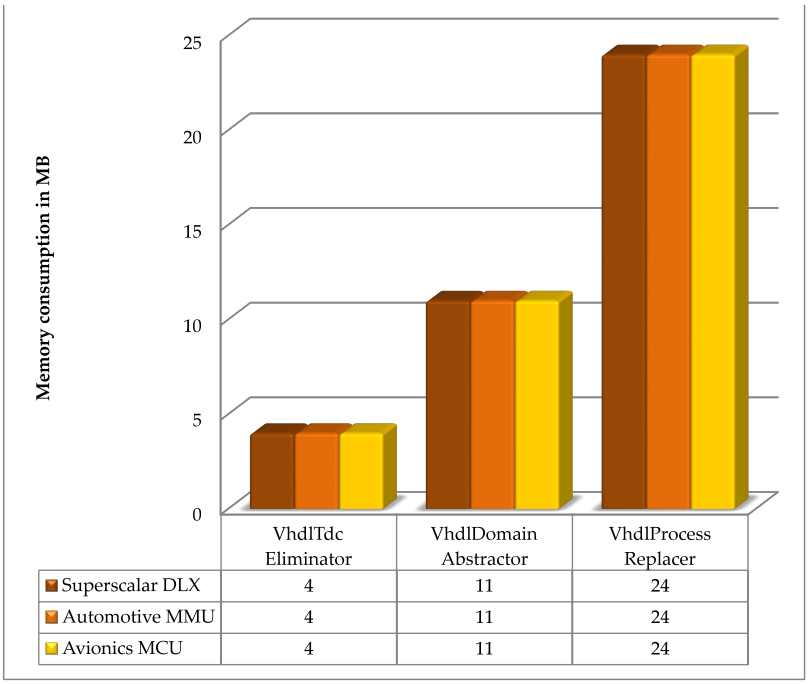
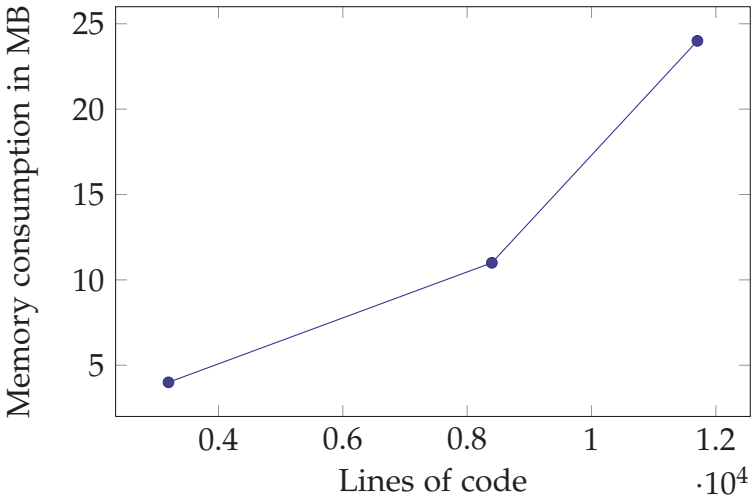


Figure 11.14 – Transformation tools memory consumption scaling function



the VHDL input model as plotted in Figure 11.14, a nearly-linear scaling performance can be observed which is analog to the runtime performance described in Section 11.3.1. As for the runtime experiments, there are only three measurement points rendering a deduction of the scaling function difficult. Examining the implemented algorithms, memory consumption is asymptotically constant except the memory footprint of the loaded CRL data structures. Here, the memory consumption dependency is linear to the size of the CRL graph.

Generators

The memory footprints of the two generator tools, namely PipelineAnalyzerGenerator and AbstractVhdlGenerator, have been examined in the same experimental setup as for the runtime experiments which have been described in Section 11.3.1 on page 250. Results of these are shown in Table 11.8 on the facing page and drawn in Figure 11.15 on the next page.

The PipelineAnalyzerGenerator needs a maximum of around 212 MB memory for generating a pipeline analyzer for the superscalar DLX timing model. For the automotive MMU and avionics memory controller, 585 MB and 802 MB are required respectively. In contrast, the AbstractVhdlGenerator only needs 22 MB to reconstruct the abstract VHDL representation for the timing model

Table 11.8 – Generation tools memory consumption distribution

VHDL Design	PAG in MB	AVG in MB
Superscalar DLX	212	22
Automotive MMU	585	60
Avionics MCU	802	81

Figure 11.15 – Generator tools memory consumption distribution diagram

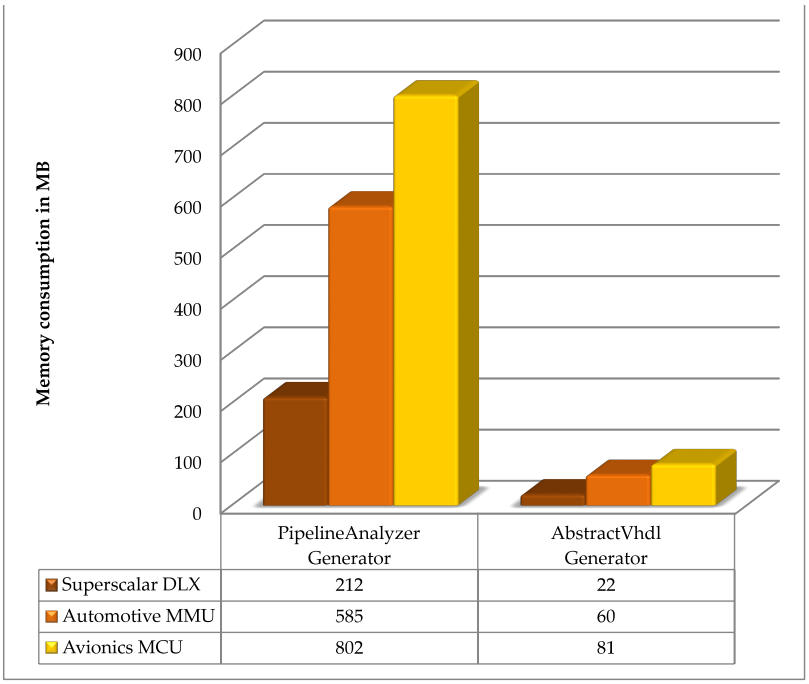
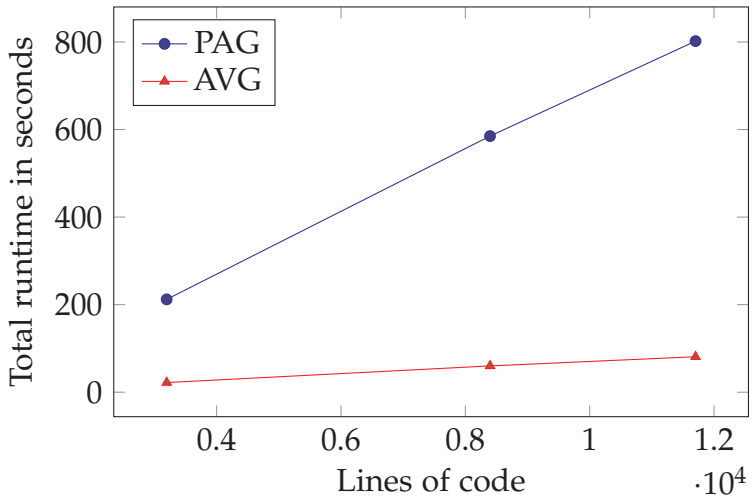


Figure 11.16 – Generator tools memory consumption scaling function



of the superscalar DLX design. Respectively, 60 MB and 81 MB are needed for the automotive MMU and avionics memory controller.

This difference in the memory consumption can be explained by the fact that the VHDL reconstruction process is computationally a bit cheaper than the pipeline analyzer generation. Partly, the needed information to produce VHDL is already contained in the CRL representation. Similar to the required memory of the transformation tools, the generators consume less memory than Vhdl2Crl2. Generally, the memory consumption is expected to scale linearly with a growing size of the input timing model. This assumption is based on the construction of the underlying algorithm whose memory allocation profile is dominated by the data structures storing the CRL representation. Figure 11.16 confirms this assumption.

11.4 VHDL Specification Size Reduction

In general, the compaction rate strongly depends on the actual timing model and its coding style as described in Chapter 9. During the derivation of a timing model for the superscalar DLX design, the CRL representation of it has been passed to the timing dead code eliminator with the results from the static analyzers [Sch13].

Table 11.9 – DLX timing model signal assumptions

Signal/Variable	Value Range
BIU_TransferErrorFetch	+0..+0
BIU_TransferErrorLoad	+0..+0
BIU_TransferErrorStore	+0..+0
DP_IssueIllegalInstrError	+0..+0
IF_TransferErrorFlagA_Input	+0..+0
IF_TransferErrorFlagB	+0..+0
IF_TransferErrorFlagB_Input	+0..+0
TransferError	+0..+0
CU_Inhibit	+0..+0
DP_TakeExternalInterrupt	+0..+0
InterruptRequest	+0..+0
Reset	+0..+0

The different signal assignment assumptions can be roughly grouped into three categories: error, interrupt and reset handling. *Error handling* of a system is actually never considered for a timing analysis because the worst-case execution time is to be determined for normal operating conditions. Safety-critical systems normally are designed for fault-tolerance, i.e., there is at least one redundant, secondary, system that carries over all responsibilities of the primary system in case of any error situation, e.g., hardware failure (exception, power off, etc.) or software detected malfunction.

The second group of assumptions is concerned with the *interrupt handling* which is ignored within the timing model for similar reasons. Interrupts are asynchronous events and their time of occurrence is therefore statically not predictable. If a bound on the execution time of an interrupt handler needs to be computed, the corresponding routines have to be analyzed separately. Then, the corresponding code cannot be ignored for the timing model.

Finally, the *reset signal* is assumed to be inactive. It is ignored for the timing model for the same reason as the error handling.

As a result of the above assumptions, inactive parts of the model have been

marked as "timing-dead" by the assumption refiner. In consequence, those parts can be removed by the timing dead code eliminator whose statistics are shown in Table 11.10. It shows the number of original elements in the CRL graph for each language construct, i.e., for routines, basic blocks, instructions and edges, in its second column. The third column then contains the number of eliminations with the corresponding ratio to the number of original elements in the fourth column. In the last column ("Propagated"), the number of eliminated elements (out of the total eliminations) due to a timing-dead propagation (cf. Section 10.4.1) is given. For the DLX model, the unchanged CRL representation consists of 923 routines, 7 695 basic blocks, 3 978 instructions and 9 476 edges. Based on the input assumption that has been mentioned above, 49 routines, 119 basic blocks, 19 instructions and 140 edges could be removed where the elements of a deleted routine do not count for the corresponding values of deleted blocks and instructions. This corresponds to a decrease of 5.3 % and 1.5 % to the original model in terms of number of routines and basic blocks respectively. Since single instructions are only occasionally removed with respect to the total number of instructions, this ratio is only 0.5 % after rounding. The number of edges in the graph has been reduced by 1.5 %. This small reduction is illustrated in Figure 11.17.

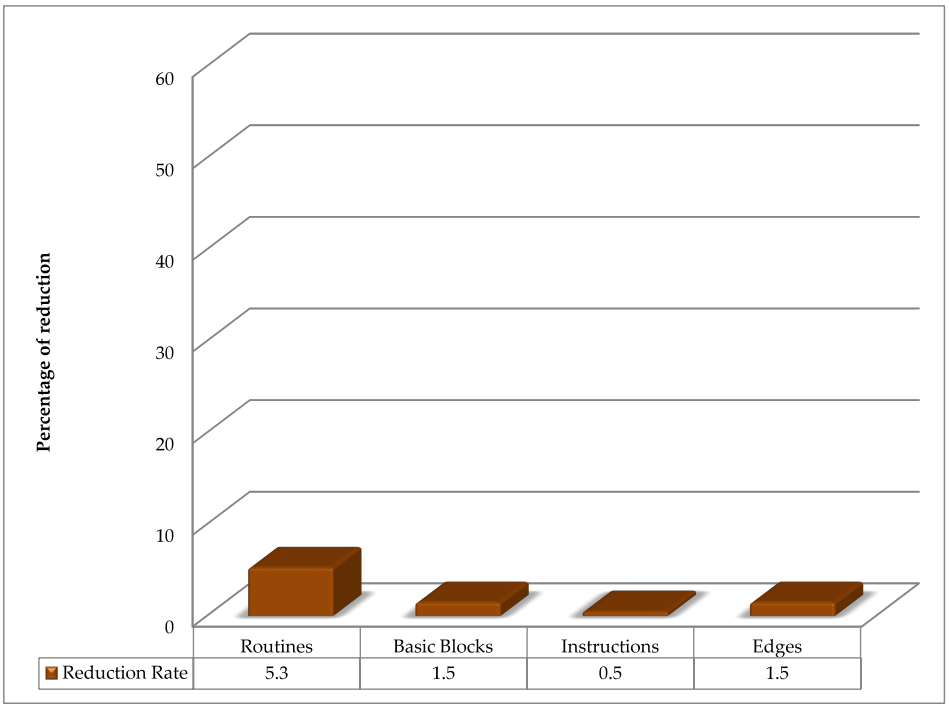
As described in Section 9.3, the compaction rate in general depends on the coding style of the VHDL design. The DLX specification contains concurrent signal assignments that specify the relation between the different control signals and the state of the units. Additionally, there are two processes: one that updates the internal data structures of the functional units and another one that updates the external bus clock. Only small parts of the model use a sequential logic which renders the application of size reducing transformations difficult. This is reflected in the reduction rates described above which are rather small but sufficient for a feasible timing analysis where feasible refers to both computational complexity and precision of computed WCET bounds. If the VHDL model is using more sequential logic and a clear logical separation of functionality in different processes, higher compaction rates are achievable.

For example, the avionics MCU is a more suitable example for a semi-automated timing model derivation due to a higher usage of sequential logic in the design. Its model compaction results are shown in Table 11.11 on page 263. The unchanged model consists of 1 210 routines where 681 – 56.3 % – of them have been eliminated. Regarding the basic blocks, the elimination rate is 25.4 %, i.e., 3 121 out of the original 12 288 blocks have been removed from the model where these blocks are not contained in the eliminated

Table 11.10 – Superscalar DLX model reduction

CRL Construct	Original Elements	Eliminations	Ratio in %	Propagated
Routines	923	49	5.3	11
Basic blocks	7 695	119	1.5	54
Instructions	3 978	19	0.5	0
Edges	9 476	140	1.5	0

Figure 11.17 – Superscalar DLX model reduction diagram



routines. For instructions and edges, the ratio of reduction is smaller with 1.8 % and 3.9 %. Having a much lower compression for instructions and edges means that the number of routine and block removals has been higher. Single instruction removals are rare in comparison because the containing basic block or routine is removed anyway. The only situation leading to such an instruction-only removal is an assignment to a variable or signal which has been assumed or determined to be constant using Schlickling's static analyzers [Sch13]. These results are additionally illustrated in Figure 11.18 on the next page.

The LEON2 specification is another example for a VHDL design not suitable for a semi-automatic timing model derivation process as introduced by this thesis. Although the authors have split the functionality into different processes in order to achieve a separation of logically different functions, they have introduced superfluous dependencies between them by the specification of signal identifiers in the process sensitivity lists. For example, the data cache updating process not only depends on the corresponding data structures. Additionally, any modification of a cell in the instruction cache triggers an execution of the data cache process. Such scenarios are not suitable for a static timing analysis because these sensitivity signals induce control dependencies between the corresponding processes and thereby lead to uncertainties in the analysis results. In that case, it is hard to exclude parts of the model not contributing to the timing behavior without changing the model itself. Schlickling [Sch13] provides details on such "bad dependencies" when statically analyzing VHDL models.

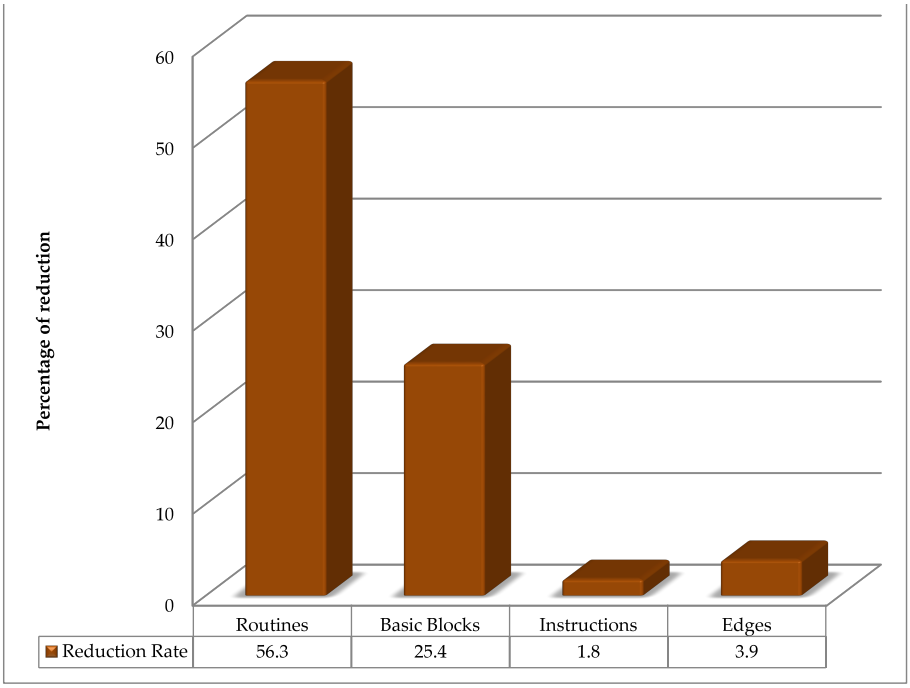
11.5 Precision of Computed WCET Bounds

By construction, the generated pipeline analyzers allow for a determination of safe upper bounds on the execution time of tasks because the underlying timing model has been derived from the formal specification of the target architecture (assuming that the performed transformations and abstractions correctly over-approximate the systems actual timing behavior). In order to be usable for industrial applications, the computed WCET bounds also need to be precise, i.e., the gap between the concrete WCET and the computed upper bound should be small which usually means not to over-approximate more than 10 % to 20 %. Showing that the generated timing analyzers fulfill such precision criteria and thereby being competitive to hand-crafted and

Table 11.11 – Avionics MCU model reduction

CRL Construct	Original Elements	Eliminations	Ratio in %	Propagated
Routines	1 210	681	56.3	36
Basic blocks	12 288	3 121	25.4	1 420
Instructions	8 954	162	1.8	0
Edges	16 785	671	3.9	0

Figure 11.18 – Avionics MCU model reduction diagram



optimized timing models is the objective of the experiments described in this section.

For this, two different kinds of experiments have been done: first, synthetic execution traces of the superscalar DLX retrieved by VHDL simulations have been compared to the predictions of the corresponding generated timing analyzer. Afterwards, the semi-automatically derived timing model of the avionics memory controller is qualitatively compared to an existing hand-crafted model where the term hand-crafted in this case means that the model has been developed by a manual examination of the VHDL design of the memory controller.

11.5.1 Superscalar DLX

In these experiments, some sample programs have been compiled with the C0 compiler [Lei08] which has been developed to produce binaries for the VAMP architecture [Bey05]. As the VAMP and the superscalar DLX both have a compatible DLX instruction set, these binaries can be processed with a VHDL simulator of the superscalar DLX and the existing aiT for VAMP tool. The combined cache and pipeline analyzer of the latter has been replaced with the derived one from the VHDL model allowing a comparison between simulations of test programs with their corresponding aiT predictions. Results of these experiments are shown in Table 11.12 on the facing page and illustrated in Figure 11.19 on the next page.

For each test program whose identifier is given in the first column, the results of its VHDL simulation in core clock cycles (column two) are given together with the corresponding WCET prediction (in column three). In the last two columns, the overestimation which is an indicator of the precision of the underlying timing model, is shown in absolute processor cycles as well as a percentage.

The results can be split into two different groups. The first consists of the programs `nac`, `minmax` and `loops`. All of them are small in code size with at most around 80 lines of code, no complex control-flow graph structure and only a small number of data dependencies. Therefore, the aiT prediction is cycle-accurate on these programs without any overestimation.

The remaining test programs, namely `prime`, `dry2_1` and `edn` are more complex compared to the others building bigger control-flow graphs with more called routines and/or loops where `dry2_1` and `edn` are benchmark

Table 11.12 – Superscalar DLX WCET prediction precision

Test Program	Simulation Time	aiT Prediction	Overestimation	
	in cycles	in cycles	in cycles	in %
nac	310	310	0	0.0
minmax	1 721	1 721	0	0.0
loops	2 024	2 024	0	0.0
prime	16 693	18 082	1 389	8.3
dry2_1	146 290	161 065	14 775	10.1
edn	871 300	966 271	94 971	10.9
Average				4.9

Figure 11.19 – Superscalar DLX WCET overestimation diagram

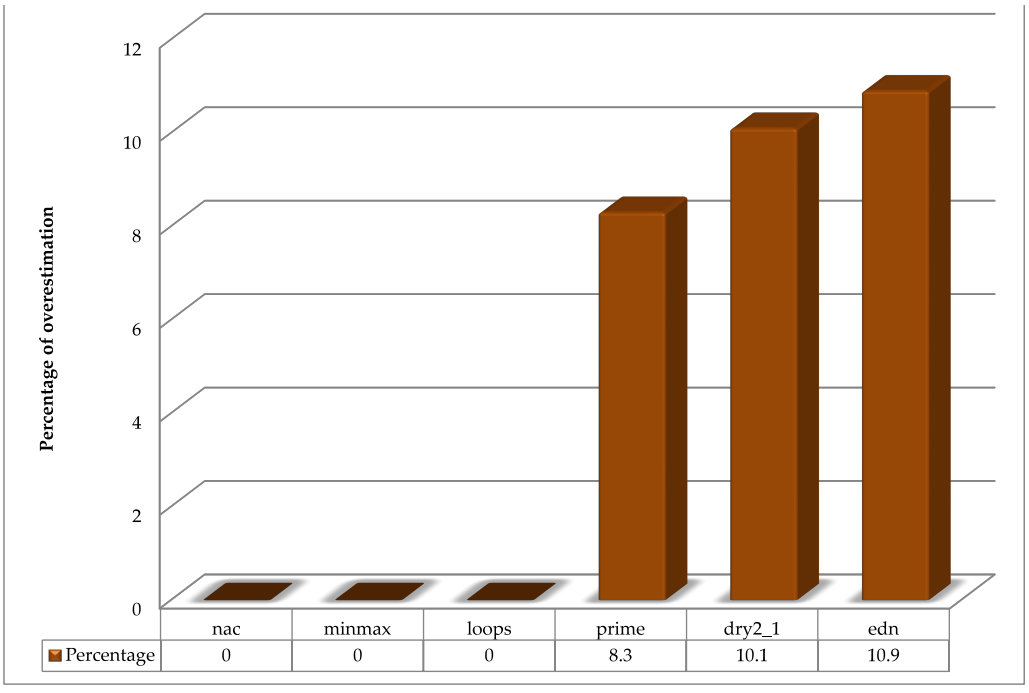


Table 11.13 – Avionics MCU WCET prediction precision

Test Program	Legacy Prediction	Prediction	Overestimation	
	in cycles	in cycles	in cycles	in %
nac	1 168	1 168	0	0.0
minmax	4 601	4 601	0	0.0
loops	3 711	3 711	0	0.0
prime	104 201	104 374	173	0.2
dry2_1	34 942	35 022	80	0.2
edn	144 206	144 421	215	0.1
Average				0.1

programs, i.e., all execution paths through these programs are typically contained in the worst-case path. Here, there is an overestimation of 8.3 %, 10.1 % and 10.9 % respectively. Reasons for these are either analysis induced ones, i.e., precision losses due to control-flow joins, or timing model induced ones, i.e., precision losses due to the employed abstractions. Actually, the encountered precision of the timing model of the superscalar DLX could have been expected because the VHDL design is the simplest one among the examined designs. Therefore, nearly everything can be modeled precisely which restricts the potentials for abstraction losses.

11.5.2 Avionics Memory Controller

The last section has provided a comparison between VHDL simulations and the aiT prediction on some test programs. Another interesting point is to show that the generated timing models are competitive regarding the precision of the computed time bounds. Because there exists a hand-crafted timing model of the avionics memory controller incorporated into the aiT framework, it can be replaced by the semi-automatically derived one in order to see the effect on the timing results. The results of these experiments are shown in Section 11.13 and Section 11.20 where the latter shows the processor cycle difference between the derived model and the corresponding legacy prediction.

Figure 11.20 – Avionics MCU WCET overestimation diagram

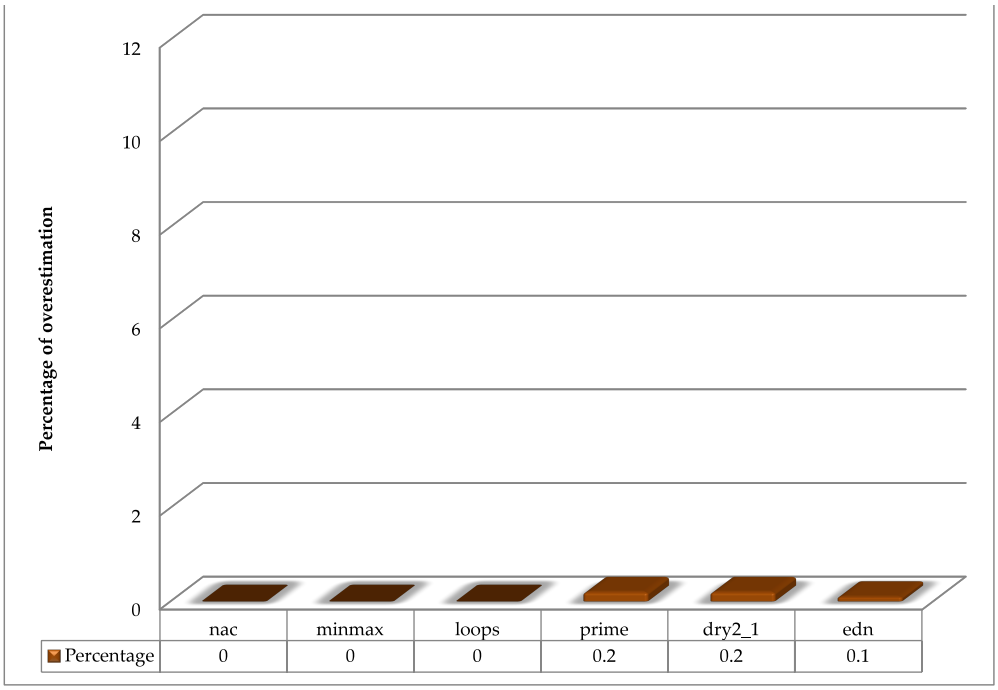
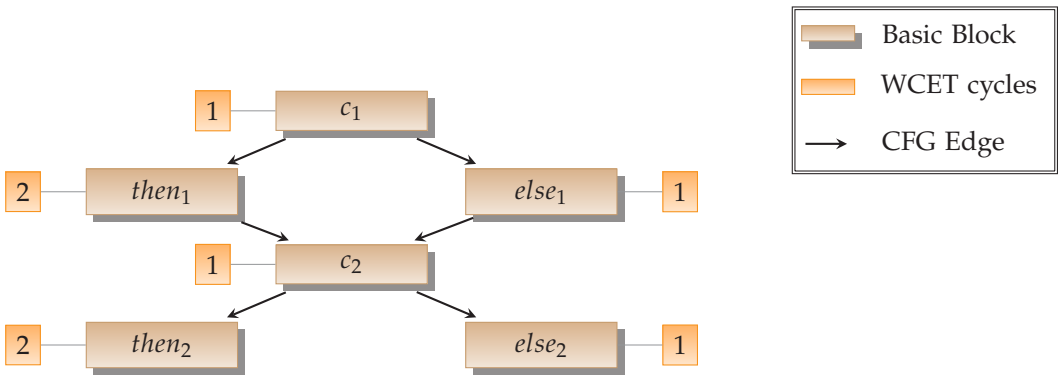


Figure 11.21 – Timing over-estimation for alternative conditionals



For the sake of completeness, it has to be mentioned that the test programs listed in Table 11.13 on page 266 originally were identical to the ones used for testing the timing analyzer for the superscalar DLX. But as mentioned in the last section, the C0 compiler [Lei08] which only accepts a restricted subset of the C language, has been used for executable creation. Two major restrictions are the lack of pointer arithmetic whose occurrence is just rejected by the C0 compiler, and floating-point instruction support, i.e., data-dependent loops are generated for multiplications and divisions. For simplification, such loops have been annotated with global loop iteration default (cf. [Hec10]). By this, the resulting execution times of the test programs cannot be compared between the two architectures although this is not the intention of this experiment, anyway.

In principle, both models for the avionics MCU should produce the same results as they are based on the same VHDL specification. The only difference is that the hand-crafted model has been developed based on manual examination and transformation of the input VHDL specification whereas the newly derived timing model performs the same type of transformations based on the VHDL derivation tool set as it is described in detail in Chapter 10. This assumption is confirmed by the results for the test programs `nac`, `minmax` and `loops` where both models compute exactly the same timing bounds. However, the remaining tests, namely `prime`, `dry2_1` and `edn`, reveal that the newly derived model is a bit more pessimistic. More concretely, the computed timing bounds are 173, 80 and 215 (processor clock) cycles higher than the corresponding bounds computed by the legacy model.

The are two reasons for these differences: imagine a given control-flow graph

as illustrated in Figure 11.21 on the preceding page where there are two successive control-flow splits (due to conditional statements c_1 and c_2) with "then"- and "else"-cases, respectively. For simplification of the description, the timing costs of all basic blocks are annotated to the graph. Another assumption is that the condition c_2 is always inverse to c_1. By this, only two paths remain feasible, namely $p_1 = (c_1, then_1, c_2, else_2)$ and $p_2 = (c_1, else_1, c_2, then_2)$, both equipped with timing costs of five cycles. In the hand-crafted model, these paths actually are exclusively handled which must not necessarily be the case for the generated pipeline analyzer. Here, two additional paths, namely $p_3 = (c_1, then_1, c_2, then_2)$ and $p_4 = (c_1, else_1, c_2, else_2)$, might be feasible, as well, because the static analyzers only determine that the expressions for c_1 and c_2 might evaluate to both possible values, i.e., *true* and *false*. The generated pipeline analyzer would then generate predictions for all four paths through this CFG snippet and the path analysis would determine the most costly path which is p_3 with six cycles. Such a scenario shows that manual optimizations to the timing model might further increase the precision of the timing behavior prediction. Stein [FMC+07] has investigated a similar problem for the path analysis and developed an analysis to identify such mutually exclusive paths in a control-flow graph. A future enhancement of the VHDL derivation tool set therefore might be to adopt and incorporate such an optimization.

Another cause for the higher WCET bounds in the experiments of this section is that the replaced memory controller is only a part of the complete timing model which covers the whole CPU. Even small changes to parts of the model often have side-effects to the global prediction. For example, if the predicted point of time for the termination of a data phase on the system bus changes, this actually affect the retirement of the instruction that has caused this memory request.

11.6 Applicability and Summary

By these experimental results, the general feasibility of the timing model derivation approach has been shown. The automotive and avionics designs are coded in a style close to the actual hardware components, i.e., more "low-level". They are written in sequential logic style which is a more structural description than a combinatorial one. Such a coding style simplifies the timing model derivation a lot and lead to satisfactory results. In contrast, the

LEON2 is not suitable for a semi-automatic derivation process because of its high-level language construct usages and poorly designed data dependencies between processes.

In general, VHDL descriptions tend to be large especially for complete processor descriptions. This is shown in the comparison of different VHDL models in Section 11.2 on page 236. For example, the LEON2 design consists of 69 144 total lines of code and the automotive CPU 2 model is even composed of 164 476 lines. The avionics memory controller is comparatively even larger with 18 986 lines as this model only represents the memory controller of the corresponding system. Such complex specifications are actually hard to examine manually without any tool support. This underlines the motivation behind the derivation tool set implementation.

Concerning the efficiency of the tool implementations, Section 11.3 details about measured computation times and corresponding memory consumptions. Summarized, even complex VHDL models as mentioned above can be handled by the VHDL compiler in acceptable time (< 16 min) and memory footprint (< 7 GB). Later employed transformation and generator tools benefit from the compact and efficient intermediate representation and have processed their inputs in the experiments below 30 s while consuming at most 802 MB.

The combination of assumption-based model refinement and subsequent dead code removal (cf. Section 6.4) has shown to be effective. The numbers of routines and basic blocks of the avionics memory controller model have been reduced by 56.3 % and 25.4 % respectively.

Experimental results show that the quality (in terms of precision and resulting analysis complexity) of the derived timing models varies depending on the employed coding style of the input VHDL design. Minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design are the most prominent and important properties which support and simplify the semi-automatic derivation process and therefore formulate kind of a predictability notion for VHDL constructs/designs. They additionally might be claimed as overall design goals since fulfilling these properties increases the readability of the code which may even help engineers during the functional verification of the hardware design.

Comparing the computational complexity and resulting analysis efficiency of a semi-automatically with a manually derived timing model is similar to the

comparison between hand-written assembler code and a compiler-generated program. The hand-written code can outperform the generated one because of manual optimizations. In contrast, the development time is much shorter using a compiler for generating code. Deriving a timing model is much like the same problem. A hand-written timing model (especially if manually derived from the formal hardware specification) can be more effective in runtime and memory consumption but its development time possibly take much longer than using the semi-automatic derivation process introduced by this thesis. And the automatically generated pipeline analyzer might be further optimized manually if necessary.

Nevertheless, WCET precision results render the whole approach competitive with hand-crafted models as shown in Section 11.5. WCET bounds computed by the superscalar DLX model have been compared to results from synthetic execution traces. They showed an average overestimation of 4.9 %. For the avionics memory controller, WCET bounds of a derived timing model have been compared to the bounds computed by a hand-crafted model. The latter have been developed by a manual examination of the VHDL model. Half of the test cases lead to identical timing bounds and the other half showed an average overestimation of only 0.1 %. These overestimations can be explained by the above mentioned manual tuning of the hand-crafted model.

12

Conclusion and Future Work

"Alea iacta est."

(Gaius Iulius Caesar)

Numerous safety-critical systems are subject to strict timing constraints, i.e., there are deadlines for the execution time of single processes or tasks. Missing a deadline can affect the functional correctness of the system as a whole. Therefore, a sophisticated analysis of its timing behavior is crucial to ensure global functional correctness. On the code level, this demands for a safe and precise worst-case scenario approximation of the execution times of single processes, tasks and interrupt service routines, so-called *worst-case execution times (WCETs)*.

State-of-the-art static timing analyzers, like aiT, determine such timing bounds based on a detailed execution model ("timing model") of the underlying hardware architecture. It includes the instruction flow of the processor pipeline integrated with a cache-behavior analysis as well as the memory access timings of peripheral devices. These timing models are hand-coded starting with a manual inspection of the specific processor/system documentation. Based on traces obtained by hardware measurements undocumented details are then re-engineered and errors in the documentation can be discovered. Unfortunately, this development process is *error-prone* and *time-consuming*.

Nowadays, hardware circuits are automatically synthesized from formal hardware specifications like VHDL or Verilog. Besides a formalization of the functional details, such specifications implicitly contain an execution model that also reflects the timing behavior of the whole system. This enables the derivation of timing models based on their formal hardware specification to simplify the above described error-prone development process. By this, this thesis bridges the gap between hardware circuit synthesis and WCET analysis development. A detailed conclusion of our different contributions are outlined in the following.

12.1 Contributions of this Thesis

12.1.1 Timing Model Derivation

Static analysis and model transformations have been invented by Schlickling and myself to extract the timing information of hardware circuits from its formal specification. Where Schlickling [Sch13] introduces abstract interpretation-based static analysis of formal hardware models, this thesis focuses on the derivation process as a whole. Starting from the hardware model, transformations and abstractions which are based on the results of

static analyzers extract the timing-relevant information, the timing model. In the end, an aiT-compatible pipeline-behavior analyzer can be generated from such a model.

Semantic Level Reduction A prerequisite for the application of both static program analysis and transformations is a common intermediate representation of the hardware model. Since the chosen analysis approach, abstract interpretation, has its origins in program analysis, the hardware model needs to be represented as a sequential program. Thus, this thesis has defined translation rules to convert VHDL language constructs into control-flow entities combined with a framework of generated routines, so that the resulting control-flow representation forms a simulation environment for the original hardware design. On top of that, abstract interpretation-based static analyzers [Sch13] are enabled to examine the model and result communication is possible via a shared intermediate format called *CRL*. This translation from VHDL into a sequential execution model is called *semantic level reduction* and is generic enough to be applied to related hardware description languages like Verilog, as well.

Derivation Process Definition A direct execution simulation of hardware models is generally prevented by the size of these specifications because the computation time for such an input-independent simulation would be too high. Therefore, a distinct process for extracting timing-relevant information from the CRL representation of a hardware model has been defined:

- *Model preprocessing*
  This step mainly removes everything that does not contribute to the timing behavior, e.g., functional details of an adder or multiplier, or configurable features which are disabled by the active system configuration. The result is a significant reduction of the size of the model.

- *Processor state abstractions*
  A fully detailed representation of the system state would lead to a prohibitively large space consumption dependent on the complexity of the system under analysis. For this, employed abstractions either leave out some details of the processor state or approximate them. The level of abstraction mainly depends on the complexity of the analyzed architecture.

Derivation of Workflow Patterns The derivation process described above represents a general methodology. But the particular abstractions and transformations depend on the concrete hardware architecture whose timing behavior has to be modeled so that defining these abstractions remains an intellectual challenge. However, we have identified typical working patterns during the implementation and experiments.

In principle, the timing behavior of a processor is dominated by the (timing-)effect of the instruction flow through the processor pipeline and latencies for memory accesses. Therefore, the resulting timing model needs to represent this flow. Combinations of program slices, model assumptions (represented as assignments of fixed values to signals/variables) and constant propagation effectively render parts of the specification unused, *timing dead*. These parts can be removed in a following step.

Afterwards, abstractions may be applied to further reduce the resulting timing analysis' complexity. We describe sample abstractions together with their application to appropriate VHDL code snippets.

12.1.2 Simulation Semantics

A pipeline-behavior analysis is an abstract (and thereby computable) simulation of a program execution. The theoretical foundation of the generated analyzers is an abstract simulation semantics: we have formalized an operational semantics for the simulation of non-abstracted VHDL models alongside an abstract variant so that the abstract simulation safely approximates any concrete simulation. Employed abstractions can render the model nondeterministic, i.e., the simulation process might compute multiple successors for a given input system state leading to multiple possible execution paths partially with different costs in terms of execution time. The defined abstract simulation semantics is able to cope with such uncertainties and simulates all potential execution paths.

12.1.3 Timing Model Validation

We have shown that a derived timing model per construction correctly represents the timing behavior of the particular hardware. Concerning the correctness of the resulting pipeline-behavior analysis, it remained to argue that the employed model abstractions and transformations do not introduce

unsafe[1] changes to the timing behavior. We therefore presented interval property checking techniques which validate the "timing-semantic" preserving translation from the input design to the timing model.

Additionally, complementing validation techniques are presented where confidence on the correctness is achieved by testing. Measurement capabilities can produce runtime observations at different levels that are compared to the corresponding prediction of the timing analyzer. Example levels are processor core events like cache hits and number of dispatched instructions at a specific execution point or visible bus transaction signals triggered by memory accesses.

12.1.4 VHDL Predictability

Experiments with different VHDL models (cf. below) have revealed that the quality of the derived timing model (in terms of the precision of computed WCET bounds as well as the computational complexity) is influenced by the VHDL coding style. We have described design choices together with their effect on the derivation process and thereby formulated a kind of predictability notion for VHDL language constructs along with advices to the corresponding hardware development. Minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design are the most prominent and important properties which support and simplify the semi-automatic derivation process.

12.1.5 VHDL Derivation Tool Set

We have implemented a set of tools, the so-called *VHDL Derivation Tool Set*, that proves the feasibility and applicability of the proposed timing model derivation workflow and pipeline analyzer generation. The different tools are:

- A *VHDL compiler* for loading designs into an intermediate control-flow representation. This contribution originates from a strong cooperation between Marc Schlickling [Sch13] and myself.

[1] under-estimations

- *Static analyzers* which are based on an analysis framework so that the model can be explored and analyzed in order to identify parts to be removed. This contribution has been done by Marc Schlickling [Sch13] and is only listed here for the sake of completeness.
- *Model transformers* that support the removal of "timing dead" code, the introduction of domain abstractions as well as the replacement of VHDL processes.
- A *pipeline analyzer generator* to automatically generate the above mentioned aiT-compatible analyzers.
- An *abstract VHDL generator* that automatically reconstructs abstract VHDL from a given timing model. This code can then be used for the validation of employed transformations (see below).

12.1.6 Experimental Results

Experiments with the tool implementations have been conducted to underline the industrial applicability of the approach in total.

We examined the following VHDL designs:

- a superscalar DLX variant similar to a PowerPC 603e,
- the LEON2 processor which is based on the SPARC V8 architecture (typically used in space applications),
- a memory controller used within modern avionics systems and
- two representative automotive processors.

Except the first one, all these models represent processors or memory controller specifications that are utilized within real-world safety-critical systems. The superscalar DLX machine is an implementation from the Technical University of Darmstadt [Hor97] that is based on the DLX presented by Hennessy [HPG06]. Although the design is not industrially used, it offers features like out-of-order execution, speculation and branch prediction. Non-disclosure agreements with the particular manufacturers forbid the exposure of the original names for the anonymous avionics and automotive designs.

Runtime and memory consumption experiments show a good performance of the implemented tools along with a linear scaling in the code size on the selected hardware models. Even big processor specifications like a LEON2

(with about 70 000 lines of code) can be translated into their sequential program representation within acceptable time (about 17 min for the LEON2). The memory consumption is high with about 7 GB but this has been expected regarding the complexity of the transformation. Resource consumption of the transformation and generator tools is low compared to the VHDL compiler and dominated by the size of the intermediate representation. Execution times are just below 30 s and the maximal observed memory consumption is 802 MB in the experiments.

During the derivation process employed model transformations are shown to reduce the size of input VHDL models (around 50 % for a modern memory controller of an avionics system) enabling the generation of aiT-compatible pipeline analyzers.

Moreover, there are two different kinds of experiments underlining the competitiveness of derived timing models against hand-crafted ones. For the superscalar DLX, synthetic execution traces retrieved by VHDL simulations have been compared to the predictions of the corresponding generated timing analyzer. Results show an average overestimation of around 10 % for the predictions. Additionally, the semi-automatically derived timing model of the avionics memory controller is qualitatively compared to an existing hand-crafted model where the term hand-crafted in this case means that the model has been developed by a manual examination of the controller design. Computed WCET bounds either are equal or over-estimate the legacy bounds only by a small percentage (less then 1 %).

12.1.7 Summary

> *What is the expected impact of the proposed timing model derivation approach?*

One of the main contributions of this work is to provide the ability to semi-automatically generate the timing analyzers alongside their provable correctness: analyzer generation instead of a manual implementation fastens development times so that the engineer is enabled to focus on the model abstractions. Generated timing analyzers correctly represent the timing behavior of the particular target architecture because both the timing model and the synthesized circuits are based on the same source, the formal specification. By this, the timing model derivation approach prevents wrong analysis results caused by errors in the documentation of the analyzed hardware.

The effectiveness of the derivation approach depends on the coding style of the VHDL model. The results are excellent when the code features minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design. Ideally, the code reflects the structural composition of the processor pipeline with explicit control signals to steer the flow of instructions and data.

The presented method can be seen as a complementary approach to the development of timing models. Legacy solutions based on system documentation will most certainly not be entirely replaced by this method because not all hardware manufacturers are willing to provide their VHDL code to the model developers.

A disadvantage of the timing model derivation based on formal specifications is the requirement to fully understand the design which might be a complex task for a complete processor including peripheral devices. But the derivation approach has shown to be effective when applied to specific components of a complex system, e.g., a memory controller. Such components are better suitable in size and complexity and have a well defined interface to the remaining system.

Legacy timing model development processes and the proposed derivation approach share the fact that the invention of abstractions cannot be fully automated. Our proposed approach supports the extraction of timing-relevant information from a formal specification as well as the application of transformations on it. The idea of what particular kind of state abstraction is suitable for the analyzed hardware model remains an engineering task which is left to human experts.

12.2 Future Work

This section briefly summarizes potential future work that has been discovered during the work on this thesis.

Hardware circuit synthesis tools actually make use of code patterns to identify typical components like an adder which then are mapped to internally available highly optimized netlist implementations. The more patterns can be found in a VHDL model, the higher is the efficiency of the synthesized hardware. An interesting work could be to compare these code patterns with the coding style guidelines from Section 9.3. If a common denominator can

be found, this increases the probability that hardware architects will follow these guidelines.

As mentioned in Section 8.4, the timing model validation based on interval property checking only exists as a proof-of-concept. From a technical point of view, this work might be continued.

An analysis of the dependencies arising from the sensitivity lists of VHDL processes might be interesting from at least two points of view. On the one hand, such dependency information might be used together with module-specific dependencies to automatically determine the input order of the different code files to the VHDL compiler. On the other hand, proposals about potentially superfluous dependencies might be determined.

As described in Section 8.3, a complete property set for an architecture contains a lot of meta information about the timing behavior. It might be interesting to see whether this information can be used (at least as additional information) for the derivation of timing models for that architecture.

List of Figures

List of Tables

Listings

Bibliography

[Abs11] AbsInt Angewandte Informatik GmbH. aiSee. Website, January 2011. http://www.aisee.de/.

[Abs12] AbsInt Angewandte Informatik GmbH. Website, March 2012. http://www.absint.com.

[AM95] Alt, Martin and Martin, Florian. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *Proceedings of the International Static Analysis Symposium (SAS)*, volume 983 of *Lecture Notes in Computer Science (LNCS)*, pages 33–50 (Springer, Glasgow, United Kingdom, 1995). doi:10.1007/3-540-60360-3_31.

[AM97] Alt, Martin and Martin, Florian. Practical Comparison of Call String and Functional Approach in Data Flow Analysis. In Herbert Kuchen, editor, *Proceedings of the Arbeitstagung Programmiersprachen (ATPS)*, volume 58 of *Arbeitsberichte des Instituts für Wirtschaftsinformatik* (Westfälische Wilhelms-Universität, Munich, Germany, 1997).

[AMR10] Altmeyer, Sebastian, Maiza, Claire, and Reineke, Jan. Resilience Analysis: Tightening the CRPD Bound for Set-Associative Caches. In Jaejin Lee and Bruce R. Childers, editors, *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 153–162 (Association for Computing Machinery (ACM), Stockholm, Sweden, 2010). doi:10.1145/1755888.1755911.

[ARM00] ARM Limited. *ARM Architecture Reference Manual*, June 2000. Revision 1.

[Ash08] Ashenden, Peter J. *The designer's guide to VHDL*, volume 3 of *Systems on Silicon* (Morgan Kaufmann Publishers, San Francisco, USA, 2008), 3rd edition.

[Ave11] Averant Incorporated. Solidify Data Sheet. Website, July 2011. `http://www.averant.com/storage/documents/Solidify.pdf`.

[BBM+07] Bormann, Jörg, Beyer, Sven, Maggiore, Adriana, Siegel, Michael, Skalberg, Sebastian, Blackmore, Tim, and Bruno, Fabio. Complete Formal Verification of TriCore2 and Other Processors. In Tom Fitzpatrick, editor, *Proceedings of the Design and Verification Conference and Exhibition (DVCon)* (San José, USA, 2007).

[BCC+03] Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M., Strichman, Ofer, and Zhu, Yunshan. Bounded Model Checking. *Advances in Computers*, 58:117–148, August 2003. doi:10.1016/S0065-2458(03)58003-2.

[BCP02] Bernat, Guillem, Colin, Antoine, and Petters, Stefan M. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 279–288 (IEEE Computer Society, Austin, USA, 2002).

[BCRS10] Ballabriga, Clément, Cassé, Hugues, Rochange, Christine, and Sainrat, Pascal. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Proceedings of the Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46 (Springer, Waidhofen/Ybbs, Austria, 2010). doi:10.1007/978-3-642-16256-5_6.

[Ber06] Berg, Christoph. PLRU Cache Domino Effects. In Frank Mueller, editor, *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 4 of *OpenAccess Series in Informatics (OASICS)* (Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2006). doi:10.4230/OASIcs.WCET.2006.672.

[Bey05] Beyer, Sven. *Putting it all together - Formal Verification of the VAMP*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, March 2005.

[BLL+96] Bengtsson, Johan, Larsen, Kim, Larsson, Fredrik, Pettersson, Paul, and Yi, Wang. UPPAAL – a tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science: Hybrid Systems III*, 1066:232–243, April 1996.

[Bor09] Bormann, Jörg. *Vollständige funktionale Verifikation*. Ph.D. thesis, Technische Universität, Kaiserslautern, Germany, June 2009.

[CAN03] CAN in Automation (CiA). *ISO 11898-1:2003 Road vehicles — Controller area network — Part 1: Data link layer and physical signalling*, 2003.

[CC77] Cousot, Patrick and Cousot, Radhia. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 238–252 (Association for Computing Machinery (ACM), Los Angeles, USA, 1977). doi:10.1145/512950.512973.

[CC79] Cousot, Patrick and Cousot, Radhia. Systematic Design of Program Analysis Frameworks. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 269–282 (Association for Computing Machinery (ACM), San Antonio, USA, 1979). doi:10.1145/567752.567778.

[CC81] Cousot, Patrick and Cousot, Radhia. *Program flow analysis: Theory and applications*, chapter 10, pages 303–342 (Prentice Hall, Englewood Cliffs, USA, 1981).

[CC92a] Cousot, Patrick and Cousot, Radhia. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992. doi:10.1093/logcom/2.4.511.

[CC92b] Cousot, Patrick and Cousot, Radhia. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the International Workshop of Programming Language Implementation and Logic Programming (PLILP)*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295 (Springer, Leuven, Belgium, 1992). doi:10.1007/3-540-55844-6_101.

[CFG+10] Cullmann, Christoph, Ferdinand, Christian, Gebhard, Gernot, Grund, Daniel, Maiza, Claire, Reineke, Jan, Triquet, Benoît, Wegener, Simon, and Wilhelm, Reinhard. Predictability Considerations in the Design of Multi-Core Embedded Systems. In *Proceedings of the International Congress and Exhibition on Embedded Real Time Software and Systems (ERTS^2)*, pages 36–42 (Toulouse, France, 2010).

[CG11] Courbin, Pierre and George, Laurent. FORTAS: Framework for Real-Time Analysis and Simulation. In Giuseppe Lipari and Tommaso Cucinotta, editors, *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 21–26 (Porto, Portugal, 2011).

[CGJ+00] Clarke, Edmund, Grumberg, Orna, Jha, Somesh, Lu, Yuan, and Veith, Helmut. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 154–169 (Springer, Chicago, USA, 2000). doi:10.1007/10722167_15.

[CH78] Cousot, Patrick and Halbwachs, Nicolas. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 84–96 (Association for Computing Machinery (ACM), Tucson, USA, 1978). doi:10.1145/512760.512770.

[Cla07] Claessen, Koen. A Coverage Analysis for Safety Property Lists. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 139–145 (IEEE Computer Society, Austin, USA, 2007).

[CLRS01] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms* (The MIT press, Cambridge, USA, 2001), 2nd edition. ISBN 0262032937.

[CMH+03] Calazans, Ney, Moreno, Edson, Hessel, Fabiano, Rosa, Vitor, Moraes, Fernando, and Carara, Everton. From VHDL Register Transfer Level to SystemC Transaction Level Modeling: a Comparative Case Study. In *Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 355–360 (IEEE Computer Society, Sao Paulo, Brazil, 2003). doi: 10.1109/SBCCI.2003.1232853.

[Dae] Daedalus Research Project. Website. `http://www.di.ens.fr/~cousot/projects/DAEDALUS/index.shtml`.

[DO92] Radio Technical Commission for Aeronautics SC-167. *Software Considerations in Airborne Systems and Equipment Certification (DO-178B)*, December 1992.

[DOT+10] Dalsgaard, Andreas E., Olesen, Mads Chr., Toft, Martin, Hansen, René R., and Larsen, Kim G. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15 of *OpenAccess Series in Informatics (OASICS)*, pages 113–123 (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2010). doi: 10.4230/OASIcs.WCET.2010.113.

[EB09] Evstyugov-Babaev, Alexander. Graph Description Language in a Nutshell. Website, July 2009. `http://www.aisee.de/gdl/nutshell/`.

[EPB+06] Eisinger, Jochen, Polian, Ilia, Becker, Bernd, Metzner, Alexander, Thesing, Stephan, and Wilhelm, Reinhard. Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis. In Matteo Sonza Reorda, Ondrej Novák, Bernd Straube, Hanna Kubátová, Zdenek Kotásek, Pavel Kubalík, Raimund Ubar, and Jiri Bucek, editors, *Proceedings of the Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20 (IEEE Computer Society, Prague, Czech Republic, 2006).

[Erm03] Ermedahl, Andreas. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Ph.D. thesis, Uppsala University, Uppsala, Sweden, June 2003.

[ESG+07] Ermedahl, Andreas, Sandberg, Christer, Gustafsson, Jan, Bygde, Stefan, and Lisper, Björn. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In Christine Rochange, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, OpenAccess Series in Informatics (OASICS (Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2007). doi:10.4230/OASIcs.WCET.2007.1194.

[Fer97] Ferdinand, Christian. *Cache Behaviour Prediction for Real-Time Systems*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 1997.

[FH08] Ferdinand, Christian and Heckmann, Reinhold. Worst-Case Execution Time–A Tool Provider's Perspective. In *Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 340–345 (IEEE Computer Society, Orlando, USA, 2008). doi:10.1109/ISORC.2008.16.

[FHL+01] Ferdinand, Christian, Heckmann, Reinhold, Langenbach, Marc, Martin, Florian, Schmidt, Michael, Theiling, Henrik, Thesing, Stephan, and Wilhelm, Reinhard. Reliable and Precise WCET Determination for a Real-Life Processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Proceedings of the International Conference on Embedded Software (EMSOFT)*, volume 2211 of *Lecture Notes in Computer Science: Embedded Software*, pages 469–485 (Springer, Tahoe City, USA, 2001). doi:10.1007/3-540-45449-7_32.

[Fle10] FlexRay Consortium. *FlexRay Protocol Specification*, October 2010. Version 3.0.1.

[FMC+07] Ferdinand, Christian, Martin, Florian, Cullmann, Christoph, Schlickling, Marc, Stein, Ingmar, Thesing, Stephan, and Heckmann, Reinhold. New Developments in WCET Analysis. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*,

volume 4444 of *Lecture Notes in Computer Science: Program Analysis and Compilation, Theory and Practice*, pages 12–52 (Springer, Berlin, Germany, 2007). doi:10.1007/978-3-540-71322-7_1.

[FMWA99] Ferdinand, Christian, Martin, Florian, Wilhelm, Reinhard, and Alt, Martin. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 35:163–189, November 1999. doi:10.1016/S0167-6423(99)00010-6.

[Fre01] Freescale Semiconductor Incorporated. *MPC750 RISC Microprocessor Family User's Manual*, December 2001. Revision 1.

[Fre02] Freescale Semiconductor Incorporated. *MPC603e RISC Microprocessor User's Manual*, 2002. Revision 3.

[Fre04] Freescale Semiconductor Incorporated. *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*, January 2004. Revision 0.1.

[Fre05a] Freescale Semiconductor Incorporated. *MPC7450 RISC Microprocessor Family Reference Manual*, January 2005. Rev. 5.

[Fre05b] Freescale Semiconductor Incorporated. *sim_G4plus v1.1 Cycle-Accurate Simulator User's Guide*, May 2005. Rev 2.4.

[Fre11] Free Software Foundation. Bison Manual. Website, May 2011. `http://www.gnu.org/software/bison/manual/bison.html`.

[FW98] Ferdinand, Christian and Wilhelm, Reinhard. On Predicting Data Cache Behavior for Real-Time Systems. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes In Computer Science: Languages, Compilers, And Tools For Embedded Systems*, pages 16–30 (Springer, Montréal, Canada, 1998). doi:10.1007/BFb0057777.

[FW99] Ferdinand, Christian and Wilhelm, Reinhard. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2-3):131–181, November 1999. doi:10.1023/A:1008186323068.

[Gai05] Gaisler Research. *LEON2 Processor User's Manual*, July 2005. Version 1.0.30.

[GCH11] Gebhard, Gernot, Cullmann, Christoph, and Heckmann, Reinhold. Software Structure and WCET Predictability. In Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Proceedings of the International Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, volume 18 of *OpenAccess Series in Informatics (OASICS)*, pages 1–10 (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2011). doi:10.4230/OASIcs.PPES.2011.1.

[Geb10] Gebhard, Gernot. Timing Anomalies Reloaded. In Björn Lisper, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15 of *OpenAccess Series in Informatics (OASICS)*, pages 1–10 (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2010). doi:10.4230/OASIcs.WCET.2010.1.

[GHC05] Gaisler, Jiri, Habinc, Sandi, and Catovic, Edvin. *GRLIB IP Library User's Manual*, 2005. Version 1.0.6.

[Gin] Gingold, Tristan. GHDL User's Guide. Website. `http://ghdl.free.fr/`.

[GK83] Gajski, Daniel and Kuhn, Robert H. New VLSI Tools. *IEEE Computer*, 16(12):11–14, December 1983. doi:10.1109/MC.1983.1654264.

[GR09] Grund, Daniel and Reineke, Jan. Abstract Interpretation of FIFO Replacement. In Jens Palsberg and Zhendong Su, editors, *Proceedings of the International Static Analysis Symposium (SAS)*, volume 5673 of *Lecture Notes In Computer Science: Static Analysis*, pages 120–136 (Springer, Berlin, Germany, 2009). doi:10.1007/978-3-642-03237-0_10.

[GR10] Grund, Daniel and Reineke, Jan. Toward Precise PLRU Cache Analysis. In Björn Lisper, editor, *Proceedings of International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15 of *OpenAccess Series in Informatics (OASICS)*, pages 23–35 (Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2010). doi:10.4230/OASIcs.WCET.2010.23.

[Gre08] Grewe, Dominik. *Static Congruence Analysis on Binaries*. Bachelor thesis, Saarland University, Saarbrücken, Germany, July 2008.

[GRG09] Grund, Daniel, Reineke, Jan, and Gebhard, Gernot. Branch Target Buffers: WCET Analysis Framework and Timing Predictability. In Patrick Kellenberger, editor, *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 3–12 (IEEE Computer Society, Beijing, China, 2009). doi:10.1109/RTCSA.2009.8.

[GRW11] Grund, Daniel, Reineke, Jan, and Wilhelm, Reinhard. A Template for Predictability Definitions with Supporting Evidence. In Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Proceedings of the International Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, volume 18 of *OpenAccess Series in Informatics (OASICS)*, pages 22–31 (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2011). doi:10.4230/OASIcs.PPES.2011.22.

[Hec10] Heckmann, Reinhold. *AbsInt Advanced Analyzer for PowerPC MPC7448*. AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, December 2010.

[Hei00] Heinkel, Ulrich. *The VHDL Reference – A Practical Guide to Computer-Aided Integrated Circuit Design* (Wiley-Blackwell, 2000).

[Her12] Herkules Research Project. Website, March 2012. `http://www.edacentrum.de/herkules`.

[HHL+11] von Hanxleden, Reinhard, Holsti, Niklas, Lisper, Björn, Ploedereder, Erhard, Wilhelm, Reinhard, Bonenfant, Armelle, Casse, Hugues, Bünte, Sven, Fellger, Wolfgang, Gepperth, Sebastian, Gustafsson, Jan, Huber, Benedikt, Islam, Nazrul Mohammad, Kästner, Daniel, Kirner, Raimund, Kovacs, Laura, Krause, Felix, de Michiel, Marianne, Olesen, Mads Christian, Prantl, Adrian, Puffitsch, Wolfgang, Rochange, Christine, Schoeberl, Martin, Wegener, Simon, Zolda, Michael, and Zwirchmayr, Jakob. WCET Tool Challenge 2011: Report. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)* (Austrian Computer Society, Porto, Portugal, 2011).

[Hor97] Horch, Joachim. Entwurf eines RISC–Prozessors in der Hardwarebeschreibungssprache VHDL. Studienarbeit, Technische Universität, Darmstadt, Germany, June 1997.

[HPG06] Hennessy, John L., Patterson, David A., and Goldberg, David. *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann Publishers, San Francisco, USA, 2006), 4th edition.

[HR09] Herter, Jörg and Reineke, Jan. Making Dynamic Memory Allocation Static To Support WCET Analyses. In Niklas Holsti, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 10 of *OpenAccess Series in Informatics (OASICS)* (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dublin, Ireland, 2009).

[HS02] Holsti, Niklas and Saarinen, Sami. Status of the Bound-T WCET tool. In Guillem Bernat, editor, *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 36–41 (Vienna, Austria, 2002).

[Hym02] Hymans, Charles. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. *Lecture Notes in Computer Science: Static Analysis*, 2477:493–498, 2002. doi:10.1007/3-540-45789-5_31.

[Hym04] Hymans, Charles. *Verification of VHDL descriptions by abstract interpretation*. Ph.D. thesis, École Polytechnique, Paris, France, September 2004.

[IBM06] International Business Machines Corporation. *IBM PowerPC 750GX and 750GL RISC Microprocessor Users Manual*, March 2006. Version 1.2.

[IEC10] International Electrotechnical Commission. *IEC 61508–Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*, April 2010. Safety Standard.

[IEE87] IEEE Standards Association. *1076-1987 IEEE Standard VHDL Language Reference Manual*, January 1987. doi:10.1109/IEEESTD.1992.101084.

[IEE93] IEEE Standards Association. *1164-1993 IEEE Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*, 1993.

[IEE95a] IEEE Standards Association. *1076.4-1995 IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, 1995. doi:10.1109/IEEESTD.1996.80811.

[IEE95b] IEEE Standards Association. *1364-1995 IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 1995.

[IEE96] IEEE Standards Association. *1076.2-1996 IEEE Standard VHDL Language Math Packages*, 1996. doi:10.1109/IEEESTD.1997.81589.

[IEE97] IEEE Standards Association. *1076.3-1997 IEEE Standard VHDL Synthesis Packages*, 1997. doi:10.1109/IEEESTD.1997.82399.

[IEE99a] IEEE Standards Association. *1076.1-1999 IEEE Standard VHDL Analog and Mixed-Signal Extensions*, 1999. doi:10.1109/IEEESTD.1999.90578.

[IEE99b] IEEE Standards Association. *1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, 1999. doi:10.1109/IEEESTD.2004.94802.

[IEE00] IEEE Industry Standards and Technology Organization – NEXUS 5001™ Forum. *IEEE-ISTO 5001™-1999*, January 2000. `http://www.nexus5001.org/`, version 1.1.

[IEE05] IEEE Standards Association. *1666-2005 IEEE Standard SystemC Language Reference Manual*, 2005. doi:10.1109/IEEESTD.2006.99475.

[Inf] Infineon Technologies AG. Website. `http://www.infineon.com`.

[Inf07] Infineon Technologies AG, Munich, Germany. *TC1797 32-Bit Single-Chip Microcontroller Target Specification*, November 2007. Version 1.5.

[Ins10] Insight Research Corporation. The 2010 Telecommunications Industry Review: An Anthology of Market Facts and Forecasts. Website, January 2010. `http://www.insight-corp.com/reports/review10.asp`.

[INT] INTERESTED Research Project. Website. `http://interested-ip.eu`.

[Int11] International Business Machines Corporation. ILOG CPLEX Optimizer. Website, January 2011. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer`.

[ISO11] International Organization for Standardization. *ISO 26262 Road vehicles – Functional safety*, April 2011. Safety Standard.

[Jas11] Jasper Design Automation. JasperGold. Website, August 2011. `http://www.jasper-da.com/products/jaspergold.htm`.

[KK67] Kreisel, Geord and Krivine, Jean L. *Elements of Mathematical Logic: (Model Theory)*, volume 42 of *Studies in Logic and the Foundations of Mathematics* (North Holland Publishing Company, 1967).

[Kop97] Kopetz, Hermann. *Real-Time Systems: Design Principles for Distributed Embedded Applications* (Kluwer Academic Publishers, Norwell, USA, 1997), 1st edition.

[KWH+08] Kästner, Daniel, Wilhelm, Reinhard, Heckmann, Reinhold, Schlickling, Marc, Pister, Markus, Jersak, Marek, Richter, Kai, and Ferdinand, Christian. Timing Validation of Automotive Software. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, volume 17 of *Communications In Computer and Information Science: Leveraging Applications Of Formal Methods, Verification And Validation*, pages 93–107 (Springer, Berlin, Germany, 2008). doi:10.1007/978-3-540-88479-8_8.

[KWN+10] Kästner, Daniel, Wilhelm, Stephan, Nenova, Stefana, Cousot, Patrick, Cousot, Radia, Feret, Jérôme, Mauborgne, Laurent, Miné, Antoine, and Rival, Xavier. Astrée: Proving the Absence of Runtime Errors. In Jean-Claude Laprie, editor, *Proceedings of the International Congress and Exhibition of Embedded Real Time Software and Systems (ERTS^2)* (Toulouse, France, 2010).

[Kä00] Kästner, Daniel. *Retargetable Postpass Optimisation by Integer Linear Programming*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, December 2000.

[Lee09] Lee, Edward A. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009. doi:10.1145/1506409.1506426.

[Lei08] Leinenbach, Dirk. *Compiler Verification in the Context of Pervasive System Verification*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, June 2008.

[LH03] Lougee-Heimer, Robin. The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, January 2003. doi:10.1147/rd.471.0057.

[Lin] Linux Kernel Archives. Website. `http://www.kernel.org`.

[Liu00] Liu, Jane W.S. *Real-Time Systems* (Prentice Hall, Upper Saddle River, USA, 2000), 1st edition.

[LL73] Liu, Jane W.S. and Layland, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.

[LM95] Li, Yau-Tsun Steven and Malik, Sharad. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS)*, volume 30 of *ACM SIGPLAN Notices*, pages 88–98 (Association for Computing Machinery (ACM), La Jolla, USA, 1995). doi:10.1145/216636.216666.

[LWS+10] Loitz, Sascha, Wedler, Markus, Stoffel, Dominik, Brehm, Christian, and Kunz, Wolfgang. Complete Verification of Weakly Programmable IPs against Their Operational ISA Model. In Adam Morawiec and Jinnie Hinderscheit, editors, *Proceedings of the Forum on Specification & Design Languages (FDL)*, pages 29–36 (Electronic Chips & Systems design Initiative (ECSI), Southampton, United Kingdom, 2010). doi:10.1049/ic.2010.0125.

[Mak07] Maksoud, Mohamed Abdel. *Generating Code from Abstract VHDL Models*. Master's thesis, Saarland University, Saarbrücken, Germany, August 2007.

[Mar98] Martin, Florian. PAG–an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):1–22, November 1998. doi:10.1007/s100090050017.

[Mar99] Martin, Florian. *Generating Program Analyzers*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, June 1999.

[Mar05] Marwedel, Peter. *Embedded System Design* (Springer, Berlin, Germany, 2005), 2nd edition.

[Mat06] Matthies, Niklas. *Präzise Bestimmung längster Programmpfade anhand von Zustandsgraphen unter Berücksichtigung von Schleifen-Nebenbedingungen*. Diploma thesis, Saarland University, Saarbrücken, Germany, February 2006.

[MAWF98] Martin, Florian, Alt, Martin, Wilhelm, Reinhard, and Ferdinand, Christian. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC)*, volume 1383 of *Lecture Notes in Computer Science: Compiler Construction*, pages 80–94 (Springer, Lisboa, Portugal, 1998). doi:10.1007/BFb0026424.

[Men08] Mentor Graphics Corporation. ModelSim DataSheet. Website, 2008. `http://www.mentor.com/products/fv/modelsim/upload/datasheet.pdf`.

[Mic] Microsoft Windows Embedded Compact 7. Website. `http://www.microsoft.com/windowsembedded`.

[Moo65] Moore, Gordon. Cramming More Components onto Integrated Circuits. *Electronic Magazine*, 38(8):114–117, April 1965. doi:10.1109/JPROC.1998.658762.

[Mot92] Motorola Incorporated. *M68020 Microprocessors User's Manual*, 1st edition, 1992. Revision 2.

[MP00] Müller, Silvia M. and Paul, Wolfgang J. *Computer Architecture: Complexity and Correctness* (Springer, Berlin, Germany, 2000).

[MPS09] Maksoud, Mohamed Abdel, Pister, Markus, and Schlickling, Marc. An Abstraction-Aware Compiler for VHDL Models. In *Proceedings of the International Conference on Computer Engineering and Systems (ICCES)*, pages 3–9 (IEEE Computer Society, Cairo, Egypt, 2009). doi:10.1109/ICCES.2009.5383321.

[MV99] Mahapatra, Nihar R. and Venkatrao, Balakrishna. The Processor-Memory Bottleneck: Problems and Solutions. *Crossroads – Computer Architecture*, 5, April 1999. doi:10.1145/357783.331677.

[MWV+04] Marwedel, Peter, Wehmeyer, Lars, Verma, Manish, Steinke, Stefan, and Helmig, Urs. Fast, predictable and low energy memory references through architecture-aware compilation. In Masaharu Imai, editor, *Proceedings of the International Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair (ASP-DAC)*, pages 4–11 (IEEE Computer Society, Yokohama, Japan, 2004).

[Neh04] Nehme, Carl. *The VAT Tool: Automatic Transformation of VHDL to Timed Automata*. Master's thesis, Massachusetts Institute of Technology, Cambridge, USA, June 2004.

[NN92] Nielson, Hanne Riis and Nielson, Flemming. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing (John Wiley & Sons Limited, 1992).

[NNH99] Nielson, Flemming, Nielson, Hanne Riis, and Hankin, Chris. *Principles of Program Analysis* (Springer, Berlin, Germany, 1999), 1st edition.

[One11] OneSpin Solutions. *OneSpin 360 MV Product Familiy Data Sheet*, July 2011.

[PCI98] PCI Special Interest Group (PCI-SIG). *PCI Local Bus Specification*, December 1998. Revision 2.2.

[PK05] Pister, Markus and Kästner, Daniel. Generic Software Pipelining at the Assembly Level. In Krishna M. Kavi and Ron Cytron, editors, *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 50–62 (Association for Computing Machinery (ACM), Dallas, USA, 2005). doi:10.1145/1140389.1140395.

[PN98] Puschner, Peter and Nossal, Roman. Testing the results of static worst-case execution-time analysis. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 134–143 (IEEE Computer Society, Madrid, Spain, 1998). doi:10.1109/REAL.1998.739738.

[PRE] PREDATOR Research Project. Website. `http://www.predator-project.eu/`.

[PSK08] Prantl, Adrian, Schordan, Markus, and Knoop, Jens. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In Raimund Kirner, editor, *Proceedings of the International*

Workshop on Worst-Case Execution Time Analysis (WCET) (Prague, Czech Republic, 2008). doi:10.4230/OASIcs.WCET.2008.1661.

[PSM09] Pister, Markus, Schlickling, Marc, and Maksoud, Mohamed Abdel. Semi-Automatic Derivation of Abstract Processor Models. Reports of ES_PASS, Saarland University, Saarbrücken, Germany, June 2009.

[QNX] QNX. QNX Realtime Operating System. Website. `http://www.qnx.com/products/neutrino-rtos/index.html`.

[Rea11] RealIntent Incorporated, Sunnyvale, USA. *CONQUEST Data Sheet*, July 2011.

[Rei08] Reineke, Jan. *Caches In WCET Analysis – Predictability, Competitiveness, Sensitivity*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, November 2008.

[RGBW07] Reineke, Jan, Grund, Daniel, Berg, Christoph, and Wilhelm, Reinhard. Timing Predictability of Cache Replacement Policies. *Real-Time Systems*, 37(2):99–122, November 2007. doi:10.1007/s11241-007-9032-3.

[RS09] Reineke, Jan and Sen, Rathijit. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In Niklas Holsti, editor, *Proceedings of International Workshop on Worst-Case Execution Time Analysis (WCET)*, OpenAccess Series in Informatics (OASICS) (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2009).

[RWT+06] Reineke, Jan, Wachter, Björn, Thesing, Stephan, Wilhelm, Reinhard, Polian, Ilia, Eisinger, Jochen, and Becker, Bernd. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *Proceedings of International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 4 of *OpenAccess Series in Informatics (OASICS)* (Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2006). doi:10.4230/OASIcs.WCET.2006.671.

[Sch03] Schneider, Jörn. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2003.

[Sch05] Schlickling, Marc. Trace Validation for aiT for PowerPC MPC755 (Hurricane Chip Set). Technical report, AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, March 2005.

[Sch10] Schlickling, Marc. ANASTASY WP1-Report: Searching for Timing Anomalies. Technical report, AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, November 2010.

[Sch13] Schlickling, Marc. *Timing Model Derivation – Static Analysis of Hardware Description Languages*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2013. To appear.

[SP07] Schlickling, Marc and Pister, Markus. A Framework for Static Analysis of VHDL Code. In Christine Rochange, editor, *Proceedings of the International Workshop on Worst-case Execution Time Analysis (WCET)*, volume 6 of *OpenAccess Series in Informatics (OASICS)*, pages 29–34 (Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2007). doi:10.4230/OASIcs.WCET.2007.1189.

[SP09] Schlickling, Marc and Pister, Markus. Worst Case Execution Time Analyzer for PowerPC MPC7448 Performance Counter Validation Report. Technical report, AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, September 2009.

[SP10] Schlickling, Marc and Pister, Markus. Semi-Automatic Derivation of Timing Models for WCET Analysis. In Jaejin Lee and Bruce R. Childers, editors, *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 67–76 (Association for Computing Machinery (ACM), Stockholm, Sweden, 2010). doi:10.1145/1755888.1755899.

[SPH+05] Souyris, Jean, Pavec, Ervan Le, Himbert, Guillaume, Jégu, Victor, Borios, Guillaume, and Heckmann, Reinhold. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In Reinhard Wilhelm, editor, *Proceedings of the International Workshop on Worst-case Execution Time (WCET)*, pages 21–24 (Mallorca, Spain, 2005). doi:10.4230/OASIcs.WCET.2005.810.

[SPPH10] Schoeberl, Martin, Puffitsch, Wolfgang, Pedersen, Rasmus Ulslev, and Huber, Benedikt. Worst-case execution time analysis for a

Java processor. *Software: Practice and Experience*, 40(6):507–542, May 2010. doi:10.1002/spe.968.

[Ste10] Stein, Ingmar. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, February 2010.

[Syn11] Synopsys Incorporated, Mountain View, USA. *VCS Data Sheet*, July 2011.

[Syn12] Synopsys Incorporated. OpenVera. Website, March 2012. `http://www.open-vera.com/`.

[TFW00] Theiling, Henrik, Ferdinand, Christian, and Wilhelm, Reinhard. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2–3):157–159, 2000. doi:10.1023/A:1008141130870.

[The02] Theiling, Henrik. ILP-based Interprocedural Path Analysis. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Proceedings of the International Conference on Embedded Software (EMSOFT)*, volume 2491 of *Lecture Notes in Computer Science: Embedded Software*, pages 349–363 (Springer, Berlin, Germany, 2002). doi:10.1007/3-540-45828-X_26.

[The03] Theiling, Henrik. *Control Flow Graphs for Real-Time System Analysis: Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, February 2003.

[The04] Thesing, Stephan. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. Ph.D. thesis, Saarland University, Saarbrücken, Germany, July 2004.

[The06] Thesing, Stephan. Modeling a system controller for timing analysis. In Sang Lyul Min and Wang Yi, editors, *Proceedings of the International Conference Embedded Software (EMSOFT)*, pages 292–300 (Association for Computing Machinery (ACM), Seoul, Korea, 2006). doi:10.1145/1176887.1176929.

[TN06] Tolstrup, Terkel K. and Nielson, Flemming. Analyzing for Absence of Timing Leaks in VHDL. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS)*, pages 71–86 (Vienna, Austria, 2006).

[TSH+03] Thesing, Stephan, Souyris, Jean, Heckmann, Reinhold, Randimbivololona, Famantanantsoa, Langenbach, Marc, Wilhelm, Reinhard, and Ferdinand, Christian. An Abstract Interpretation–Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 625–632 (IEEE Computer Society, San Francisco, USA, 2003). doi:10.1109/DSN.2003.1209972.

[Tur09] Turley, Jim. Growth Is No Longer in PCs, It's In Embedded Systems. Website, June 2009. `http://www.glgroup.com/News/Growth-Is-No-Longer-in-PCs-Its-In-Embedded-Systems-40088.html`.

[USB+10] Urdahl, Joakim, Stoffel, Dominik, Bormann, Jörg, Wedler, Markus, and Kunz, Wolfgang. Path predicate abstraction by complete interval property checking. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 207–215 (IEEE Computer Society, Lugano, Switzerland, 2010).

[Veg82] Vegdahl, Steven R. Phase coupling and constant generation in an optimizing microcode compiler. *SIGMICRO Newsletter*, 13(4):125–133, October 1982.

[Ver] Verisoft XT Research Project. Website. `http://www.verisoftxt.de`.

[Wae06] van de Waerdt, Jan-Willem. *The TM3270 Media-processor*. Ph.D. thesis, Delft University of Technology, Delft, Netherlands, October 2006.

[WEE+08] Wilhelm, Reinhard, Engblom, Jakob, Ermedahl, Andreas, Holsti, Niklas, Thesing, Stephan, Whalley, David, Bernat, Guillem, Ferdinand, Christian, Heckmann, Reinhold, Mitra, Tulika, Mueller, Frank, Puaut, Isabelle, Puschner, Peter, Staschulat, Jan, and Stenström, Per. The Worst-Case Execution Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, April 2008. doi:10.1145/1347375.1347389.

[Weg11] Wegener, Simon. *Improving Static Analysis of Loops*. Master's thesis, Saarland University, Saarbrücken, Germany, June 2011.

[Wei95] Weiser, Mark. The Computer for the 21st Century. In Ronald Baecker, Jonathan Grudin, William A. Buxton, and Saul Greenberg, editors, *Readings in Human-Computer Interaction: Toward the Year 2000*, pages 933–940 (Morgan Kaufmann Publishers, San Francisco, USA, 1995), 2nd edition.

[WFC+09] Wilhelm, Reinhard, Ferdinand, Christian, Cullmann, Christoph, Grund, Daniel, Reineke, Jan, and Triquet, Benoît. Designing Predictable Multicore Architectures for Avionics and Automotive Systems. In Lothar Thiele, Reinhard Wilhelm, Theo Ungerer, Bengt Jonsson, and Jian-Jia Chen, editors, *Proceedings of the Workshop on Reconciling Performance with Predictability (RePP)* (Grenoble, France, 2009).

[WGR+09] Wilhelm, Reinhard, Grund, Daniel, Reineke, Jan, Schlickling, Marc, Pister, Markus, and Ferdinand, Christian. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009. doi: 10.1109/TCAD.2009.2013287.

[Win] Wind River Systems. VxWorks. Website. `http://www.windriver.com/products/vxworks/`.

[WM95] Wilhelm, Reinhard and Maurer, Dieter. *Compiler Design*. International Computer Science Series (Addison-Wesley, 1995).

[Wol00] Wolf, Wayne. *Computers as Components: Principles of Embedded Computing Systems Design* (Morgan Kaufmann Publishers, San Francisco, USA, 2000), 1st edition.

[WRKP05] Wenzel, Ingomar, Rieder, Bernhard, Kirner, Raimund, and Puschner, Peter. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proceedings of the International Conference and Exhibition on Design, Automation and Test in Europe (DATE)*, pages 606–611 (IEEE Computer Society, Munich, Germany, 2005). doi:10.1109/DATE.2005.76.

[WW09] Wilhelm, Stephan and Wachter, Björn. Symbolic State Traversal for WCET Analysis. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 137–146 (Association for

Computing Machinery (ACM), Grenoble, France, 2009). doi: 10.1145/1629335.1629354.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W