UNIVERSITÄT DES SAARLANDES
Naturwissenschaftlich-Technische Fakultät I

Fachbereich 6.2 Informatik
Lehrstuhl für Programmiersprachen und Übersetzerbau

Timing Model Derivation

Static Analysis of Hardware Description Languages

Dissertation

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

von

Diplom-Informatiker
Marc Schlickling

Saarbrücken
2012

Kolloquium

Tag des Kolloquiums	Montag, 17. Dezember 2012
Dekan	Prof. Dr. Mark Groves
Prüfungsausschuss	
Vorsitzender	Prof. Dr. Jan Reineke
Berichterstatter	Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm Prof. Dr.-Ing. Wolfgang Kunz Dr. Susanne Graf
Akad. Mitarbeiter	Dr. Mario Albrecht

Impressum

Druck und Verlag: epubli GmbH, Berlin, www.epubli.de

Printed in Germany

ISBN: 978-3-8442-4513-4

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 hard real-time systems are subject to strict timing constraints. In order to derive guarantees on the timing behavior, the worst-case execution time (WCET) of each task comprising the system has to be known. The *aiT* tool has been developed for computing safe upper bounds on the WCET of a task. Its computation is mainly based on abstract interpretation of timing models of the processor and its periphery. These models are currently hand-crafted by human experts, which is a time-consuming and error-prone process.
Modern processors are automatically synthesized from formal hardware specifications. Besides the processor's functional behavior, also timing aspects are included in these descriptions. A methodology to derive sound timing models using hardware specifications is described within this thesis. To ease the process of timing model derivation, the methodology is embedded into a sound framework.
A key part of this framework are static analyses on hardware specifications. This thesis presents an analysis framework that is build on the theory of abstract interpretation allowing use of classical program analyses on hardware description languages. Its suitability to automate parts of the derivation methodology is shown by different analyses. Practical experiments demonstrate the applicability of the approach to derive timing models. Also the soundness of the analyses and the analyses' results is proved.

Zusammenfassung

Sicherheitskritische Echtzeitsysteme unterliegen strikten Zeitanforderungen. Um ihr Zeitverhalten zu garantieren müssen die Ausführungszeiten der einzelnen Programme, die das System bilden, bekannt sein. Um sichere obere Schranken für die Ausführungszeit von Programmen zu berechnen wurde *aiT* entwickelt. Die Berechnung basiert auf abstrakter Interpretation von Zeitmodellen des Prozessors und seiner Peripherie. Diese Modelle werden händisch in einem zeitaufwendigen und fehleranfälligen Prozess von Experten entwickelt.
Moderne Prozessoren werden automatisch aus formalen Spezifikationen erzeugt. Neben dem funktionalen Verhalten beschreiben diese auch das Zeitverhalten des Prozessors. In dieser Arbeit wird eine Methodik zur sicheren Ableitung von Zeitmodellen aus der Hardwarespezifikation beschrieben. Um den Ableitungsprozess zu vereinfachen ist diese Methodik in eine automatisierte Umgebung eingebettet.
Ein Hauptbestandteil dieses Systems sind statische Analysen auf Hardwarebeschreibungen. Diese Arbeit stellt eine Analyse-Umgebung vor, die auf der Theorie der abstrakten Interpretation aufbaut und den Einsatz von klassischen Programmanalysen auf Hardwarebeschreibungssprachen erlaubt. Die Eignung des Systems, Teile der Ableitungsmethodik zu automatisieren, wird anhand einiger Analysen gezeigt. Experimentelle Ergebnisse zeigen die Anwendbarkeit der Methodik zur Ableitung von Zeitmodellen. Die Korrektheit der Analysen und der Analyse-Ergebnisse wird ebenfalls bewiesen.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, *Prof. Reinhard Wilhelm*, for his continuous support of my research, for his scientific advice, patience, motivation, enthusiasm, and knowledge. He left me a lot of freedom and created a pleasant and unique working atmosphere in his group.
Special thanks to the other members of my committee, *Prof. Wolfgang Kunz*, *Dr. Susanne Graf*, *Prof. Jan Reineke*, and *Dr. Mario Albrecht*, for their time, interest and insightful questions; I owe them my heartfelt appreciation.
I shared my office at university as well as in industry with my colleague and friend, *Dr. Markus Pister*; he was my crony in the AVACS subproject R2. I owe him my most sincere gratitude for sumless discussions, lending me a sympathetic ear when my thoughts got entangled, his technical expertise, and hilarious lunch breaks.
I will forever be thankful to my former colleague, *Dr. Stephan Thesing*. He has been helpful in providing advice many times during my research and he also provides invaluable feedback while reading this thesis.
My colleagues at *AbsInt Angewandte Informatik GmbH* and the *Compiler Design Group* at Saarland University also deserve my sincerest thanks; their friendship and assistance has meant more to me than I could ever express.
I would like to thank *Dr. Philipp Lucas* for being supportive throughout my time at the chair and for helping me with reviewing drafts of this thesis. *Dr. Reinhold Heckmann* deserves my gratitude for proofreading large parts of my thesis; his suggestions on mathematical corner cases are of inestimable value.
My family has always supported me. I am grateful to my parents, *Elisabeth* and *Heinz*, for their love and support, and waiting patiently for me to finish this thesis. Thanks also to my parents-in-law, *Margot* and *Rolf*, for standing by me and supporting me all the time. I thank my son, *Lenn Mattis*, for all the joy and happiness, he brings to my life. Finally, my thanks to my wife, *Bettina* – this is not new, but absolutely necessary.

– To Lenn for changing my life –

This work has been supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), and by the BMBF as part of the "Verisoft^{XT}" project.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

> One never knows what will happen if things are suddenly changed. On the other hand, however, does one know what will happen if they are not changed?
>
> *(Elias Canetti)*

Embedded systems are computer systems that are designed for specific control functions within larger systems. Usually, they are embedded as part of a complete device often including mechanical parts. The complexity of such systems varies from low employing a single micro-controller chip to high with multiple units and peripherals that are interconnected. However, the key characteristic of embedded systems is that they are dedicated to handle a particular task. By contrast, general-purpose computers such as personal computers are designed for flexibility in order to meet a wide range of user needs.

Embedded systems can be found in various devices in common use today. They range from portable devices such as mobile phones and MP3 players, or safety systems like the anti-lock braking system to large stationary installations like traffic lights and factory controllers. Use of embedded systems within safety critical areas as part of the occupant restraint system (e.g., airbags), or their usage within the control of nuclear power plants induce real-time computing constraints which are dictated by the surrounding physical environment.

A system is said to be a *real-time system* if the correctness of its operation depends not only upon its logical correctness, but also upon the time in which it is performed. Depending on the consequences of missing a *deadline*, real-time

systems can be classified into hard and soft real-time systems. A system is called a *hard real-time system*, if missing a deadline may cause a severe problem. A classical example of a hard real-time system is the airbag used within cars whose time for detecting a crash, deciding which airbag to fire, and inflating the envelope is limited to a few milliseconds. By contrast, a system is called a *soft real-time system*, if missing a deadline only results in a degrade of the system's quality of service. DVD players are a typical example of a soft real-time system: decoding of frames is subject to timing constraints, but the violation of constraints only results in a degraded quality while the player can still continue to operate.

Thus, one crucial issue in the design and development of a hard real-time system is to ensure that all deadlines are met. A schedulability analysis proves that all tasks comprising the system meet their respective deadlines. The input of a schedulability analysis are the runtimes of these tasks. The runtimes of a task vary depending on the task's actual inputs, the hardware state, and the interference of the physical environment (e.g., memory refreshes or DMA). In order to guarantee a system's timeliness, the *worst-case execution times* (WCET) of the tasks need to be known. Due to the halting problem, determining a task's runtime is impossible in general, but due to programming restrictions that are used within embedded real-time applications, runtime estimates can be given. The increasing demand for more and more computing power even in the embedded area has led to the introduction of hardware features from the field of personal computers such as out-of-order execution, deep processor pipelines, caches, and different kinds of speculation. Use of these features increases a system's average-case performance, but results in less timing-predictable architectures making a precise determination of a task's WCET impossible. WCET analysis has to deal with all these features and must provide a safe, but also precise upper bound on a task's execution times.

Determining safe and precise bounds on the WCET of a task has been subject to research since the last three decades; the resulting approaches can be classified into two main categories: measurement-based methods and static methods. *Measurement-based methods* utilize program executions and hardware execution traces to obtain the WCET, but in general, results cannot be guaranteed to be correct, as the results rely on some specific input that cannot be guaranteed to trigger a program's longest execution time path.

In contrast to that, *static methods* only consider the executable program and combine it with some (theoretical) model of the system to obtain a WCET estimate. Due to the history sensitivity of modern hardware features (e.g., an access might hit the cache due to a prior access of the same address), simple static approaches like instruction counting are rendered obsolete. The state space of inputs and initial hardware states that influences a task's runtime is too large to exhaustively explore all possible executions. Some abstractions of

the execution platform are 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.

Saarland University and AbsInt Angewandte Informatik GmbH have successfully developed the *aiT* WCET analyzer (cf. `http://www.absint.com/wcet.htm`) for computing safe upper bounds on the WCET of a task. The tool is used in the aeronautics [SPH+07] and automotive industries [KWH+08]. The tool architecture comprises three main phases: First, the control-flow of the executable program is reconstructed. In a second phase, basic block timings are determined using an abstract model of a processor and its periphery (*timing model*) to analyze how instructions pass through the processor's pipeline taking cache-hit or miss information into account. This computes a cycle-level abstract semantics of the instruction's execution yielding a certain set of final system states. Analysis is then restarted for the next instruction in all those states. Here, the timing model introduces non-determinism that leads to multiple possible execution paths in the analyzed program. The pipeline analysis needs to examine all of these paths. Finally, the resulting timing bounds for basic blocks are the coefficients in an Integer Linear Program (ILP) whose maximal solution yields the final WCET bound.

Timing models underlying the *aiT* analyzer are currently hand-crafted by human experts using processor documentation and hardware execution traces as source of modeling. [The04] describes the general workflow of utilizing abstract interpretation to obtain a timing model of a system. The system is partitioned into units maintaining an inner state, and update rules describe the evolution of states. The interaction of units is realized via typed signals. The resulting system model allows for a cycle-precise simulation of program execution. As system models tend to be large, even when focusing on timing relevant parts, components (i.e. parts of the units) can be abstracted using the framework of abstract interpretation [CC77, CC79]. Resulting abstract timing models can still be cycle-precisely simulated, but may include non-determinism. Development of a system model is time-consuming and error-prone. Finding abstractions that are suitable for timing analysis is the task of an experienced software engineer.

Current sophisticated processors used in the embedded area employ more and more advanced hardware features that improve the system's average-case performance. The increasing demand for more and more computing power and a reduced time to market require large development teams designing a system. Reliability, and thus simulation and verification of the designs are crucial issues. Formal hardware description languages are explicitly designed to support both, design and verification. The final hardware – realized on an ASIC or FPGA – is synthesized out of the formal hardware description. Formal verification, simulation and intensive testing are used to ensure that hardware

and specification are equivalent. Thus, cycle-accurate timing of the components comprising a system is already part of the formal specification.

However, hardware specifications of modern processors tend to be large. For that very reason, simulation of large programs with unknown inputs and unknown initial hardware state is impossible. Deriving a timing model that is suitable for the usage within static timing analyzers from a formal specifications would render both problems – the time-consuming and error-prone implementation – obsolete. Furthermore, the resulting timing model could be proven to be correct since both, hardware and model, rely on the same source.

This thesis summarizes the methodology for the derivation of timing models from hardware description languages that is presented in [Pis12] in more detail. Based on a formal specification of a processor (or component), a systematic reduction in size and complexity has to be achieved. A process consisting of two phases is presented that comprises a structured workflow to derive an abstract timing model suitable for its usage within the *aiT* timing analyzer framework.

The first phase, called *model preprocessing*, reduces the size of a hardware specification by employing statically available information. Model preprocessing is composed of three constituents: Parts of a specification that are irrelevant for the processor's (or component's) timing can be pruned out in order to reduce the specification's complexity. From the derivation methodology's point of view, a hardware specification is assumed to be correct. Thus, arithmetic operations are assumed to be correctly modeled, and therefore, information on the internal function of a multiplier are not interesting for timing analysis. Instead, it is sufficient to know, how many clock cycles an instruction occupies each stage of the multiplier's pipeline. Pruning out this information is called *timing dead code removal*.

However, the specification of a processor is still large and complex after timing dead code removal. Current embedded hardware architectures offer a huge variability concerning their configurability. E.g., caches can be configured to be used as scratchpads, partial cache locking can be enabled, unified caches can be used for instructions or data only, caching can be disabled, etc. Within the specific field of application in the embedded area, the configuration of the hardware is fixed. Thus, the fixed configuration renders parts of the specification inactive and allows for their removal. Besides these features, there exist a variety of events that are either asynchronous (e.g., DMA, or interrupts) or simply not predictable in their occurrence (e.g., ECC errors). A timing analysis cannot deal with these events, thus parts in a specification dealing with the handling of these events can be safely purged for timing analysis. Within the derivation methodology, this step is called *environmental assumption refinement*.

The third step in model preprocessing is the *removal of data paths* preparing a specification to be used within the *aiT* analyzer framework. In general, the

latency of instructions is not influenced by the contents of registers and memory cells. In other cases, the timing depends on such information, but we may choose to lose the exact timing knowledge in order to make the analysis more efficient, or even to make it possible at all. [Sic97] has shown that addresses generated by a task on a hardware can be separated from the task's execution. Thus, data paths can be pruned out of a specification. Whenever an address is needed for hardware simulation, an interface to the external address analysis results has to be introduced.

For complex architectures, the space required to represent a certain processor state may still be too large even after model preprocessing, rendering a timing analysis infeasible in terms of memory consumption. Therefore, the derivation methodology proposes a second phase, namely *processor state abstractions*, that deals with approximating parts of the processor state by providing powerful abstractions requiring less space but also loses some precision. One example of such a state approximation is the replacement of a concrete address by an interval abstraction. For cache-inhibited memory accesses, this loss in precision might be acceptable, as for timing analysis, the latency of the cache-inhibited access is only determined by the memory type that is addressed.

Using the methodology of abstract interpretation for processor state abstractions, we can trade precision of the analysis against efficiency by choosing different processor abstractions and concretization relations between the concrete processor state and the abstract one. To ease timing model derivation, the derivation methodology is embedded into a sound framework. The semi-automatic workflow starting with a processor specification given in VHDL – one of the most prominent hardware description languages – to derive a timing model suitable to be used within the *aiT* analyzer framework is presented in [Pis12].

A key part of the derivation framework are static analyses of hardware description languages that support especially environmental assumption refinement and timing dead code elimination, but also aid in model understanding supporting an engineer in finding suitable abstractions. This thesis presents a novel framework for generating sound and precise static analyses from high-level specifications. The static analysis framework is based on the framework of abstract interpretation, which is the most powerful de facto standard for designing static analyses. Naturally, the proposed analysis framework enjoys all the powers of the abstract interpretation framework such as provable correctness of analyses, a generic fixed-point engine for implementation, and a methodology for tuning analyses results.

To ease the development of static analyses, an abstract semantics for hardware description languages at the example of VHDL is introduced. The analysis framework and the abstract semantics are not limited to full designs, nor to synchronous circuits.

The abstract semantics is based on a transformed semantics, where the two-level semantics of VHDL is expressed by an equivalent sequentialized one-level semantics. The sequentialized representation enables applicability of common static analyses that are well-known from the compiler construction domain even for hardware description languages. In general, specifications given in VHDL can be interpreted in two different ways. The *standard simulation approach* is similar to the simulation of imperative languages. By this, a larger subset of specifications including behavioral descriptions can be handled. The *synthesis approach* to interpret a VHDL specification is closer to the final hardware (i.e. a specification is given at the level of register transfer), and is thus more interesting for the derivation of timing models. The running parts within a register-transfer level specification are processes, which are all collections of actions to be executed in sequence. These processes all run in parallel executing their list of actions; afterwards a process suspends. Dependencies between processes (via signals) enforce reactivation of certain processes, which induces *delta-delays* into a simulation. Delta-delays take conceptually zero time, and simulation time is only advanced, if all processes have been suspended. A static process schedule within a loop that is used by the synthesis approach to simulation assumes that delta-delays have already been eliminated. The analysis framework presented in this thesis combines both approaches to simulation by choosing a fixed process execution order, and adding a dedicated virtual simulation process to handle the delta-delays. Thus, process reactivation chains need not to be known as they are handled implicitly by the analysis framework.

To show the suitability of the analysis framework and the abstract semantics, and to manifest the applicability of the methodology to derive abstract timing models, this thesis further describes some static analyses supporting the derivation process. Realizing abstractions of the state of a processor that result in a more efficient analyzability, but also result in a still precise WCET estimate remains the task of an experienced software engineer, but static analyses can help in finding them. They support hardware model understanding in all phases and can assist a software engineer in the model preprocessing phase.

Assisting an engineer in environmental assumption refinement can be automated by the combination of two static analyses. First, the reset behavior of a system needs to be examined. The language standard defines initial values for all signals comprising a system, but usually, these values will be altered during the system's boot-up phase. As a system needs to sustain a stable and reproducible state also after a reset, the initial state (and also the initial values of signals and variables) is assigned to the system during the reset handling. In a complex computer system composed of different components connected via a system bus, and different peripheral devices, assigning the initial values to signals often takes more than one clock cycle. Identification of initial values assigned to signals can be automated by a constant propagation analysis that yields for each program point, whether or not a signal has a constant value

whenever execution reaches that point. Using the static analysis framework and the abstract semantics, a constant propagation can be easily extended to cope with the semantical characteristics of hardware description languages. Combining the resulting values for rising and falling clock cycles, the extended analysis, which is called *reset analysis*, computes the initial values assigned to signals during the system's reset handling. Besides this, reset analysis also computes a set of possible clock domains that are present in the specification.

Basing on the fixed configuration of a processor for its specific application context and building on a set of assumptions on the behavior of the surrounding environment (e.g., no occurrence of asynchronous events like DMA), parts of a processor specification become inactive. In order to identify these parts, we present the *assumption evaluation analysis* that takes a user-provided set of assumptions on the behavior of the system and its configuration, and the results of a prior reset analysis as inputs and computes a set of signals that become stable under the set of assumptions. Assignments to these stable signals are pruned out of the specification, and the remaining read references are replaced by the constant value. Also smaller co-domains for signals and variables being valid under the set of assumptions are computed. Results are then used to prune out parts of the specification that become unreachable resulting in a smaller specification. As this static analysis computes the transitive closure on stable signals and restricted co-domains, assumptions can also be provided iteratively. Thus, the assumption evaluation analysis allows to automate the environmental assumption refinement phase of the derivation methodology.

Beyond that, also the identification and removal of timing dead code can be automated. The goal of timing dead code elimination within the derivation methodology is to restrict a given hardware specification to statements that contribute to a processor's timing. Instructions executed by a processor can only have an influence on the processor's state as long as they are active within the execution pipeline. Thus, computing backward slices (cf. [Wei79]) for those points in a specification, where instructions leave the pipeline yields all program points that are influenced by instructions. All statements of a hardware specification that are not contained in the union over all these slices are definitively known to not influence a processor's timing behavior, and can thus be safely pruned from the specification in order to reduce the complexity. Identification of those points where instruction retire within a given hardware specification is to be done manually, but slicing can aid an engineer during this task. This thesis also presents a generic *backward slicing* algorithm for hardware description languages that supports both tasks. The algorithm is based on static analyses that are used to reconstruct control and flow dependencies contained in the hardware specification. Furthermore, process activation dependencies are taken into account. Given the retirement points of instructions as described above, slicing as presented in this thesis is able to automatically identify timing-dead code. Besides its applicability within the timing dead code elimination step of

the model preprocessing phase, slicing is useful in all other phases comprising the derivation methodology to support model understanding. Therefore, also an interactive version of the slicer has been developed.

Since all analyses presented within this thesis rely on the proposed analysis framework that is based on the framework of abstract interpretation, analyses can be proven to be correct. Thus, this thesis also presents a correctness proof for the three analyses that compose the slicing tool. Soundness considerations of analyses results utilizing a technique from the hardware verification domain, namely interval property checking [Bor09], are also subject of this thesis.

1.1 Related Work

This thesis presents a static analysis framework for hardware description languages that is based on the framework of abstract interpretation. To the best of our knowledge, nobody has tried before to partly automate the development process of a timing model for processors with the goal of WCET determination. Although the framework has been developed with the goal to support this automation, it is also applicable to other domains.

Static analysis techniques are still subject to research, but techniques for analysis of hardware description languages are scarce. The work that has been published in this area is mostly concerned with security aspects of hardware designs [LTH+10].

In the mid nineties, the SAVE project (Static Analysis for VHDL Evaluation) aims at defining and implementing a set of tools that support a circuit designer to improve the quality of VHDL descriptions. The quality of VHDL code is expressed using different metrics that are concerned with the complexity of the code, the simulation efficiency, the feasibility of the synthesis, and the testability of the circuit. In [Ste94], they present a framework that aims at visualizing these metrics and advising a designer to improve the VHDL code.

[BBS96] describes a static analysis technique based on data-flow analysis [Kil73] to analyze behavioral VHDL descriptions that are syntactical similar to procedural programming languages. In contrast to our approach that focuses on the analysis of hardware descriptions at the level of register transfer, their goal is the discovery of implementational errors, the identification of critical code fragments with respect to the synthesizability, and the derivation of test-patterns from the behavioral description that can be used to validate a register-transfer level simulation.

The analysis of information flow on synthesizable VHDL programs is described in [TNN05, Tol06]. Tolstrup et al. utilize static program analyses, such that

they result in tools that can automatically and efficiently verify specifications of integrated circuits and give exhaustive and correct assurances of Common Criteria objectives (cf. [ISO98]).

The closest related work on applying abstract interpretation to hardware description languages is proposed by Hymans in [Hym04]. In this work, the author describes a method for applying abstract interpretation to the verification of VHDL designs. The advantage of the approach is that data abstractions that are commonly applied in program analysis can be utilized to hardware designs. Given a property, the abstract interpreter can automatically confirm it, if it is satisfied. Thus, Hymans' technique allows to verify designs that are data-dominated, as opposed to control-dominated designs that are typically verified using model checking. The drawback of the method compared to interval property checking is the absence of temporal aspects that can be expressed within the properties. The work follows a simulation-based approach to model interpretation allowing also the analysis of behavioral descriptions, whereas the work presented within this thesis combines simulation- and synthesis-based approaches and focuses on register-transfer level descriptions. To mitigate inaccuracies and imprecision induced by delta-delays in the abstract interpretation when following a simulation-based approach, Hymans adds additional steps in the interpreter framework.

Besides these different techniques to apply classical program analyses to hardware description languages, there exist also some approaches to compute slices on them.

[CFR+99, CFR+02] describes an approach for slicing of VHDL that is also based on a program dependency graph. Language constructs from VHDL are mapped to language constructs of traditional procedural languages like C or ADA. In contrast to slicing as presented in this thesis introducing a new kind of dependency, namely activation dependency, to cope with the reactive nature that is special to hardware description languages, the authors introduce a particular master process for the simulation of process reactivations.

Slicing as described in [RKK04] is limited to synchronous circuit specification and focuses especially on the event-oriented communication structure of VHDL designs, whereas our slicing algorithm is not limited to synchronous circuit specifications.

1.2 Structure of this Thesis

In the next chapter, the mathematical foundations of this thesis will be briefly described. Basics of abstract interpretation, fixed-point theory, and data-flow

analysis are given. Also algorithms for solving data-flow problems are detailed.

In Chapter 3, current methods for computing worst-case execution time bounds of a task are detailed, and an overview over nowadays tools for computing runtime estimates is given. The *aiT* WCET analyzer is described in more detail, and its micro-architectural analysis part is presented.

Chapter 4 discusses hardware features that have been mainly developed to increase the average-case performance of modern processors. Also their impact on the timing predictability of the system employing these features is investigated. Furthermore, current hardware development using formal hardware description languages that ease interoperability, simulation, synthesis, and verification is detailed, and the semantics of VHDL as a prominent example is defined.

Chapter 5 introduces the concept of transformations of VHDL and provides its abstracted semantics. Building on these transformations, the methodology for the derivation of timing models is introduced. To ease timing model derivation, the derivation methodology is embedded into a consistent framework building on static analyses and transformations of VHDL.

The novel framework for static analysis of VHDL that is the key part of the derivation framework and the transformation of VHDL to the new abstract semantics is detailed in Chapter 6. The usability of the abstract semantics is illustrated by three different analyses that also support the derivation methodology. A correctness proof is also given.

The subject of Chapter 7 is the implementation of the described analyses. Research as well as industrial specifications of processors or components are used to evaluate the applicability of the tools. Also soundness considerations of the analyses results and code of work rules for deriving timing models are given.

Chapter 8 summarizes the contributions of this thesis and provides an outlook to future work.

2

Preliminaries

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 most bitter.

(Confucius)

This chapter gives a brief survey on the mathematical concepts underlying this thesis. Besides a short introduction to lattice theory and fixed-point theory, also the theory of static program analysis in the framework of abstract interpretation [CC77, CC79, Cou81, CC91, CC92a, CC92b, Cou01, NNH99] is presented. Finally, algorithms for computation of safe results and the concepts of interprocedural analyses are presented.

2.1 Lattice Theory

Definition 2.1.1 (Partially ordered set)
Given a set P. A relation $\sqsubseteq_P \subseteq P \times P$ is called *partial ordering*, iff it is reflexive, transitive and anti-symmetric. $(P, \sqsubseteq_P)$ is called a *partially ordered set*.

Definition 2.1.2 (Ascending chains, descending chains)
Given a partially ordered set $(P, \sqsubseteq_P)$. A subset $X \subseteq P$ is called a *chain*, iff

$$\forall x_1, x_2 \in X : (x_1 \sqsubseteq_P x_2) \vee (x_2 \sqsubseteq_P x_1).$$

X is also called a *totally ordered* subset of P.

A sequence $p_0, p_1, p_2, \ldots$ with $p_i \in P$ is called an *ascending chain*, if

$$a \leqslant b \implies p_a \sqsubseteq_P p_b.$$

Similarly, the sequence is called a *descending chain*, if

$$a \leqslant b \implies p_b \sqsubseteq_P p_a.$$

A partially ordered set $(P, \sqsubseteq_P)$ is said to satisfy the *ascending chain condition* if every ascending chain of elements eventually stabilizes, i.e. there exists a positive integer m such that

$$p_m = p_{m+1} = p_{m+2} = \ldots$$

Similarly, $(P, \sqsubseteq_P)$ is said to satisfy the *descending chain condition* if every descending chain of elements eventually stabilizes, i.e. there is no infinite descending chain.

Definition 2.1.3 (Upper bounds, lower bounds)
Given a partially ordered set $(P, \sqsubseteq_P)$ and a set $X \subseteq P$. An element $u \in P$ is called *upper bound* of X, iff

$$\forall x \in X : p \sqsubseteq_P u.$$

An element $u \in S$ is called *least upper bound* of X, $\bigsqcup X$, iff u is an upper bound of X and for all other upper bounds a of X, $u \sqsubseteq_P a$ holds.

Analogously, an element $l \in P$ is called *lower bound* of X, iff

$$\forall x \in X : l \sqsubseteq_P x.$$

$l \in S$ is called *greatest lower bound* of X, $\bigsqcap X$, iff l is a lower bound of X and for all other lower bounds a of X, $a \sqsubseteq_P l$ holds.

For convenience, $a \sqcup b$ is used for $\bigsqcup\{a, b\}$, and $a \sqcap b$ for $\bigsqcap\{a, b\}$.

Lemma 2.1.1
In a partially ordered set satisfying the ascending chain condition, every ascending chain $p_0, p_1, \ldots$ has a least upper bound, namely $\bigsqcup_{n \in \omega} p_n = p_m$, with $\omega = (N, \leqslant)$, where m is the index where the chain stabilizes.

Definition 2.1.4 (Complete lattice)
A partially ordered set $(L, \sqsubseteq_L)$ is called a *complete lattice*, iff for every set $P \subseteq L$, $\bigsqcap P$ and $\bigsqcup P$ exist. A complete lattice has a *least element* $\bot$ with $\bot = \bigsqcup \varnothing$ and a *greatest element* $\top$ with $\top = \bigsqcup L$. Though it is not mathematically precise, we often write a complete lattice as the tuple $(L, \sqsubseteq_L, \sqcup, \sqcap, \bot, \top)$.

2.2 Fixed-Point Theory

Sometimes, one needs to solve a recursive equation system, i.e. an equation system, where the value to be defined also occurs on the right side of the formula. A well-known example of such a recursive equation system is the factorial function defined by:

$$fac(n) = \begin{cases} 1 & \text{if } n = 0, \\ n * fac(n-1) & \text{otherwise.} \end{cases}$$

Every solution has to fulfill this equation. In order to solve a recursive definition, a simple iterative approach can be used: Starting with the smallest element of the solution space $\bot$, the definition for the proximate larger element is obtained. Applying this approach n-times results in a function being defined for the interval $[0, \ldots, n-1]$. Generating the limit $n \mapsto \infty$ yields the function defined for the natural numbers.

This iterative method can be formalized as follows:

Definition 2.2.1 (Montone function, distributive function)
A function $f\colon D_1 \to D_2$, where $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ are partially ordered sets, is called

- *monotone function*, if $\forall d, d' \in D_1 : d \sqsubseteq_1 d' \implies f(d) \sqsubseteq_2 f(d')$,
- *distributive function*, if $\forall d, d' \in D_1 : f(d \sqcup_1 d') = f(d) \sqcup_2 f(d')$.

Definition 2.2.2 (Pre-fixed point)
Given a function $f\colon D \to D$ on a partially ordered set $(D, \sqsubseteq_D)$. A *pre-fixed point* of f is an element $d \in D$, such that $f(d) \sqsubseteq_D d$.

Definition 2.2.3 (Fixed point)
Given a function $f\colon D \to D$. An element $d \in D$ is called *fixed point* of f, iff $f(d) = d$.

Theorem 2.2.1 (Fixed-point iteration)
Given a partially ordered set $(L, \sqsubseteq_L)$ satisfying the ascending chain condition and having a least element $\bot$, and a monotone function $f\colon L \to L$. Let $lfp(f) = \bigsqcup_{n\in\omega} f^n(\bot)$, with $\omega = (N, \leqslant)$. Then, $lfp(f)$ is a fixed point and also the least pre-fixed point of f. The following holds:

1. $f(lfp(f)) = lfp(f)$, and
2. $f(l) \sqsubseteq_L l \implies lfp(f) \sqsubseteq_L l$.

Proof 2.2.1
In order to prove this, both propositions must be shown separately:

to 1.

$$\begin{aligned} f(\mathit{lfp}(f)) &= f\left(\bigsqcup_{n\in\omega} f^n(\bot)\right) \\ &= \bigsqcup_{n\in\omega} f^{n+1}(\bot) \\ &= \left(\bigsqcup_{n\in\omega} f^{n+1}(\bot)\right) \sqcup \{\bot\} \\ &= \bigsqcup_{n\in\omega} f^n(\bot) \\ &= \mathit{lfp}(f) \end{aligned}$$

to 2. Let l be a pre-fixed point. It holds that $\bot \sqsubseteq_L l$. Due to the monotonicity of f, $f(\bot) \sqsubseteq_L f(l)$. l being a pre-fixed point of f implies $f(\bot) \sqsubseteq_L l$. By induction, $f^n(\bot) \sqsubseteq_L l$ and thus, $\mathit{lfp}(f) = \bigsqcup_{n\in\omega} f^n(\bot) \sqsubseteq_L l$. Since every fixed point is also a pre-fixed point, $\mathit{lfp}(f)$ is the least fixed point of f.

□

2.3 Abstract Interpretation

Abstract interpretation [CC77] is a theory of sound approximation of program semantics, based on monotonic functions over ordered sets. It is applicable in static analysis, i.e. the extraction of information about the possible executions of programs. Often, a system's semantics needs to be approximated. To guarantee that only valid properties are derived, the approximation needs to be sound. The framework of abstract interpretation provides a theory of deriving a sound approximation of a concrete semantics. This section describes the framework in more details.

A program might start its execution in different initial states. All of them need to be taken into account in a static analysis in order to compute reliable properties of the program. In order to be able to take all initial states into account, the concrete semantics working on one individual state is to be lifted to a collecting semantics working on sets of concrete states.

Definition 2.3.1 (Control-flow graph)
A program P can be represented by its *control-flow graph* $G = (V, E, s, x)$, with a set V of *vertices*, and a set $E \subseteq V \times V$ of *edges*, where:

1. the nodes $v \in V \backslash \{s, x\}$ represent program statements from P,
2. the edges E represent possible control flow,

3. s represents the start node, i.e. $\forall v \in V : (v, s) \notin E$, and

4. x represents the end node, i.e. $\forall v \in V : (x, v) \notin E$.

The control-flow graph representation is useful for program analysis. After running an analysis, information needs to be extracted from the control-flow graph. The information extracted is the *collecting semantics*. It can be *state-based* (also called "first-order", [Nie82]) mapping a program point to the range of values that enter the program point, or *path-based* ("second-order") mapping a program point to the set of paths that lead into the program point. In the following, the thesis focuses on the more general path-based collecting semantics.

Definition 2.3.2 (Path)
A sequence $\pi = (v_1, v_2, \ldots, v_n)$ with $v_i \in V$ is called a *path* through a control-flow graph (V, E, s, x), iff

1. $v_1 = s$, and

2. $\forall i \in \{1, \ldots, n-1\} : (v_i, v_{i+1}) \in E$.

A sequence $\pi = (v_1, v_2, \ldots, v_n)$ is called a *path to* v, iff π is a path and $v_n = v$. The set $P[v_a, v_b]$ denotes all subsequences $(v_a, \ldots, v_b)$ of paths to v_b that contain v_a.

Given a transformer $f : E \to D \to D$ that computes the effect of the program statement assigned to a node v being the source of the edge $e = (v, w) \in E$ for a given state $s \in D$, the semantics of a path π can be defined.

Definition 2.3.3 (Path semantics)
Given a transformer $f : E \to D \to D$, the *semantics of a path* $\pi = (v_1, v_2, \ldots, v_n)$ is defined as

$$[\![\pi]\!] = \begin{cases} \lambda x.x & \text{if } \pi = (s), \\ f((v_{n-1}, v_n)) \circ [\![(v_1, \ldots, v_{n-1})]\!] & \text{otherwise.} \end{cases}$$

Program analyses normally shall compute properties for sets of initial states. Thus, the semantics of a path has to be lifted to compute the semantics of a path for a set of initial states $S \in \mathcal{D} = \mathcal{P}(D)$. Therefore, also the definition of the transformer-function f must be extended to deal with set of states:

$$\mathfrak{f} : E \to \mathcal{D} \to \mathcal{D} : \mathfrak{f}(e)(S) := \{f(e)(s) \mid s \in S\}$$

Using this transformer $\mathfrak{f}$, the collecting path semantics can be defined.

Definition 2.3.4 (Collecting path semantics)
Given a transformer $\mathfrak{f}\colon E \to \mathcal{D} \to \mathcal{D}$, the *collecting path semantics* for a path $\pi = (v_1, v_2, \ldots, v_n)$ is defined as

$$[\![\pi]\!]_{coll} = \begin{cases} \lambda x.x & \text{if } \pi = (s), \\ \mathfrak{f}((v_{n-1}, v_n)) \circ [\![(v_1, \ldots, v_{n-1})]\!]_{coll} & \text{otherwise.} \end{cases}$$

In program analysis, one often wants to know a certain property being valid at a dedicated program point, i.e. a node of the control-flow graph. E.g., a constant propagation shall determine the set of variables being constant at a particular program point. This can be formalized by the "sticky" collecting semantics mapping a node v to the set of concrete states that might be observable at v.

Definition 2.3.5 (Sticky collecting semantics)
Given a program P represented by its control-flow graph $G_P = (V, E, s, x)$, the *sticky collecting semantics* $Coll_P\colon V \to \mathcal{D}$ for P on a set of concrete initial states $I \subseteq D$ is defined as:

$$Coll_P(v) = \bigcup \{[\![\pi]\!]_{coll}(I) \mid \pi \text{ is a path to } v\}$$

It should be noted that different properties that shall be computed for a program may require different concrete domains. Whereas the sticky collecting semantics is defined under union, other properties may require a semantics build up under intersection. Nevertheless, the above definition is sufficient, as a problem to be solved on a lattice under disjunction can be transformed to a problem to be solved on the *dual lattice* under union. Anyhow, the sticky collecting semantics as presented is adequate for a constant propagation analysis, whereas a live-variables analysis would require a trace-based semantics (cf. [Nie82]).

Both the sticky collecting semantics and the trace semantics are not computable in general. The set of initial states might be tremendously or even infinitely large, and also the number of paths leading to a certain point may be infinite due to loops. To solve nonetheless problems as described above, replacing the concrete domain by a more abstract one makes computation feasible again, usually at the cost of precision.

Sets of concrete states in $\mathcal{D}$ can be represented by elements from an abstract domain $\mathcal{D}_{abs}$. Ideally, the new abstract domain is a complete lattice with an ordering relation $\sqsubseteq_{abs}$. The existence of a least upper bound for all subsets (cf. Definition 2.1.4 on page 12) allows to uniquely combine elements of the abstract domain. Analogously to the collecting path semantics, a transformation function $\mathfrak{f}_{abs}\colon E \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$ is required to compute the effect of program statements on the abstract domain.

Arguing about the correctness of an abstraction requires to establish a relation between an element from the abstract domain $\mathcal{D}_{abs}$ and the concrete domain $\mathcal{D}$.

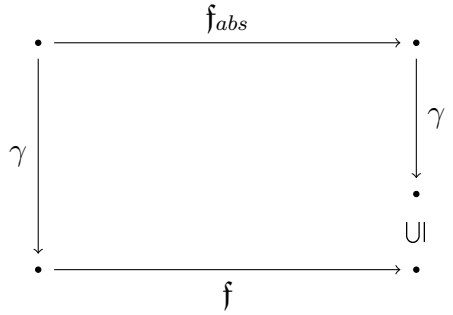


Figure 2.1 – Basic concept of abstract interpretation.

This can be achieved by a monotone *concretization function* $\gamma\colon \mathcal{D}_{abs} \to \mathcal{D}$ mapping an abstract state to the set of concrete states it represents. Figure 2.1 depicts the general idea of abstract interpretation. The goal is to find an abstraction where computation is much faster than in the concrete domain. This usually leads to some loss of precision. The monotonicity of γ ensures that the partial order $\sqsubseteq_{abs}$ on $\mathcal{D}_{abs}$ orders abstract states according to their precision. Furthermore, the concretization of an abstract state will be a superset of the concretization of a more precise abstract state, i.e.

$$a \sqsubseteq_{abs} b \implies \gamma(a) \sqsubseteq \gamma(b).$$

Using the abstract transformation function $\mathfrak{f}_{abs}$, the abstract collecting path semantics can be defined as follows.

Definition 2.3.6 (Abstract collecting path semantics)
Given an transformation function $\mathfrak{f}_{abs}\colon E \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$, the *abstract collecting path semantics* for a path $\pi = (v_1, v_2, \ldots, v_n)$ is defined as

$$[\![\pi]\!]_{abs} = \begin{cases} \lambda x.x & \text{if } \pi = (s), \\ \mathfrak{f}_{abs}((v_{n-1}, v_n)) \circ [\![(v_1, \ldots, v_{n-1})]\!]_{abs} & \text{otherwise.} \end{cases}$$

Definition 2.3.7 (Local consistency)
Given a control-flow graph $G_P = (V, E, s, x)$ and a concretization function $\gamma\colon \mathcal{D}_{abs} \to \mathcal{D}$. An abstract transformer $\mathfrak{f}_{abs}\colon E \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$ is said to be *locally consistent* with a collecting transformer $\mathfrak{f}\colon E \to \mathcal{D} \to \mathcal{D}$, iff

$$\forall e \in E, s \in \mathcal{D}_{abs}\colon \mathfrak{f}(e)(\gamma(s)) \subseteq \gamma(\mathfrak{f}_{abs}(e)(s)).$$

Thus, $\mathfrak{f}_{abs}$ shall over-approximate the behavior of $\mathfrak{f}$, i.e. $\forall e \in E\colon \mathfrak{f}(e)(\gamma(S_{abs})) \sqsubseteq \mathfrak{f}_{abs}(e)(S_{abs})$.

Local consistency guarantees soundness of the construction. If for all edges e in a control-flow graph $G_P = (V, E, s, x)$, it holds that

$$\mathfrak{f}(e)(\gamma(s)) = \gamma(\mathfrak{f}_{abs}(e)(s)),$$

the abstract transformer $\mathfrak{f}_{abs}$ is also called *best transformer*.

Theorem 2.3.1 (Soundness of $[\![\ ]\!]_{abs}$)
The abstract collecting path semantics defined by $[\![\ ]\!]_{abs}$ is a sound approximation of the collecting path semantics $[\![\ ]\!]_{coll}$, i.e.

$$\forall d \in \mathcal{D}_{abs} \colon ([\![\pi]\!]_{coll} \circ \gamma)(d) \subseteq (\gamma \circ [\![\pi]\!]_{abs})(d),$$

iff $\mathfrak{f}_{abs}$ is locally consistent.

Proof 2.3.1
see [Rei08]. □

Similar to the concretization function γ, it is sometimes possible to give a monotone function α that computes for a given set of concrete states the *best* abstract state. Both functions, α and γ shall form a Galois connection:

Definition 2.3.8 (Galois connection)
Let L and M be complete lattices $(L, \sqsubseteq)$ and $(M, \leqslant)$, and $\alpha \colon L \to M$ and $\gamma \colon M \to L$ be monotone functions. The tuple (L, α, γ, M) is called a *Galois connection*, iff

$$\forall l \in L \colon l \sqsubseteq \gamma(\alpha(l))$$

and

$$\forall m \in M \colon \alpha(\gamma(m)) \leqslant m.$$

α and γ are called *abstraction* and *concretization* function, respectively.

Figure 2.2 on the facing page illustrates the relationship induced by the abstraction and concretization function. Intuitively, the first condition ensures that by abstracting and concretizing an element, the result will be less precise than the starting element. Loosing some precision is acceptable since the abstract domain shall abstract some details of the concrete domain, and thus, an element from the abstract domain usually represents a set of concrete elements. The second condition guarantees that α computes precise approximations of concrete states. A Galois connection (L, α, γ, M) is called a *Galois insertion*, if $\alpha(\gamma(m)) = m$ holds for all $m \in M$.

Definition 2.3.9 (Abstract sticky collecting semantics)
Given a program P represented by its control-flow graph $G_P = (V, E, s, x)$, the *abstract sticky collecting semantics* $Coll_P^{abs} \colon V \to \mathcal{D}_{abs}$ for P on an abstract initial state $I_{abs} \in \mathcal{D}_{abs}$ is defined as:

$$Coll_P^{abs}(v) = \bigsqcup \{[\![\pi]\!]_{abs}(I_{abs}) \mid \pi \text{ is a path to } v\}$$

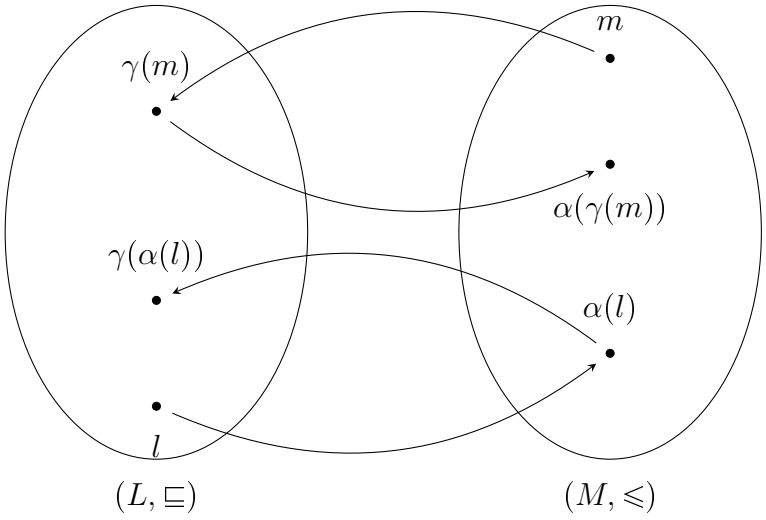


Figure 2.2 – Galois connection (L, α, γ, M).

Theorem 2.3.2 (Soundness of $Coll_P^{abs}$)
The abstract sticky collecting semantics $Coll_P^{abs}$ is a sound approximation of the sticky collecting semantics $Coll_P$, i.e.

$$\forall v \in V : \; Coll_P(v) \subseteq \gamma(Coll_P^{abs}(v)),$$

iff $\mathfrak{f}_{abs}$ is locally consistent and I_{abs} is the abstraction of the set of concrete initial states I, i.e. $I_{abs} = \alpha(I)$.

Proof 2.3.2
see [Rei08]. □

2.4 Data-Flow Analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program [Kil73]. In contrast to abstract interpretation, data-flow analysis does not have an inherent notion of correctness in matters of a concrete semantics. Given the abstract transformation function $\mathfrak{f}_{abs}$, data-flow analysis can compute an over-approximation of the abstract sticky collecting semantics $Coll_P^{abs}$. Abstract interpretation provides semantical soundness to data-flow analyses, however, abstract interpretation is also applicable to other domains.

Data-flow analysis is inherently flow-sensitive and typically path-insensitive, but it is possible to define data-flow equations that yield a path-sensitive analysis.

A *flow-sensitive* analysis takes into account the order of statements in a program, whereas a *path-sensitive* analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions, i.e. at nodes in the control-flow graph with more than one successor [Hec77, BA98]. This thesis will use the more general definition in order to allow also path-sensitive analyses.

Definition 2.4.1 (Data-flow problem)
Given a control-flow graph $G = (V, E, s, x)$, a complete lattice $(L, \sqsubseteq_L, \sqcup, \sqcap, \bot, \top)$ and a transformation function $f : E \to L \to L$. The tuple (G, L, f) is called a *data-flow problem*. The functions $f(e)$ are called *transfer functions*.

Please note that the abstract transfomer $\mathfrak{f}_{abs}$ on the abstract domain $\mathcal{D}_{abs}$ and a control-flow graph G_P also forms a data-flow problem $dfp_{abs} = (G_P, \mathcal{D}_{abs}, \mathfrak{f}_{abs})$.

Definition 2.4.2 (Monotone data-flow problem)
Let $dfp = (G, L, f)$ be a data-flow problem. dfp is called *monotone*, iff $f(e) : L \to L$ is monotone for all $e \in E$.

Traditional data-flow analysis does not define correctness in terms of a concrete semantics, however in data-flow analysis, the sticky collecting semantics is called *meet-over-all-paths* (MOP) solution [KU76, Nie82]. In the following, this thesis will use MOP_P to denote its solution.

Unfortunately, the definition of the MOP solution does not provide a direct way for its computation. Especially in programs containing loops, computing the semantics of all paths is impossible. [KU77] has shown that computing the MOP solution is not possible in general. However, under certain conditions, it is possible to compute a safe approximation to the MOP solution, the so-called *maximal fixed-point solution* (MFP).

Definition 2.4.3 (Maximal fixed-point solution)
Given a data-flow problem (G_P, L, f) with $G_P = (V, E, s, x)$ of a program P, a complete lattice L and a transformation function $f : E \to L \to L$. The *maximal fixed-point solution* $MFP_P : V \to L$ for an initial state $i \in L$ is the least fixed-point of the recursive equation:

$$MFP_P(v) = \begin{cases} i, & \text{if } v = s, \\ \bigsqcup\{f(e)(MFP(v')) \mid e = (v', v) \in E\}, & \text{otherwise.} \end{cases}$$

The name "maximal fixed-point solution" is somewhat misleading, but for all that used for historical reason; classical literature tends to focus on analyses where $\sqcup$ equals to $\cap$, which implies that the least fixed-point with respect to $\supseteq$ equals the greatest fixed-point with respect to $\subseteq$.

Using the definitions from above, the data-flow problem $dfp_{abs} = (G_P, \mathcal{D}_{abs}, \mathfrak{f}_{abs})$ can be safely approximated by the MFP solution. If the transformation function satisfies some properties, one can show the correctness, and also the coincidence of both solutions, the MOP and MFP solution.

Theorem 2.4.1 (Correctness and coincidence)
Given a control-flow graph $G_P = (V, E, s, x)$ for a program P and a monotone data-flow problem $dfp = (G_P, L, f)$. The maximal fixed-point solution is a safe approximation of the meet-over-all-paths solution:

$$\forall v \in V : MOP_P(v) \subseteq MFP_P(v)$$

Additionally, if $f : E \to L \to L$ is distributive, the following holds:

$$f \text{ is distributive} \implies \forall v \in V : MOP_P(v) = MFP_P(v)$$

Proof 2.4.1
see [NNH99]. □

Implicitly, definitions given here can be used to tackle *forward problems*, where information is propagated in program order, i.e. the computed information depends on the control-flow predecessors. But there exist also problems (e.g., live variables) that require a *backward analysis*, where information is computed considering the control-flow successors. However, the definitions presented here suffice, as a backward problem on a control-flow graph $G = (V, E, s, x)$ can be solved as a forward problem on the inverted control-flow graph $G^{-1} = (V, E^{-1}, x, s)$, with $E^{-1} = \{(n, m) | (m, n) \in E\}$.

2.4.1 Algorithms

There exist different approaches to solve data-flow problems. The most significant ones are the iterative algorithms, which will be presented here in more detail.

Intuitive Method

The maximal fixed-point solution as defined in Definition 2.4.3 on the preceding page already describes a recursive formula for its computation, which can be directly implemented.

Given a control-flow graph $G_P = (V, E, s, x)$ for a program P, a complete lattice L and a transformer function $f : E \to L \to L$, the algorithm depicted in Listing 2.1 on the following page computes the desired information. Termination of the algorithm is only guaranteed if the data-flow problem $dfp = (G_P, L, f)$ is monotone, and L satifies the ascending chain condition.

```
forall n ∈ N
    MFP(n) = ⊥;
MFP(s) = i;
done = false;

while (!done)
    done = true;
    forall (m, n) ∈ E
        temp = f((m, n))(MFP(m))
        if (temp ≠ MFP(n))
            MFP(n) = MFP(n) ⊔ temp;
            done = false;
```

Listing 2.1 – Intuitive method for computing the MFP-solution.

Worklist Iteration

Termination of the intuitive computation method can be enforced: instead of recomputing the data-flow value for each node in the given control-flow graph in each loop iteration, only the data-flow value of those nodes needs to be recomputed, where the input value has changed. Therefore, a worklist containing all nodes that need to be revisited is introduced. The modified version of the algorithm is shown in Listing 2.2 on the next page.

A further computational speed-up can be achieved by using some heuristics for selecting the next node (cf. [TMAL98]).

Basic-block Optimization

Another important way in speeding up the computation is to make use of basic blocks instead of single nodes of the control-flow graph.

Definition 2.4.4 (Basic block)
Given a control-flow graph $G = (V, E, s, x)$. A sequence of nodes $v_1, \ldots, v_n$ is called a *basic block*, iff $\forall i \in \{1, \ldots, n-1\}$:

$$\begin{aligned} & \nexists (v, v_{i+1}) \in E \backslash \{(v_i, v_{i+1})\} \\ \wedge \quad & \nexists (v_i, v) \in E \backslash \{(v_i, v_{i+1})\} \end{aligned}$$

and $\nexists v \in V$, such that $v, v_1, \ldots, v_n$ nor $v_1, \ldots, v_n, v$ have this property.

During the iteration, it is sufficient to store the data-flow value only at nodes with more than one successor. Data-flow values at nodes with one successor are

```
forall n ∈ N
    MFP(n) = ⊥;
MFP(s) = i;
workset = {n | (s, n) ∈ E};

while (workset ≠ ∅)
    let
        n ∈ workset
    in
        workset = workset\{n};
        temp = ⨆{f(e)(MFP(m)) | e = (m, n) ∈ E}
        if (temp ≠ MFP(n))
            MFP(n) = temp;
            workset = workset ∪ {m | (n, m) ∈ E};
```

Listing 2.2 – Worklist method for computing the MFP-solution.

propagated to that successor by means of the transfer function of the intervening edge. This is repeated until a node with more than one successor is reached (*chain optimization*). After the iteration, the information is propagated to all nodes by a post-processing step.

Widening and Narrowing

A possibility to enforce termination of a data-flow analysis is using widening and narrowing as introduced in [CC77] and [CC92a]. Using these operators guarantees termination even though the underlying lattice does not satisfy the ascending chain condition. Using widening even makes sense in cases where computing a precise result would take too much computational effort, at the cost of precision. The approximation found in this way can then be improved using narrowing.

Definition 2.4.5 (Widening)
Let $(L, \sqsubseteq_L, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. An operator $\bigtriangledown\colon L \times L \to L$ is called *widening operator*, iff:

$$\begin{aligned} & \forall l, m \in L\colon l \sqsubseteq_L (l \bigtriangledown m) \\ \wedge \quad & \forall l, m \in L\colon m \sqsubseteq_L (l \bigtriangledown m) \end{aligned}$$

and for every ascending chain $l_1 \sqsubseteq_L l_2 \sqsubseteq_L \cdots$, the ascending chain $m_1 \sqsubseteq_L m_2 \sqsubseteq_L \cdots$, constructed by

$$\begin{aligned} m_1 &= l_1 \\ m_{i+1} &= m_i \bigtriangledown l_{i+1} \end{aligned}$$

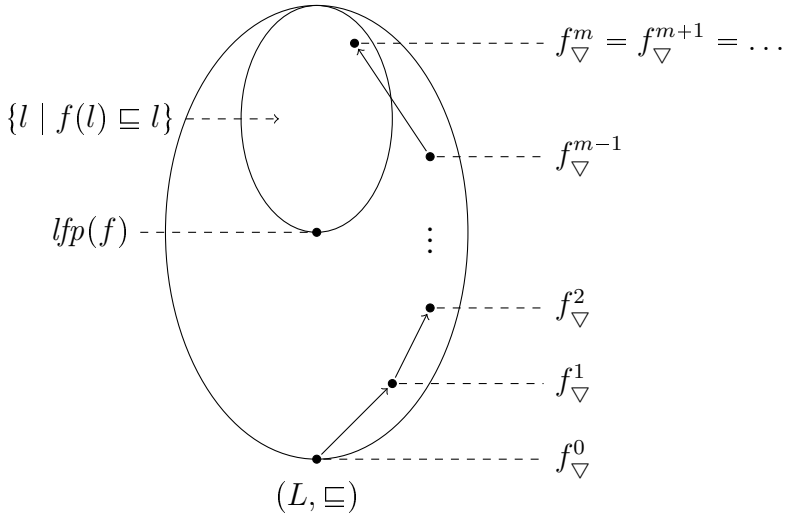


Figure 2.3 – The widening operator $\triangledown$ applied to f.

eventually stabilizes.

Given a monotone function $f\colon L \to L$ on a complete lattice L and given a widening operator $\triangledown$ on L, one can construct the sequence $f^0_\triangledown, f^1_\triangledown, \ldots$ by

$$f^n_\triangledown = \begin{cases} \bot & \text{if } n = 0, \\ f^{n-1}_\triangledown & \text{if } n > 0 \wedge f(f^{n-1}_\triangledown) \sqsubseteq_L f^{n-1}_\triangledown, \\ f^{n-1}_\triangledown \triangledown f(f^{n-1}_\triangledown) & \text{otherwise.} \end{cases}$$

Following this construction, f is *reductive* at a certain point $f^m_\triangledown$, i.e. $f(f^m_\triangledown) \sqsubseteq_L f^m_\triangledown$. Theorem 2.2.1 implies that $f^m_\triangledown \sqsupseteq lfp(f)$, and thus, the widening operator yields the desired approximation. Figure 2.3 illustrates this iteration sequence.

Use of a widening operator thus leads to an over-approximation of the desired least fixed-point. In order to improve the approximation provided by $f^m_\triangledown$, a narrowing operator establishing a descending chain criterion can be used.

Definition 2.4.6 (Narrowing)
Let $(L, \sqsubseteq_L, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. An operator $\triangle\colon L \times L \to L$ is called *narrowing operator*, iff:

$$\forall l, m \in L\colon (l \sqsubseteq_L m) \implies (l \sqsubseteq_L (m \triangle l) \sqsubseteq_L m)$$

and for every descending chain $l_1 \sqsupseteq_L l_2 \sqsupseteq_L \cdots$, the descending chain $m_1 \sqsupseteq_L m_2 \sqsupseteq_L \cdots$, constructed by

$$\begin{aligned} m_1 &= l_1 \\ m_{i+1} &= m_i \triangle l_{i+1} \end{aligned}$$

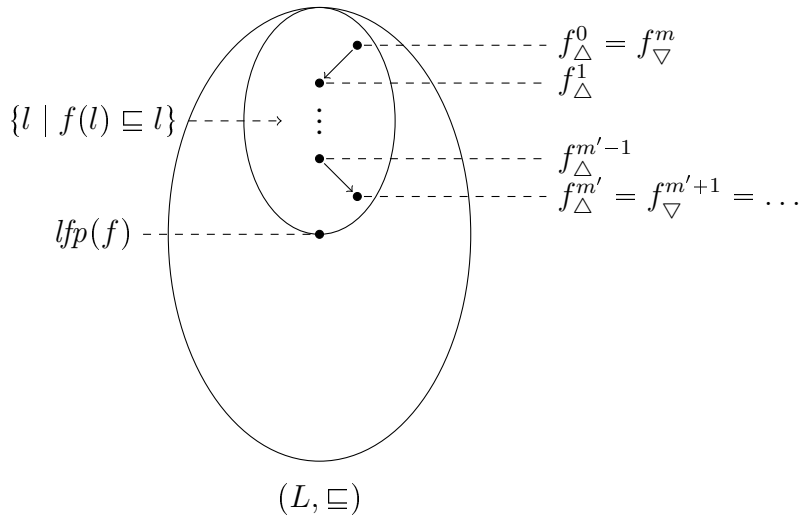


Figure 2.4 – The narrowing operator $\triangle$ applied to f.

eventually stabilizes.

Given a narrowing operator $\triangle\colon L \to L$, and a point $f^m_\triangledown$ satisfying $f(f^m_\triangledown) \sqsubseteq_L f^m_\triangledown$, one can construct the sequence $f^0_\triangle, f^1_\triangle, \ldots$ as follows:

$$f^n_\triangle = \begin{cases} f^m_\triangledown & \text{if } n = 0, \\ f^{n-1}_\triangle \triangle f(f^{n-1}_\triangle) & \text{if } n > 0 \end{cases}$$

Figure 2.4 illustrates the sequence. Intuitively, the narrowing operator is used to reduce the over approximation introduced by use of a widening operator. [NNH99] proofs that the sequence $f^0_\triangle, f^1_\triangle, \ldots$ is a descending chain in the *reductive subset*[1] of the given complete lattice L, and thus $f^n_\triangle \sqsupseteq_L lfp(f)$ for all n.

2.4.2 Interprocedural Analysis

Up to now, only programs with one procedure have been considered. However, this is not the general case, and preferably, data-flow analysis has to deal with more complex programs including procedures, loops and recursion.

A trivial approach to deal with complex programs would be to treat procedures as conservatively as possible. From the point-of-view of a data-flow analysis, all data-flow values computed so far need to be invalidated at every call instruction. This approach is easy to implement, but imprecise.

[1]The reductive subset of a lattice L is defined by $\{l \in L \mid f(l) \sqsubseteq l\}$.

A typical program consists of more than one procedure, which may be called from different places in the program with different arguments and different values for global variables. Best data-flow analysis results can be obtained by inlining the called function at the place of the call instruction. This approach definitively separates all procedure calls, but cannot be applied to recursive procedures.

In general, a procedure called more than once will have different call contexts, which lead to different updates of data values in the called procedure. Best results can be obtained when the procedure is analyzed separately for each call context. First, one needs to define a special interprocedural control-flow graph, the so-called supergraph.

Definition 2.4.7 (Supergraph)
Given a program P with procedures $P_0, P_1, \ldots, P_n$. Let P_0 be the main procedure of P, and let $G_0, G_1, \ldots, G_n$ with $G_i = (V_i, E_i, s_i, x_i)$ be the control-flow graphs of these procedures. The *supergraph* $G^* = (V^*, E^*, s^*, x^*)$ for the program P is the collection of control-flow graphs $G_0, G_1, \ldots, G_n$. Each procedure call in the program P is represented in G^* by two nodes, a *call node* and a *return node*. In addition to the ordinray intrapocedural edges that connect the nodes of the individual control-flow graphs G_i, for each procedure call – represented by call node c and return node r – G^* contains three egdes: an intraprocedural *local edge* from c to r, an interprocedural *call edge* from c to the starting node s_i of the called procedure P_i, and an interprocedural *return edge* from the exit node x_i of the called procedure P_i to the return node r.

To separate the different contexts of a procedure call, the *call-string approach* introduced by [SP81] can be used:

Definition 2.4.8 (Call string)
Given a supergraph $G^* = (V^*, E^*, s^*, x^*)$ and a path $\pi = (v_1, v_2, \ldots, v_n)$. The *call string* cs is defined as the sequence $(call_a, call_b, \ldots, call_z)$, with $call_i \in \{v_1, v_2, \ldots, v_n\} \wedge return_i \notin \{v_1, v_2, \ldots, v_n\}$. Intuitively, the call string of a path π is the number of pending calls in π.

Using the call-string approach on a given supergraph allows using the algorithms presented in Section 2.4.1 in order to solve a data-flow problem on an interprocedural control-flow graph. In the following, the terms control-flow graph and supergraph are used both to denote an interprocedural control-flow graph.

3

WCET Estimation

> To err is human but to really foul up requires a computer.
>
> *(Dan Rather)*

Embedded systems are the most used computer systems supporting our every day life. They can be found in mobile phones, multimedia players, cars, etc. In the area of safety-critical systems, human life often depends on a correct behavior of these systems. *Will my airbag save my life if I have an accident with my car?* is a crucial question. It depends on the correct behavior of the airbag controller detecting the crash timely and firing the right airbags. Another example for such a *hard real-time* system is the flight control computer of an airplane, which is responsible for the stability and attitude. All of these systems have one thing in common: they are subject to stringent timing constraints (beside their functional correctness) which are dictated by the surrounding physical environment.

A schedulability analysis needs to prove that all tasks involved in the decision process meet their physically dictated deadlines. As an input for the schedulability analysis, the runtimes for the tasks need to be known. In general, it is not possible to determine the runtime for programs due to the halting problem. But for embedded real-time applications, it is possible since they are using only a small programming subset which guarantees the termination of the application. Recursion is only allowed in cases where the recursion depth is known, and also for loops, the least and upper bounds of the iteration counts can be given.

Nevertheless, runtimes of a task vary depending on the actual input data, the initial hardware state, and the interference from the physical environment. Whereas the functional behavior of a task normally is only dependent on the

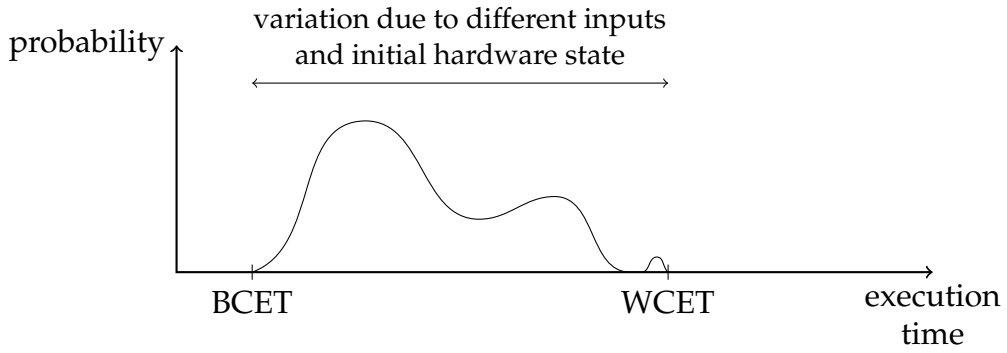


Figure 3.1 – Distribution of execution times for a task. The minimal and maximal times are the best-case (BCET) and worst-case execution times (WCET), respectively.

input, the timing behavior of a task is also dependent on the current state of the hardware, especially in the presence of caches, execution pipelines and other hardware features. Preemptions and interrupts do also influence the timing behavior of a task.

Figure 3.1 shows the distribution of execution times for a task. The minimal runtime for a task is known as the *best-case execution time (BCET)*, the maximal runtime is called the *worst-case execution time (WCET)*.

3.1 Methods

Determining the worst-case execution time for a task has been subject to research since the last three decades. An overview on the different approaches to tackle the problem of computing safe and precise WCET estimates is given in the following sections.

According to [WEE+08], existing approaches for the WCET estimation can be classified into two categories:

- *Measurement-based methods* utilize program executions and traces to obtain the WCET, and
- *Static methods* only consider the executable program for all possible concrete inputs.

3.1.1 Measurement-based Approaches

Measurement-based methods execute the task or parts of the task on the target hardware or a simulator for some set of inputs. The runtime of a task can be obtained by augmenting the program with additional code to use hardware timers or by using a logic analyzer and identifying the relevant signal changes directly. This leads to a distribution of measured times as shown in Figure 3.2. The method is not safe since in nearly every case, the program input leading to the WCET is not known. Running and measuring a program for every possible input will lead to the correct WCET, but is practically infeasible. Using a small input subset also does not guarantee a full path coverage of the analyzed task since it cannot be guaranteed that every path is triggered by the test vectors.

Furthermore, the method is quite expensive in terms of hardware and human resources needed since real-time computer systems are connected to their physical environment via a set of sensors and actuators. For the measurements, the sensor data has to be provided by external simulation hardware since running the tests on the concrete system might be too dangerous. In many of these systems, the actuators output directly influences the values of the sensors, which also has to be taken into account. Since the values of the sensors directly influence the program path taken, the whole system may behave different than in the concrete application domain.

Augmenting the task with additional code triggering timers or interrupts is also not allowed in some application domain, e.g., in avionics. Due to certification requirements for real-time systems, runtime guarantees must be given for the unmodified application, and on complex architectures, the additional code might also change a task's runtime behavior.

Another approach combines measurement with static methods. Only small pieces of code are measured, some safety margin is added and the results

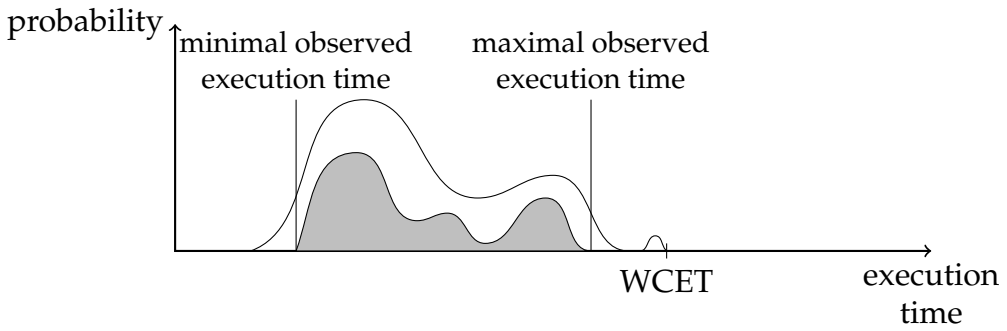


Figure 3.2 – Measured execution times for a task (filled area). The best and worst runtimes are not discovered.

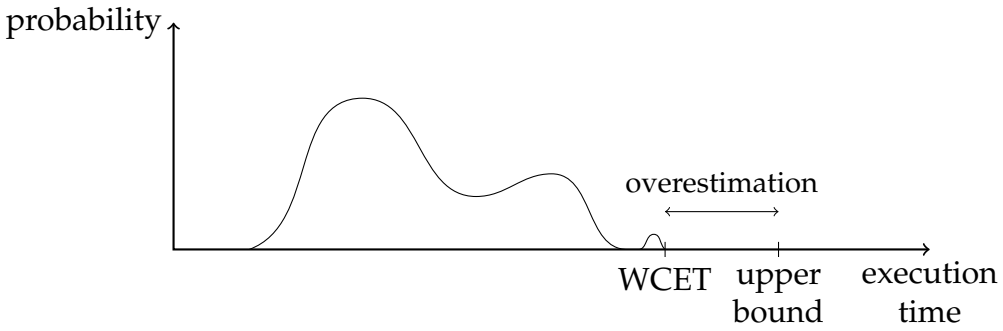


Figure 3.3 – Upper execution time bound. The overestimation is caused by uncertainties that cannot be excluded statically.

are statically combined according to the structure of the analyzed task. These approaches are called *hybrid approaches*. In contrast to purely measurement-based approaches, the path-coverage problem can be solved using these approaches. Furthermore, the measurement overhead can be reduced to a certain degree since program snippets often are reused in the program and have to be measured only once. However, in the presence of modern hardware (e.g., caches), guarantees may also not be given for this approach since the measured time for the pieces cannot be validated.

All measurement-based approaches share one large disadvantage: the surrounding environment and the actual hardware must be available. Especially in the early development stages, this might not always be possible.

3.1.2 Static Approaches

In contrast to measurement-based methods, static approaches do not rely on executing the program code on the real hardware. They take the program and combine it with some theoretical model in order to obtain a WCET estimate. All static methods do only compute *upper bounds* on the execution time of a task leading to a WCET overestimation (cf. Figure 3.3). The overestimation itself depends on uncertainties (e.g., on the input or about the hardware state) that cannot be excluded statically.

One of the first approaches being applicable for a wide range of programming languages and target architectures was presented by [Sha89] and makes use of so-called *timing schemes*. The method is based on the assumption that the runtime for atomic operations can be determined. Table 3.1 on the next page shows a small example program and the corresponding times for the atomic operations. The worst-case execution time can be calculated by summing up the time for evaluating the condition of the `if`-statement, adding the maximum

Program	Timing
`while (x < 4)`	1
`  if (x > 1)`	1
`    r = r * x;`	4
`  else`	
`    r = 1;`	1
`  x++;`	2

Table 3.1 – Simple program with execution times for atomic operations.

of the execution times of the branches – 4 in this example – and by adding the time for the iterator increment. For loops, iteration bounds can be used. To compute the WCET, the maximal iteration count has to be multiplied with the execution time for the body plus the time for evaluating the condition. If the condition of a loop is checked at the start, the condition will be executed iteration count plus one times, which has also to be respected. For the above example, assuming an input greater or equal to 0, this leads to a worst-case execution time of $4 \times (1 + 4 + 2) + 5 \times 1 = 33$.

This method produces safe WCET estimates (assuming safe execution times for the atomic instructions and correct loop bounds), but the precision may be very bad. Even in the small example given, the WCET bound is overestimated since the execution of the loop body is much faster for some values of x.

Another drawback of this approach is the assumption that the worst-case execution times for the atomic instructions are invariant. Thus, the WCET estimate is only safe, if the *local* worst-case execution time of an instruction does not depend on the *execution history*. But especially on newer architectures using deep pipelines, parallel execution units and caches, this is no longer the case. Thus, the *unit-time* (executing an instruction always takes exactly one unit of time) or *constant-time* abstractions used in many approaches are rendered obsolete.

A similar approach by [LMW96] models the flow through a program as an *integer linear program*. This approach also assumes fixed execution times for atomic instructions. The WCET bound is then computed by maximizing the objective function defining the execution time.

A different static method is *simulation* using models of the hardware which are usually provided by the manufacturer. These simulators are often available for architectures that are difficult to debug, so simulation might ease the development process on these architectures. Unfortunately, these models are sometimes not cycle-accurate. [SP09] could also show some underestimation for the *simG4 simulator* [Fre06] for the Freescale PowerPC MPC7448 processor

[Fre05a]. For simulation, the same restrictions regarding the input as mentioned in Section 3.1.1 on page 29 for measurement-based approaches hold. Additionally, simulation is much slower than executing the program on the real hardware. [Bor09] state that nowadays simulation models are a million times slower than the real hardware.

The most promising approach for the computation of safe upper bounds on the WCET of a program is the use of *data-flow analysis* (cf. Section 2.4). The technique is well known in the area of compiler construction [WM95] to derive properties of a program holding for all executions. For instance, data-flow analysis is used to obtain values of variables – the result of the analysis may then be used for a constant propagation. So, data-flow analysis can be used to compute safety properties holding for all executions of a program. Thereby, concrete values can be *abstracted*, and computations are made on these abstracted values.

The theory underlying this approach is *abstract interpretation* (cf. Section 2.3 on page 14), a framework that eases correctness proofs of the results. Figure 3.4 on the facing page roughly shows how the worst-case execution time can be estimated using the data-flow analysis approach: The domain of the data-flow analysis is an abstracted processor state containing all information necessary for timing, cache contents, etc. of the complete system. Basing on the basic block structure of the program to be analyzed, a set of abstracted states can be cycle-wise simulated using an abstract processor model for each instruction in the basic block. The resulting set of abstract states is then propagated to each successor basic block. In contrast to traditional simulation, this method does not have the drawback of unknown input since computation is made on abstract values. The method can be viewed as an *abstract simulation*, each abstract state thereby represents a set of concrete states of the target system.

Today, a variety of tools for worst-case execution time estimation exists. A short overview over these tools, their functionality and also their limitations will be given in the next section.

3.1.3 Related Tools

As stated earlier, determining worst-case execution time bounds has been subject of research since the last three decades. As a result, a variety of tools for WCET estimation has been developed. Nowadays existing tools, their functionality, restrictions and limitations will be described in this section.

Most tools are able to analyze binary code since compiler optimizations make it difficult to argue about the timing behavior of high-level source programs or specifications [TSH+03]. Additionally, standards released by certification

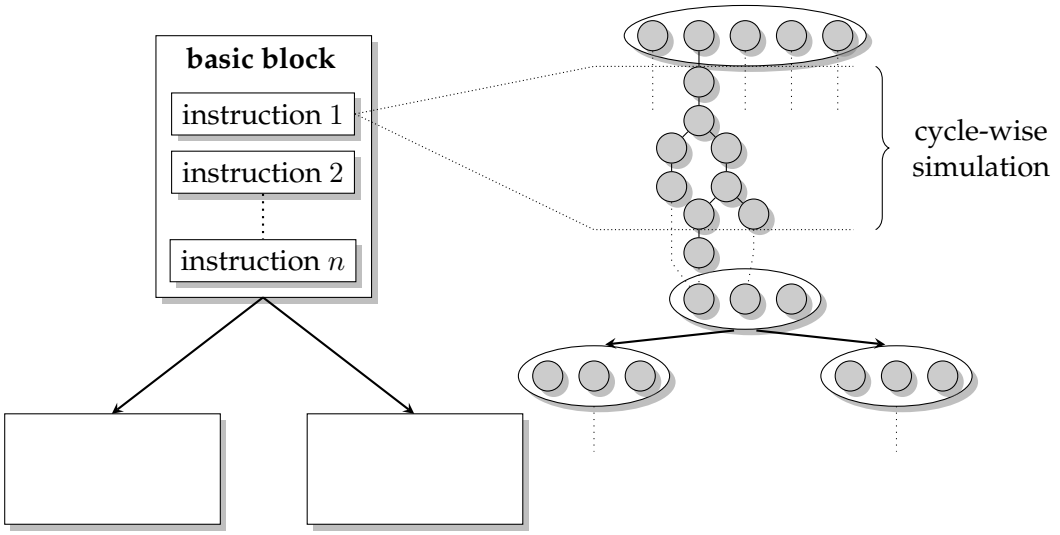


Figure 3.4 – WCET estimation using abstract simulation.

authorities, e.g., DO-178B [Spe92], require that validation has to be performed on the unmodified code that actually runs on the real-time system[1].

The most prominent and successful tools in the area of measurement-based methods are RapiTime [BBN05] and a research prototype from the TU Vienna [PN98]. Both tools rely on input vectors triggering the worst-case execution path to be executed during measurement. As stated earlier, this is not always possible, so test cases can be generated to achieve basic block coverage. Combining the basic block timings to a worst-case execution path is then in the responsibility of the user.

The field of static WCET estimation tools is much more widespread. The most powerful and industrially usable tool is *aiT*, developed at Saarland University and AbsInt Angewandte Informatik GmbH. More details on this tool can be found in Section 3.2, an industrial evaluation of the tool on avionics control software can be found in [TSH+03]. Besides the *aiT* tool, different other static WCET estimation tools exist. In contrast to the *aiT* tool, these tools do not support complex processor architectures with hardware features like speculation, out-of-order execution, etc. More details on these hardware features can be found in Chapter 4 on page 41.

The Bound-T tool of Tidorum [HLS00] was originally developed at Space Systems Finland Ltd under contract with the European Space Agency and was used for the verification of spacecraft on-board software. Nowadays, Tidorum Ltd is extending the tool to other application domains. The tool is able to produce safe upper bounds on the execution time of a subroutine, including all called

[1] "Fly what you test and test what you fly"

functions. Unfortunately, the tool is limited to a set of non-complex architectures, and currently, the tool is not able to cope with caches.

Another tool is SWEET [Erm03] (for SWEdish Execution Time), which was developed by the Mälardalen University, C-Lab in Paderborn and Uppsala University. For now, development has been fully moved to Mälardalen University. The underlying tool architecture is held modular, allowing the different parts to act rather independently. WCET analysis with SWEET consists of three major phases: a flow analysis reconstructing the program flow of the task to be analyzed, a processor behavior analysis and an estimate calculation. Estimates can be computed using three different approaches: a *fast path-based* technique, a *global IPET* and a hybrid *clustered* technique. SWEETs behavioral analysis is limited to in-order pipelines with no long-term effects like timing anomalies (cf. Section 4.4 on page 60).

Only a subset of ANSI-C is supported by the research prototype from the TU Vienna [KLFP02]. The tool cooperates with a C compiler translating programs written in WCETC to object code and producing additional information used by the WCET tool.

The research prototype from Chalmers University of Technology [LS99b] is currently limited to a subset of the PowerPC instruction set architecture [Fre05b]. This tool uses an extended version of an instruction set simulator capable to handle unknown input data.

Quality of service (QoS) guarantees for hard real-time systems are the point of interest that led to the development of the research prototype from Florida State University [AMWH94]. For this, a modified compiler has to be used producing additional information like loop bounds, control flow, and instruction characteristics. As a result, the tool produces a lower and an upper estimate for the runtime of a task.

Heptane [CP00] combines two WCET computation methods in one tool. Besides the timing scheme-based method which quickly produces a, under some circumstances, rather overestimated upper bound, a computation method based on integer linear programming can be used to tighten the WCET estimate. In contrast to the *aiT* tool, no automatic flow analysis is performed to identify mutually exclusive paths.

In contrast to all other tools, Chronos [LLMR07] is an open-source static WCET analysis tool supporting out-of-order execution and branch prediction. The functionality of the tool is rather similar to the *aiT* tool. Unfortunately, the tool currently does not support data caches, which are widely used in today's embedded systems.

Besides the classical measurement-based and static methods, a variety of tools exists combining measurement with static path analysis methods. SymTA/P

(for Symbolic Timing Analysis for Processes) [WEY01, WKE02] combines a platform independent path analysis on source code level and platform dependent measurement on object code level. The execution times can be obtained by simulation using a cycle-accurate processor model or by using an evaluation board. A similar approach can be seen in the hybrid research prototype from the TU Vienna [WKRP05, WRKP05]. The key feature of this tool is the automatic segmentation of the program code into segments of reasonable size, and the automatic generation of test cases used to measure the execution times of all paths within each program segment. The tool was especially designed for the analysis of automatically generated code, e.g., code generated from Matlab/Simulink models.

More details on available tools, supported hardware architectures, limitations and the underlying analysis techniques can be found in [WEE+08]. In the following section, the *aiT* framework for WCET analysis will be described in more detail.

3.2 *aiT* WCET-Analyzer Framework

An overview of the *aiT* WCET analyzer is shown in Figure 3.5 on the next page, details can be found in [FKL+99, FHL+01]. Inputs to the *aiT* tool are the executable to be analyzed, additional user annotations, a description of the memories and buses used in the target system (i.e. a list of minimal and maximal access latencies for each memory region), the processor's cache configuration, flow constraints and a task (identified via its start address). Effects of interrupts, IO and co-processors are not reflected in the predicted runtime estimate and have to be considered separately (e.g., by a quantitative analysis). *aiT* consists of different tools that can be influenced by several configuration files. The main data is shared via the separated analyses in a human-readable representation that was designed to simplify analyses and optimizations. This intermediate representation is called CRL2 and can be augmented with attributes [Lan98]. The different phases are now described in more detail.

First, *control-flow reconstruction* (also called decoding phase) has to be performed. *aiT* analyzes the binary of the task since analyzing C-code or assembly code is not sufficient for certain certification requirements: a compiler might change the structure of a program making the use of source code for timing consideration impossible. Furthermore, several analyses within *aiT* require the use of binary code since source code does not offer information about the use of registers and absolute addresses of data accesses. The latter is of particular importance for the cache analysis and for memory accesses since different regions may have different access timings. As a result, only the binary level offers sufficient information for safe WCET estimates.

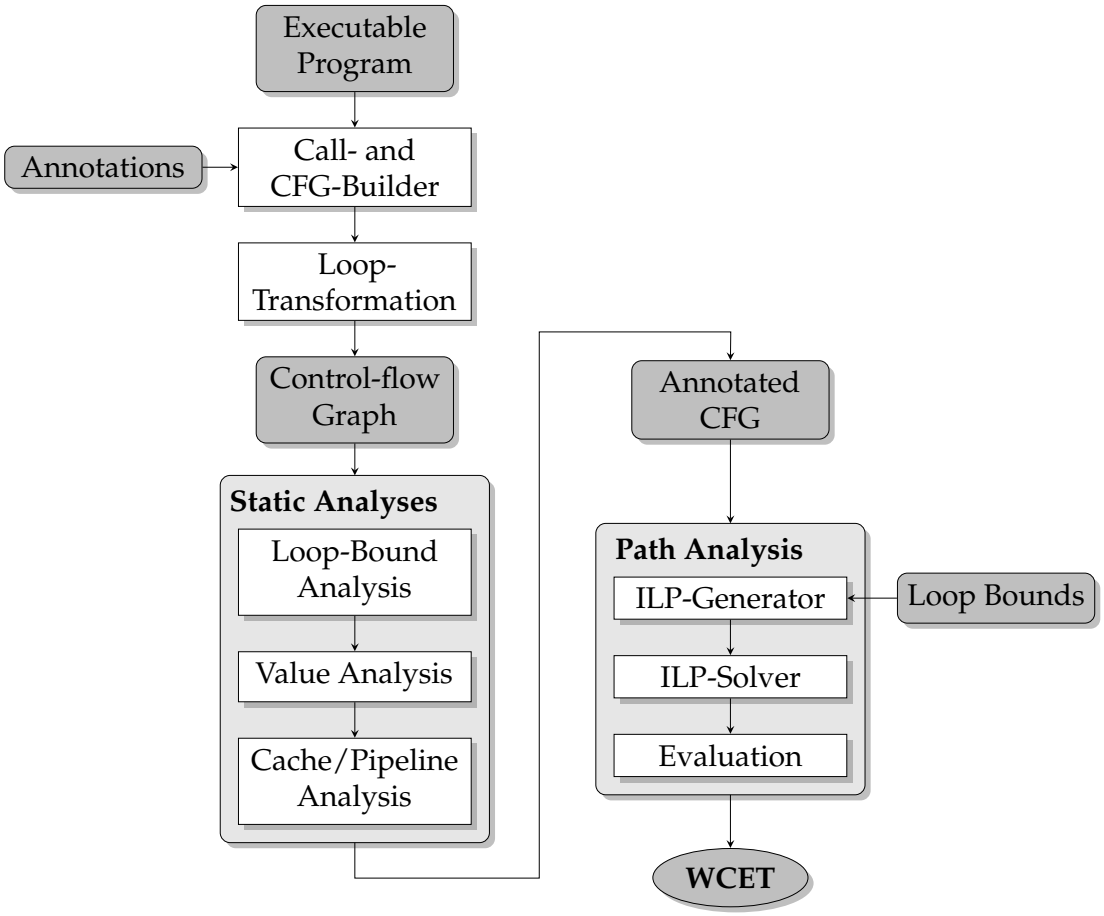


Figure 3.5 – Schematic overview of the *aiT* WCET-analyzer framework.

The control-flow reconstruction uses a bottom-up approach. Details can be found in [The00]. The reconstructed control-flow graph must be a sound approximation, i.e. it must cover all possible execution paths. Problems may arise when function pointers are in use. Thus, the user can aid the decoder by providing annotations about the compiler used, call targets and targets of dynamically computed branches. The result of the reconstruction is stored in the CRL2 intermediate format, where the instructions are augmented with attributes that further classify them (e.g., registers used and modified, instruction class information, etc.).

The original control-flow graph obtained from the decoder phase is modified by the so-called *loop transformation* (cf. [Mar99]), which turns loops into separate routines that call themselves recursively. This is done in order to enhance the precision of the timing analysis in the presence of loops (allowing the usage of common context separation techniques). Please note that loop transformation is only performed on the reconstructed control-flow graph, the executable itself

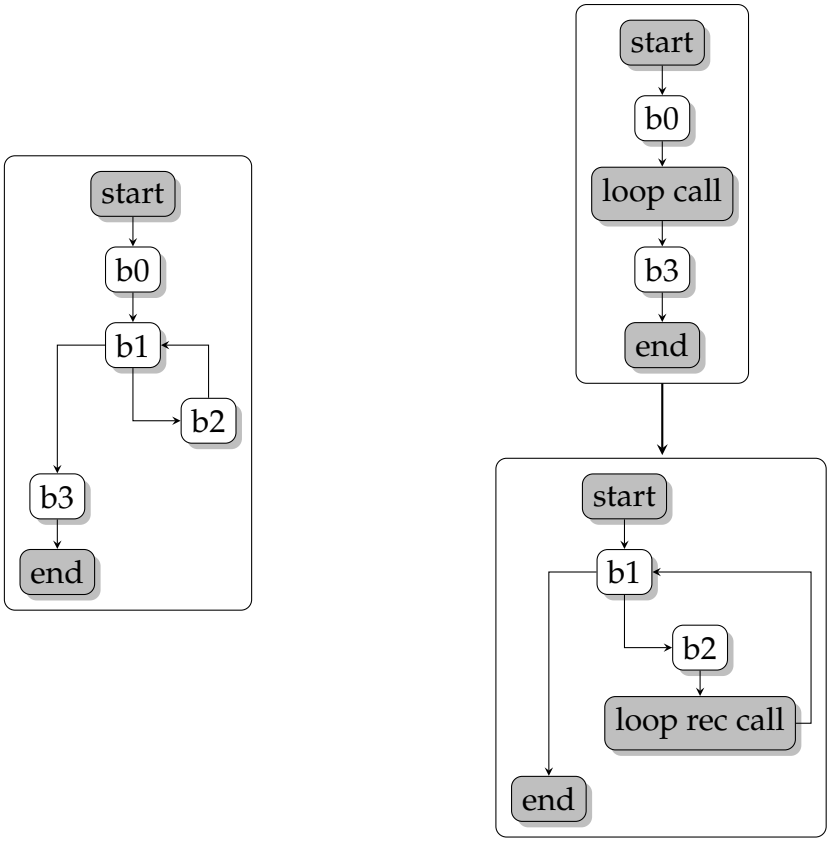


Figure 3.6 – Loop transformation – unmodified control-flow graph on the left, transformed control-flow graph on the right side.

remains unmodified. The effect of loop transformation on a simple loop is shown in Figure 3.6. The graph on the left shows an example containing such a simple loop. The entire routine is entered at the special node `start` and left at the special node `end`. The picture on the right hand side of Figure 3.6 shows the result of the loop transformation. The loop has been turned into a separate *loop routine*. The calling relationship is indicated via an arrow between the two routines. As a result of the loop transformation phase, iteration has been replaced by recursion. Loop transformation is not applicable to loops with multiple entries. The loops remain unchanged. Please note that there might also exist a variety of loops that cannot be transformed by loop transformation, especially loops with multiple entries. The loop-transformed control-flow graph serves as input for all further analyses within the *aiT* toolchain.

In the third phase, the *loop-bound analysis* tries to automatically find the upper bounds for loop iterations. Only for a small amount of loops, the user must provide this bound via an external annotation. The analysis is implemented via a data-flow analysis [NNH99].

The *value analysis* phase uses a data-flow analysis to determine safe intervals for data accesses to the memory performed by the program. These intervals are safe over-approximations for the possible addresses of each access. Since instruction accesses (e.g., prefetches) are precisely known and fixed by the structure of the control-flow graph, it is sufficient to compute the access intervals for data accesses. The analysis is an abstract interpretation of the machine set semantics on an interval domain. It was first developed by [Sic97]. The underlying principles of interval analysis can be found in [CH78, WW08]. The result of this phase is context-sensitive intervals for all instructions accessing the memory. Register contents are also modeled within the analysis. Thus, the outcome of branches can often be determined by this phase. So, the not-taken part in a dedicated context can often be marked as infeasible and thus, will be no longer investigated by subsequent analyses. The quality of the results computed by this phase can be seen in [FHL+01]. More details about the functionality can be found in [Abs02]. Recent developments concerning the structure of the *aiT* framework have led to a join of the loop-bound analysis and the value analysis.

The combined *cache/pipeline analysis* is the heart of the WCET analyzer and is also called micro-architectural analysis. It models the cycle-wise evolution of abstract processor states and determines maximal timings for each basic block in the control-flow graph. This phase is important for the purpose of this thesis and will be described in more detail in the next section.

The *path analysis* generates an *integer linear program* (ILP) from the control-flow graph of the program and the results of the combined cache/pipeline analysis. The objective function is the execution time of the program, which is to be maximized. Various configuration parameters, e.g., flow constraints, can be used to model special relationships between several basic blocks. More details on the ILP generation can be found in [The00].

The last steps in the *aiT* toolchain then solve the generated ILP by using an *ILP solver*, map the result back to a path in the control-flow graph and visualize it. The path shown is one possible WCET path; there might exist more than one path in the control-flow graph leading to identical worst-case executions. Additionally, this step computes WCET contributions for each basic block and routine contained in the control-flow graph.

The whole *aiT* toolchain is controlled via an interactive driver allowing the user to control several parameters and providing feedback. Novel developments like a refined loop analysis and a refined infeasible-path detection are described in [FMC+07]. A new path analysis on *prediction files* – a graph-representation of the evolution of abstract pipeline states – is described in [Ste10].

3.3 Micro-architectural Analysis

As stated earlier, the micro-architectural analysis or cache/pipeline analysis is the heart of the *aiT* WCET analyzer.

The *cache analysis* classifies the accesses to main memory. Within *aiT*, the analysis is based on [FW99], which handles analysis of caches with least recently used (LRU) replacement strategy. However, it had to be modified to reflect the non-LRU replacement strategies of common microprocessors: the pseudo-round-robin replacement policy of the ColdFire MCF5307, and the PLRU (Pseudo-LRU) strategy of the Freescale PowerPC MPC750 and MPC755. The modified algorithms distinguish between sure cache hits and unclassified accesses. The deviation from perfect LRU is the reason for the reduced predictability of the cache contents in case of Motorola ColdFire MCF5307 and Freescale PowerPC MPC750/755 compared to processors with perfect LRU caches [HLTW03]. A more detailed discussion on caches and the influence of the replacement strategy on cache predictability is given in [Rei08, WGR+09].

The *pipeline analysis* models the pipeline behavior of the target architecture to determine execution times for a sequential flow (i.e. a basic block) of instructions (cf. Figure 3.4 on page 33). It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and classification of memory references as cache hits or misses. The result is an execution time for each basic block in each distinguished execution context.

Since *aiT* follows a static analysis approach, also the pipeline analysis has to deal with missing or unknown information. That is, for example, missing information on the input of the task to be analyzed, or the outcome of some branches. Also information computed by the value analysis might be sometimes imprecise, or, in the worst case, totally unknown. Thus, the target region of a memory access, and also the fact if the requested information is already contained in the cache cannot be deterministically answered. So, micro-architectural analysis has to deal with *non-determinism* and must follow all possibilities in order to compute safe estimates for the execution time of a sequential instruction block. In contrast, execution of a task on the target architecture always is deterministic. This is shown in Figure 3.7 on the following page. Consider the case shown on the left: execution of the `then`-branch would take much more time then executing the `else`-case. Running the application on the target system would show exactly one execution path, but in static analysis, the outcome of the condition statement at the beginning often cannot be determined, so micro-architectural analysis has to split the state(s) propagated along the control-flow graph and must follow both possibilities.

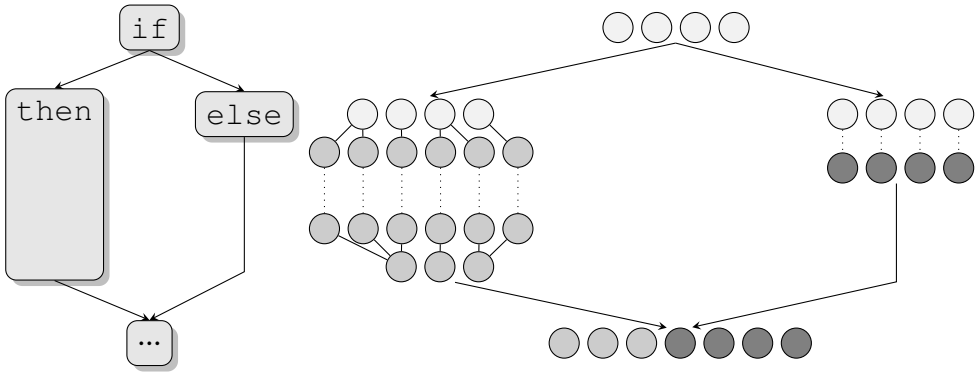


Figure 3.7 – Sample program structure – the height of a block indicates its instruction count.

Moreover, not only missing input may lead to these *pipeline splits*, also abstractions used to make micro-architectural analysis feasible may introduce uncertainties that result in splits. Thus, in contrast to normal simulation or execution on the real hardware, which results in an *execution trace* of a program, micro-architectural analysis has to deal with an *execution tree*.

The pipeline analysis within *aiT* relies on a timing model that is currently *handcrafted* by human experts [The04, The06, FMC+07] using mainly the processor documentation as input. Thus, timing model creation and implementation are time-consuming and also error-prone processes. Also correctness of the documentation used cannot be guaranteed, thus a rather complex and sophisticated process of *timing model validation* has to be performed afterwards. As of today, this is done by comparing the timing model with hardware traces taken from programs, for which the worst-case input is already known. Interpretation of possible discrepancies between the micro-architectural analysis and the hardware trace often offers a multiplicity of explanations and thus is not easy.

A more reliable and complete source of information about the timing behavior of the processor pipeline is given by formal processor descriptions in a *hardware description language* (HDL). In [The06], general ideas on how to manually derive a timing model from these formal descriptions are described, at least for small components.

Modern architectures offer a variety of hardware features coupled with deep processor pipelines and out-of-order execution making timing model derivation in the old fashioned way more and more complex. The contribution of the thesis is the automation of some analysis steps required to (semi-) automatically derive a timing model for a whole processor given in a hardware description language. An overview over current embedded hardware development and its formal description will be presented in Chapter 4.

4

Modern Processor Development

> What is chiefly needed is skill rather than machinery. . . . It is possible to fly without motors, but not without knowledge and skill.
>
> *(Wilbur Wright)*

The need for more computing power is increasing rapidly, resulting in an increasing demand for faster computer systems. The microprocessors used in those systems are traditionally classified according to the design of their instruction set. According to [HPG03], they can be categorized as *complex instruction set computers* (CISC), *reduced instruction set computers* (RISC), and *very long instruction word architectures* (VLIW). Their characteristics according to [WM95] are as follows:

- CISC architectures are designed to close the *semantic gap* between high-level programming languages and machine languages. These architectures are characterized by:
  - a large number of different and complex addressing modes to provide efficient access to different data structures,
  - diverse versions of operations for different operand length and combinations of different sorts of operands,
  - only a few processor registers,

  - a huge variety of execution times for instructions (depending on the concrete instance),
  - and a microprogrammed control logic.

- The design goal of RISC architectures focuses on increasing the execution speed of machine instructions by simplifying them. The main characteristics of these architectures are:
  - the restriction of memory accesses to dedicated load/store instructions,
  - the ability of executing more than one machine operation per clock cycle,
  - few addressing modes,
  - and a hard-wired control logic.

- VLIW architectures are explicitly designed to provide statically determined instruction level parallelism. Thereby, a fixed number of operations can be composed to form a VLIW instruction. Execution of the operation is then started in parallel. The arrangement of operations to exploit a high degree of parallelism is the task of the compiler.

Nowadays, classifying microprocessors according to this traditional scheme becomes more and more difficult. Historically seen, the need for computing power in articles of daily use has led to the development of RISC architectures. The increase in computing power compared to traditional CISC architectures then has also influenced the development of other architectures.

As of today, the market for embedded systems and also the demand for more computing power are still increasing. The biggest bottleneck for system performance still remains the connection of the processor core(s) to the main memory. Thus, modern processors offer a variety of features for speeding up program execution. The key features are described in the next section. Other components that are paired with the processor core to form the whole system are described in Section 4.2 on page 52. Section 4.3 on page 56 is concerned with the predictability of modern hardware architectures. Section 4.4 on page 60 details different classes of timing anomalies decreasing a system's predictability. Due to the complexity of modern processors, formal hardware description languages are used for development. This is described in Section 4.5 on page 62.

4.1 Processor Cores

The demand for more performance of the cores has led to the development of sophisticated features like caches, pipelining, or branch prediction. Modern architectures spend most of the time waiting for slow main memory, which is mainly a cost factor. Currently, a *memory hierarchy* is introduced placing a small, but fast cache between the CPU and the main memory. Basing on the fact that normal programs exhibit a high degree of temporal and spatial locality, the use of caches improves the overall performance dramatically. Nowadays, usage of two or three level of cache hierarchy becomes more and more common.

In contrast to caches aiming at reducing the latency of memory accesses, deep processor pipelines are used to make use of high parallelism in the machine code. Execution of an instruction can be split into different phases: first, it has to be *fetched* from the memory, then it is to be *dispatched* to its execution unit. While in *execution stage*, the instruction competes for resources. After having finished its execution, results are *written back*. Having passed this stage, all changes to operands, etc. are part of the architectural state, i.e. the instruction itself does not influence the behavior of the processor anymore. Ideally, an instruction resides exactly one processor cycle in one stage. Since the different stages of the processor pipeline can execute in parallel, this allows a maximal performance of one instruction per clock cycle.

Caches as well as pipelines increase the *average case* performance of a system (cf. Figure 3.1 on page 28), but there are also some programs that exhibit a bad runtime. Both, caches and pipelines are *history-sensitive*, i.e. their behavior strongly depends on instructions executed before. E.g., whether an instruction fetch results in a cache hit or a miss depends on the instructions fetched before. The use of pipelines and caches combined with their history-sensitivity make it impossible to easily decide, how long an instruction resides in one of the previously mentioned pipeline stages and thus renders timing scheme-based WCET estimation methods inadequate. An overview on caches and pipelines as well as their functionality and corner cases is given in the next sections.

4.1.1 Caches

A *cache* is a small piece of fast memory that is placed between the connection of the processor core and the main memory to bridge the gap in performance. There can exist different levels of caches that are hierarchical organized. The fastest one is the closest to the processor and is called level-1 cache (or L1 cache for short), the next one level-2 cache, and so on. For a memory access, whose data is not found in the cache Ln, the L$n+1$ cache is asked for the corresponding data. If there is no next cache, data is loaded from the main memory. The

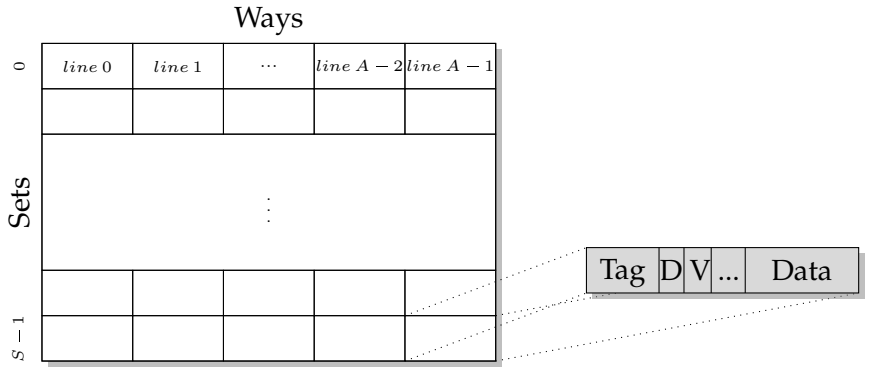


Figure 4.1 – Cache architecture.

architecture of a cache is depicted in Figure 4.1. Depending on the architecture, there can be separated caches for instructions and data, but also unified cache architectures are widely used. Also mixed cache hierarchies (e.g., separated L1 caches and unified L2 cache) are common.

Since caches are used to bridge the gap between processor core performance and memory performance, caches uses a memory type, which is much faster but also more costly than the memory type that shall be cached. In order to solve the problem, which data shall be available in the cache, locality attributes are used:

- *Temporal locality*: Programs often access data repeatedly (e.g., when executing a loop).Thus, it is likely that data already accessed by the processor will be accessed again in the near future. This "new" data shall reside in the cache, whereas "older" data shall be removed from the cache/replaced by other data. This removal/replacement is called *eviction*.

- *Spatial locality*: Program code and data are not arbitrarily placed in the address space, moreover, they reside in special address areas (e.g., code segment, data segment, stack segment, etc.). Thus, it is likely that after an access to a dedicated address there will be an access in the neighborhood of this address (i.e. the delta of both addresses is very small). This assumption is valid for instructions, which are located consecutively in the memory, as well as for array data structures.

Due to spatial locality, not only single bytes are placed into the cache. Instead, the cache collects data of a complete address area. This area is called *cache line* or *cache block*. A cache consists of S sets, each *set* consists of A ways. Each *way* stores the contents of one cache line. A is called the *associativity* of the cache. If $S = 1$, the cache is called *fully associative*, otherwise, it is called A*-way set associative*; if $A = 1$, the cache is called *direct mapped*. Each cache line consists of

data obtained from the memory, a *tag* - the high order address bits of the address - that is used for identifying the line, and some *status bits*. Besides others, the V-bit is used to indicate that the cache line contains valid data, and, for data caches, the dirty bit (D) marks cache lines that have been modified and have not yet been written back to main memory. In case of multi-core systems, additional flags for coherency, etc. are available.

A cache can be characterized by the following parameters: its *capacity* C, the size of a cache line L, its associativity A, and its number of sets S. Thus, the capacity of a cache is $C = S \cdot A \cdot L$, where the line size L and the number of sets S is always a power of 2. The same usually holds for the associativity A as well.[1]

Another important characteristics of a cache is its *replacement policy*, which is responsible for selecting the line to be evicted from the cache, if it is full and further data is to be accessed. The location, where newly accessed data is to be placed, is determined by using the address a of the access. The index i of the set, data is placed in, is computed by $i = (a/L) \ mod \ S$.

For instruction fetches and load instructions (read accesses), the upper address bits are checked in parallel against the tags stored in the ways of the set i (ignoring those ways that are not valid). If a match is found, the access *hits* the cache and the required data is returned to the processor. Otherwise, the data must be loaded from memory, which is called a *miss*. It is the task of the replacement strategy to determine the place in the set, where the data shall be stored in. Usually, invalid ways will be filled first. If there are no invalid lines in the selected cache set i, the line to be replaced is selected by the replacement strategy. The new data is placed into the cache, the tag bits are updated with the address of the new access, the valid bit is set and the dirty bit is cleared. Then, the processor starts to fetch the data of the new cache line from the memory hierarchy (next cache level or main memory). Due to the size of a cache line, it normally requires more than one transaction to fill the whole line. If the data word of the referenced line is returned first, this is called *critical word first*, the line fetch wraps around at the end of the line. Otherwise, the line is filled from its beginning. A cache that can further process accesses during a cache line fill hitting the cache is called *hit-under-miss* capable. A further cache miss will lead to a stall in the cache architecture. More advanced cache architectures are also able to process more than one outstanding cache miss. Those architectures are called *miss-under-miss* capable.[2]

For store instructions, cache behavior can be configured. If the data that is modified by the store operation is directly issued to the main memory, and, if available in cache, also modified in place, the cache is said to be *write-through*. Each modification is made directly visible in the main memory. If the data is

[1] One exception is the instruction cache of the SuperSPARC architecture, where $A = 5$.
[2] Miss-under-miss caches also stall, when there are too many outstanding misses.

only modified in the cache, the cache is said to be *write-back*. In this case, the dirty flag is set, indicating that the cache line was modified, but the modified data is not yet visible in main memory. Thus, data must be flushed to main memory to obtain a consistent state. This could be achieved by explicit flush instructions, or by triggering a write flush, when the cache line is to be replaced. If a line that is to be modified is not in the cache, the line must be fetched from memory in order to modify it (*write-allocate*), or the data is just passed to the main memory. Normally, write-back implies write-allocation.

Most cache architectures allow to *lock* the cache, or at least parts of the cache. No replacements for these parts will occur, but invalid lines will be filled until no empty lines exist. Normally, this feature is used for critical code or data, or the stack area is locked increasing the performance of the application.

Within the embedded area, it is also possible to mark memory regions as not cached. Accesses to these regions completely bypass the cache and do not modify the state of it. All accesses to these regions are directly served by the main memory over the external bus.

As stated before, the replacement policy of a cache is an important characteristics. One replacement strategy that is often implemented is the *least recently used* strategy (LRU). Thereby, the line in a set that has been unreferenced for the longest time is the one being replaced next. The sets of the cache are independent, i.e. counting of referenced lines is done separately for each set. Therefor, each line in a set is assigned an age, so for each set A ages must be maintained. The line being referenced last gets the youngest age, the line being unreferenced for the longest time gets the oldest age. Each access of the set, no matter if hit or miss, updates the ages of all lines in the set. If the access is a cache hit, the referenced line becomes the youngest age (age 0), and all lines younger than the referenced one age by 1. In case of a cache miss, the newly loaded line becomes the youngest, the oldest line will be evicted from the cache and all other lines age by 1. For a 4-way set-associative cache, line aging is depicted in Figure 4.2 on the next page. On top, line aging in case of a hit is shown; below, aging in case of a miss is shown.

Unfortunately, storing the age of each cache line per set is quite expensive leading to the development of another strategy called *pseudo least recently used* (PLRU)[3]. This strategy is used in the caches of the Intel 486 and in many processors in the Power Architecture (formerly PowerPC) family, such as Freescale's PowerPC MPC7448. In contrast of storing the age of each line, PLRU requires only $A - 1$ bits per set to build up a binary search tree. Each node of the tree has an one-bit flag denoting "go left" or "go right" to find an element for replacement. To update the tree, it has to be traversed and each node flag has to be set to the direction denoting in the direction that is opposite to the direction taken.

[3] also known as Tree-LRU

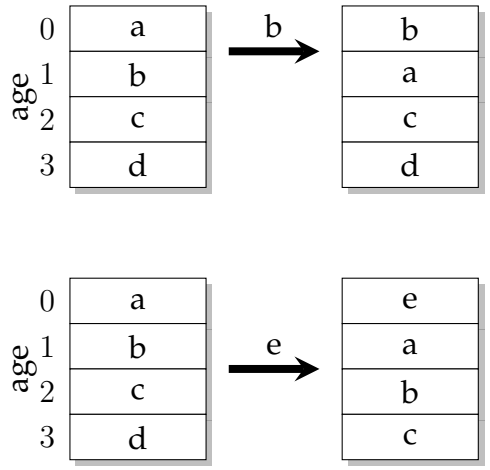


Figure 4.2 – Cache update under LRU replacement policy.

For a 8-way set-associative cache, this is shown in Figure 4.3 on the following page.

4.1.2 Pipelines

The process of executing an instruction can be split into several disjoint phases. E.g., the pipeline of the simple DLX machine [HPG03, MP00] can be split into:

- instruction *fetching* from memory,
- instruction *decoding*,
- *execution* of the instruction,
- *memory access* of load and store instructions, and
- *write-back* of results into registers.

Each of these phases is called a *pipeline stage*. Instead of waiting for an instruction to finish all sub-operations (which is also called instruction *retirement*) before a subsequent instruction can be started (cf. Figure 4.4 on the next page), the idea behind pipelining is to overlap the execution of different pipeline stages of subsequent instructions. If there are no dependencies between the several stages and assuming no dependencies between the instructions, a perfect pipelining of instructions as shown in Figure 4.5 on page 49 can be achieved, assuming a 1 cycle latency for each instruction in each stage. Once filled, the CPU is thus able to retire one instruction per clock cycle.

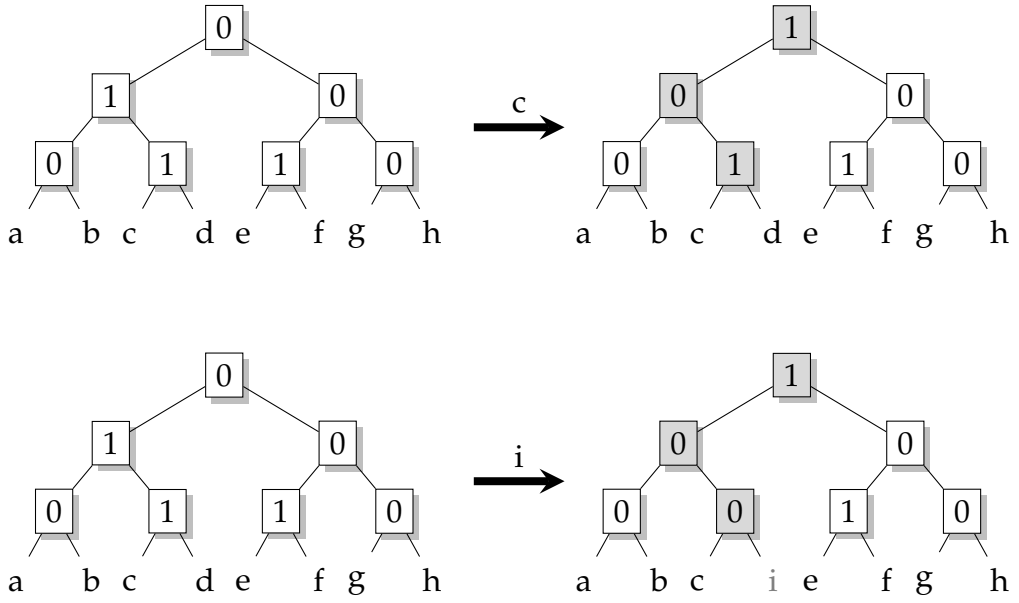


Figure 4.3 – Cache update under PLRU replacement strategy.

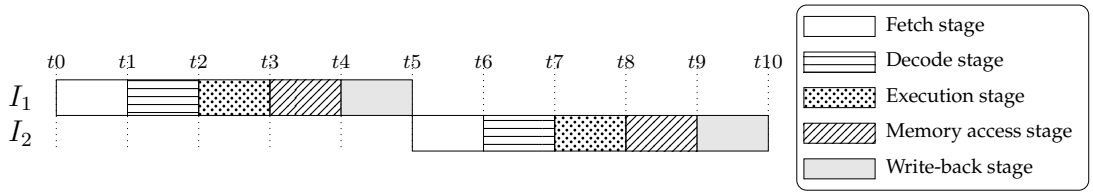


Figure 4.4 – Sequential instruction execution.

Pipeline Hazards

Pipelines massively exploit the parallelism that is inherent in programs. In practice, a retirement of one instruction per clock cycle would never be achieved due to dependencies between several instructions and other conflicts. The situations preventing the next instruction from executing in a certain pipeline stage is called *hazards*. Hazards thus reduce the performance that could be theoretically achieved by ideal pipelining. Moreover, hazards can make it necessary to *stall* certain pipeline stages or even the whole pipeline. Stalling a certain stage inserts *bubbles* into the pipeline, i.e. the stage remains empty and no work will be performed in the next clock cycle.

Hazards can be classified into three categories:

- structural hazards,

- data hazards, and
- control hazards.

Pipelining overlaps the execution of instructions. This overlapping requires pipelining of functional units and also duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination is not possible due to resource conflicts, this is called a *structural hazard*. E.g., the memory bus of the DLX machine mentioned before is a resource, which is available only once. The bus is used by the memory access stage to process load and store operations as well as by the instruction fetch stage acquiring new instructions from the memory. So, if both stages want to simultaneously transfer data over the bus, the fetch stage has to be stalled until the memory fetch stage has completed. A bubble must be inserted after the instruction fetch stage.

Data hazards arise from data dependencies between operands of subsequent instructions and are the most common ones. Listing 4.1 on the next page shows a small example code sequence. The second instruction requires the result of the first one in form of operand `r5`. Thus, the second instruction cannot start execution until the first one has written back the result into the register file (which happens in the write-back stage). In general, one can distinguish between three kinds of data hazards:

- *Read after write* hazards occur, if a subsequent instruction reads a register that is written by the previous instruction (cf. the first two instructions in Listing 4.1 on the following page). To speed up the pipelines and to reduce the stall time, *forwarding* can be used.
- *Write after read* hazards occur, whenever an instruction writes a register that is read by a previous one. In this case, it has to be guaranteed that the write occurs after the read. For the DLX machine, this can be guaranteed due to the in-order execution paradigm. Other more sophisticated architectures like the Freescale PowerPC MPC7448 support out-of-order

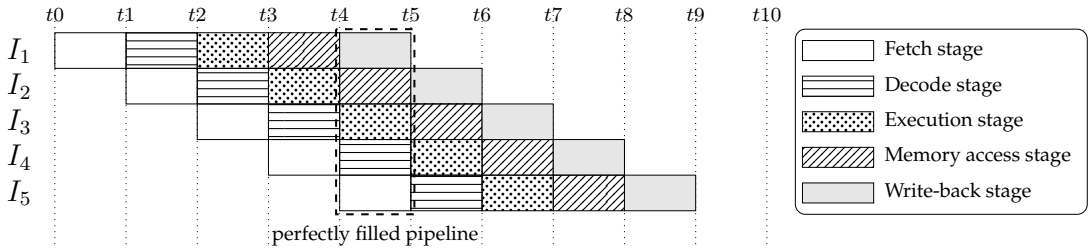


Figure 4.5 – Pipelined instruction execution.

```
add r5, r6, r7;
sub r2, r8, r5;
add r6, r7, r7;
add r8, r6, r6;
add r6, r5, r5;
```

Listing 4.1 – Data dependencies between instructions.

execution requiring a special logic to resolve this type of hazard. In the above example, instruction four and five reveal this type of hazard.

- *Write after write* hazards occur, if a subsequent instruction writes the same register file as an instruction before (cf. instruction three and five in Listing 4.1). For correctness, it must be guaranteed that only the last write operation is performed to the register file. For an in-order pipeline with only a single write-back stage, this hazard cannot occur. On the Freescale PowerPC MPC7448, a reorder buffer and in-order retirement are used to maintain the order of writes.

Pipelining only is useful, if the pipeline is filled with instructions. Whenever a dynamic branch instruction in form of a conditional branch or an indirect jump is encountered, the fetch stage cannot fetch the next instruction after the branch until the target of the dynamic branch is known, i.e. until the instructions computing the arguments for the branch have written back their results. This form is called a *control hazard* resulting in an empty pipeline, which has to be refilled with instructions, when the depending instructions have retired.

Performance Features

As stated before, efficient pipelining requires all stages to continuously perform work. Thus, hazardous events shall be prevented or their effect must be limited. In order to achieve this, several features have been implemented to improve the performance of pipelines.

To overcome structural and data hazards, a processor can execute instructions *out-of-order*. Execution of an instruction may start even if the predecessors of this instruction have not yet been started and wait for some input and no dependency-issues for the instruction to be started exists. Out-of-order execution thus increases the utilization of functional units of a processor, but also makes pipeline design more sophisticated since the order of reading and writing results must still be maintained. Additionally, a new problem arises: the processor must still be able to provide *precise exceptions*, i.e. it must always be clear, which

instruction triggers an exception. This is necessary for the processor to correctly restart the program after a system trap due to a page miss, etc.

To solve this problem and to keep a correct machine state, there exists two approaches: the use of a scoreboard [Tho65] and Tomasulo's algorithm [Tom67]. The latter introduces reservation stations in front of each functional unit for handling instructions waiting for inputs and a reorder buffer keeping track of sequential dependencies and register updates.

Pipelines can be designed to continue fetching even if later pipeline stages stall. Newly fetched instruction are then inserted into a *prefetch queue*, instructions to be decoded and issued are taken from this queue. Additionally, prefetching minimizes the delays due to fetching.

In order to reduce the effect of control hazards, prefetching can be coupled with an early decoding of branches. This is called *branch prediction*. The idea behind this is to redirect the fetching of instructions to the known target address of static branches. Additionally, the branch instruction can be removed from the instruction stream (*branch folding*).

This technique is also applicable for dynamic branches, but branch folding is limited to cases, where the result of the branch condition is still present, when the branch is decoded. Otherwise, the branch has to be *predicted* and remains in the instruction stream. For conditional branches, there exist only two possible outcomes: taken and not-taken. Even in this case, the fetch address has to be redirected assuming that the condition is known. All instructions being fetched based on this assumption must be marked as *speculative*. If the branch turns out to be *mispredicted*, instructions marked as speculative must be removed from the prefetch queue. Otherwise, the branch was well predicted and can be resolved, marked instructions become normal instructions.

The performance improvement due to out-of-order execution is reduced, if a dynamic branch is encountered. The instructions at the predicted target address cannot be executed until the outcome of the branch is resolved. Tomasulo's algorithm can be extended to allow speculative instructions to continue their execution by copying their mark also to the reorder buffer. If a prediction turns out to be wrong, speculatively executed instructions are simple flushed out of the pipeline and their entries in the reorder buffer have to be freed. Additionally, speculatively executed instructions are not allowed to write-back their results into register files or the memory until the prediction can be confirmed, ie. speculation can be resolved.

Another technique used to especially reduce the impact of read after write hazards is to use *forwarding*. The result of an operation can be directly passed to any other stage holding an instruction depending on this result. Thus, stall times can be reduced. Additionally, shortcuts are useful to speed up the resident time of instructions within functional units, e.g., if one argument of a multiplication

is 0, computation time can be shortened. Using these techniques, the latency of an instruction also depends on values of operands making timing analysis more complicated (cf. Section 4.4).

To increase the utilization of different functional units, *superscalar* (also known as *multiple-issue*) processors are used. In contrast to the simple DLX machine from above, the pipeline can issue more than one instruction to the corresponding functional unit in one clock cycle. How many instructions are issued in a dedicated cycle is computed dynamically based on the dependencies among the instructions in the prefetch queue and the availability of functional units.

Memory accesses still remain the bottleneck for a system's performance. *Store gathering* or *store merging* is a technique used to reduce the number of store operations to the memory subsystem. Subsequent store instructions that target the same location in memory are identified. The operations specified by the first and second store instructions are merged into a single store operation that subsumes the original ones. Thereafter, the new single store operation is performed reducing the overall amount of write operations to the memory subsystem. Using this technique strongly requires forwarding to ensure that correct values are read by other operations.

4.2 System Components

So far, only caches and pipelines of processors have been subject to timing considerations. Besides these parts, also other parts, which together form the whole system, do have an influence on the timing and the behavior of the system. Some of the most important components are shortly described in this section.

4.2.1 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 logically connects several peripherals over the same set of wires using a dedicated protocol.

In general, buses can be classified by the involved components, e.g., system buses like the *60x-bus* [Fre04] on the PowerPC architectures, memory buses connecting the memory controller with the memory slots, internal computer buses like Peripheral Component Interconnect (*PCI*) and external computer buses like *CAN* or *FlexRay* (cf. [WGR+09]).

Due to their functioning, most buses are clocked with a lower frequency than the CPU. E.g., the clock frequency of the PCI bus is specified to 33 MHZ in Rev. 2.0 and to 66 MHz in Rev. 2.1 of the PCI standard [PCI02].

The increasing demand for more computing power has forced a decoupled system development, i.e. the peripherals and the processors can be developed independently. A bus controller acting as the interface between the CPU and the other various devices allows for increasing the CPU speed without affecting the bus and the connected resources.

The analysis of the timing behavior of memory accesses is somehow special because these accesses cross the CPU/bus clock boundary. The gap between CPU and bus clock must be modeled within a micro-architectural analysis, as the time unit for those analyses is one CPU cycle; the analysis needs to know when the next bus cycle begins. If the analysis does not have this information it needs to account for all possibilities including the worst case: the bus cycle has just begun and the CPU needs to wait nearly a full bus cycle to perform a bus action. This pessimism would lead to less precise WCET bounds.

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 CPU 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 displacement of phase has to be modeled within a micro-architectural analysis in order to obtain a more precise WCET bound.

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*. The number of accesses that can overlap is called the *pipeline depth* of the bus. Hence, one distinguishes between pipelined and non-pipelined buses. The advantage of bus pipelining is better performance due to reduced idle time. On the other hand, pipelined buses need to arbitrate the incoming bus requests, e.g., if there is an instruction fetch and a data access at the same time, the arbitration logic needs to decide which bus request is issued first.

Figure 4.6 on the following page and Figure 4.7 on the next page show artificial timing diagrams for non-pipelined and pipelined access sequences on the 60x-bus, respectively. Note that the examples only show the most relevant parts of

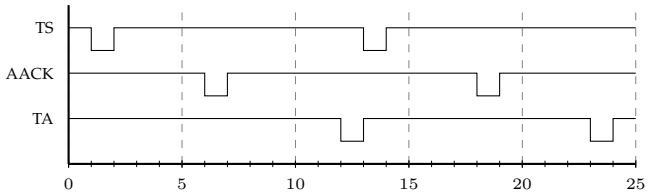


Figure 4.6 – Timing diagram of a non-pipelined access sequence of two accesses on the 60x-bus. Accesses finish after cycle 13 and 24, respectively.

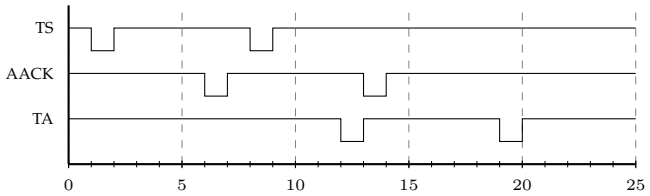


Figure 4.7 – Timing diagram of a pipelined access sequence of two accesses on the 60x-bus. Accesses finish after 13 and 20 cycles respectively.

the 60x bus protocol; the full protocol can be found in [Fre04]. A bus request from the CPU is initiated by issuing the `TS` (Transfer-Start) signal. At this point the CPU drives the requested memory address on the address wires. The memory controller reads this address and acknowledges it by issuing the `AACK` (Address-Acknowledge) signal to the CPU. After that the memory device drives the requested data on the data wires and signals this to the CPU by issuing the `TA` (Transfer-Acknowledge) signal.

Instances that can request access 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. Whereas the timing behavior of a single bus master is deterministic, more masters on a bus increase the difficulty of analyzing the traffic on the bus resulting in less precise bounds on latencies that can be guaranteed.

4.2.2 Memory

The component with the greatest impact on the timing of a processor is the main memory. Accesses are performed via a memory controller translating transactions on the system bus of the processor to the access protocol of the memory chip. The memory controller may be directly part of the CPU or it has to be provided as an external circuit. Often the RAM controller is integrated into

the main system controller (also named northbridge) also acting as the interface for other peripheral buses, e.g., the PCI bus.

Main memory can be roughly categorized into two major groups:

- static RAM (SRAM), and
- dynamic RAM (DRAM).

Static RAM normally uses 6 transistors per bit to prevent information from being disturbed when reading [HPG03]. This makes SRAM chips quite expensive, high capacity chips are not easily available. But the access timings for SRAMs are fixed, which provides for good predictability and eases WCET prediction.

Dynamic RAM uses only a single transistor to store a bit allowing a tight packing of RAM cells. Thus, accessing the information requires a lot of address lines. Due to the growing capacity of DRAMs, address lines were multiplexed, the number of address pins is cut. The first half of the address, which is sent first, is called the *row access strobe* (RAS). It is followed by the second half of the address called *column access strobe* (CAS). The names reflect the internal organization of a memory chip, i.e. a rectangular matrix addressed by rows and columns. An array contributes 1 bit to the output data. Since the output data normally ranges between 4 and 64 bits, the same amount of arrays is required and are then accessed in parallel within the chip. The contents of DRAM chips must be *refreshed* periodically, because each transistor loses its bit content. Refreshing copies all bits in a row and writes them directly back. During the periodical refresh phase, the chips cannot be accessed.

4.2.3 Peripherals

Embedded systems are defined as information processing systems embedded into enclosing products [Mar05]. Many of these system, especially real-time systems, have a number of sensors to receive their inputs and trigger a bunch of actuators used to pass the computed action to the environment. Accessing sensors and actuators can be realized in two ways:

- via special I/O instructions, or
- normal load/store operations for memory-mapped peripherals.

Accessing peripherals might be restricted, i.e. an access is only allowed, if the sensor's data is definitively needed by the processor core. Preventing a CPU from speculatively accessing those data is realized via marking the mapped memory regions as *guarded*.

4.2.4 Multiprocessors and DMA

Systems with more than one processor core induce a new problem, named *cache coherency*. Usually, those multiprocessor systems share the main memory and are connected through one system bus (including the problems arising with more than one bus master, cf. Section 4.2.1 on page 52). Fast and small caches are located near the processor's pipeline, so, the caches exists for each processor core in those systems. If one processor changes data in the cache, cache and memory coherency must be preserved. Thus, processors *snoop* the system bus for accesses of other components to the memory and implement a coherency protocol to maintain a coherent picture of the cache state w.r.t. the main memory. E.g., if a dirty line exists in the cache of one processor and a second one writes data to the main memory at the same address, the dirty cache line must no longer be written back to the main memory and must be invalidated from the cache. Otherwise, wrong data could end up in the main memory. Coherency protocols and snooping thus directly influence the state of the cache and the behavior of processor and must thus be covered by a WCET analysis.

Direct memory accesses (DMA) by other peripherals sharing the same main memory as the CPU cause conflicts for concurrent accesses by the CPU. As only one device is allowed to access memory at a certain point in time, other devices must wait. As DMA activity runs in parallel with normal program execution and is also not synchronized with it, the access latency for a memory access may vary due to the concurrent activity.

4.3 Hardware Predictability

In the previous sections, many features for increasing the performance of systems and for bridging the gap between fast processor cores and the slow main memory have been discussed. Most of these features lead to an increase of the average-case performance of the system, but their effect on the worst-case behavior is often not that obvious. E.g., speculative prefetching used for consecutively providing instructions to subsequent pipeline stages directly influences the state of the instruction cache of a CPU. If a conditional branch instruction is encountered, whose outcome has to be predicted, and this prediction is wrong, instructions that are fetched are not needed on the one hand, but on the other hand, may evict something useful from the instruction cache. So, this feature may have a bad impact on the performance of the system, at least on the worst-case performance.

As already discussed in Chapter 3 on page 27, only static WCET estimation methods can produce safe results on the runtime of a task on complex target architectures. But also the static methods are subject to timing predictability of

a system. According to [GRW11], *predictability* is the possibility to forecast the behavior of hardware. Thus, *timing predictability* is the possibility to predict the timing of a system within static analysis. Since static analyses compute valid results independently from any concrete input and concrete system state, the behavior of the (normally) deterministic hardware must be simulated resulting in some non-determinism.

As stated before, some of the features described in Section 4.1.1 and Section 4.1.2 directly influence the timing predictability of a system. The use of dynamic branch prediction, where the outcome depends on the history of program execution massively increases the search space, a static WCET estimation method must cover. Whenever the outcome could not be definitively predicted, the analysis must follow all possibilities, which, in case of branch prediction, means that both possibilities must be covered.

An identical problem arises in the presence of store gathering or store merging. Due to the nature of static WCET methods, accesses to memory may be imprecise. E.g., in case of an array access within a loop, where several loop iterations are merged into one collecting context (cf. [Weg11]), the access is not sharp, i.e. the access addresses an interval, whose boundaries are not the same. Thus, it cannot be safely determined, whether two successive store instructions can be merged or not, which increases the search space for timing analysis.

Imprecise data accesses are also problematic within one execution unit, when forwarding is used. E.g., the load/store unit of Freescale's PowerPC MPC7448 [Fre05a] massively uses forwarding for bridging the gap of slow memory accesses and the fast core by forwarding not yet written data to subsequent load instructions. Especially slow store operations are delayed until the load/store unit is idle, or the whole load/store unit is stalled by this operation. Thus, stores are inserted into a queue, and their effect is viewed as part of the architectural state, even if the store is not yet written back to main memory. Whenever a load instruction is to be processed, the queue picking all stores must be checked, whether the load operation addresses something that is also to be modified by a store in this queue. If there is a match in the modified address, data must be taken from the queue (since it is modified) and can be delivered fast, otherwise, the load operation must be passed to the memory subsystem. For a static WCET estimation tool, imprecise accesses are once again a problem since all possibilities, i.e. a matching address or no match in this example, must be simulated, which again results in a larger search space.

Another problem arises in the presence of several different buses connecting different peripherals. The system bus connecting the fast core with the slower memory subsystem is typically *clock synchronized* with the core clock of the processor. But for other buses, this need not hold. E.g., the internal PCI bus normally is not synchronized with the system bus. Thus, communication points, i.e. timestamps, where both the PCI bus and the system bus can exchange data,

are harder to predict. [Sch09] shows a formula that is used to tackle the problem of asynchronous clocks within the *aiT* WCET analyzer frameworks. It shows a formula for the safe conversion of external bus cycles into system bus cycles. The duration of an access given in external cycles corresponds to the interval given by

$$\left[\left\lfloor k\frac{F_{sb}^{min}}{F_{eb}^{max}}\right\rfloor ; \left\lceil\left((k+1)\frac{F_{sb}^{max}}{F_{eb}^{min}}\right)+1\right\rceil\right]$$

where

- k is the latency in external bus cycles of the access,
- F_{sb}^{min} and F_{sb}^{max} are the minimal and maximal frequencies of the system bus clock, and
- F_{eb}^{min} and F_{eb}^{max} are the minimal and maximal frequencies of the external bus clock.

Communication between both buses thus is possible at each integral value contained in this interval.

Whereas features mentioned so far increase the search space for a WCET analysis, other features cannot be statically predicted. As stated in Section 4.1.1 on page 43, the replacement policy of a cache influences its behavior and thus directly influences the performance of a system. E.g., the L2 cache of the Freescale PowerPC MPC7448 uses a random replacement strategy. Since the line to be evicted is randomly chosen, this strategy can be viewed as not predictable, or, to be more precise, only one (namely the last) cache line can be predicted. A detailed discussion of cache replacement policies and their impact on WCET estimation can be found in [Rei08, WGR+09].

Also the presence of asynchronous events as used for implementing exchange protocols in multi-core systems are currently not predictable in general. Especially the concurrent accesses of different cores to the main memory, but also the synchronization needed on cache level, cannot be predicted by todays static WCET estimation tools.

In contrast to these asynchronisms, periodic or sporadic events can be covered by statistical means. E.g., for DRAM refreshes that occur every x milliseconds, the impact of a refresh is well known. Each page of the dynamic memory must be loaded and written back to save the content of the transistors. In other words, the effect of a DRAM refresh can be bounded to the time for closing the actual open page in the controller, time for open, read, write and close any page of the memory and reopen the previously closed page again. Since these timings are constant, and refreshes occur periodically, the effect of these asynchronous events can be covered by a static WCET analysis. The same statistical approach is feasible for direct memory accesses, if the frequency of DMA is statically known.

Another predictability issue arises due to the combination of several features. Whereas a write-back in the presence of a LRU replacement cache policy and an empty initial cache content can be predicted, this becomes disappointing for a PLRU replacement policy. In contrast to LRU, the PLRU replacement strategy allows only to predict the content of 2 ways [RGBW07]. As a result, predicting evictions becomes impossible and as a result, from the point of view of a static analysis, it is unclear, if a write-back memory operation is necessary.

Currently, the trend in embedded systems tends to employ multi-core architectures to satisfy the increasing demand for more and more computing power [Low06]. All characteristic challenges from single-cores are still present in the multi-core design, but the multiple cores can run multiple instructions at the same time. To interconnect the several cores, buses (e.g., Freescale's PowerPC MPC603e, [Fre02]), meshes, crossbars, and also dynamically routed communication structures (e.g., Freescale's QorlQ P4080, [Fre08]) are used. Most multi-core architectures offer a sophisticated memory hierarchy including private L1 caches, but also some shared caches. Access to the interconnect usually requires an arbitration of accesses from the different cores. The shared physical address space requires additional effort in order to guarantee a *coherent* system state: Data resident in the private cache of one core may be invalid, since modified data may already exist in the private cache of another core, or data might have already been changed in the main memory. Thus, additional communication between different cores is required. In general, access to a shared resource might cause the following traffic to appear on the processor's interconnect: A cacheable read access issued by one core

- may cause no communication on the processor's interconnect in case of a cache hit,
- may initiate a read request in case of a cache miss, and
- may initiate a write access first to evict modified data from the cache.

A write access to a cacheable memory area issued by one core

- may cause no traffic on the processor's interconnect in case of a cache hit,
- may cause some coherency traffic in case of a cache hit to update directories of other cores [Mar08],
- may initiate a read access in case of a cache miss, and
- may initiate a write access first to evict modified data from the cache.

Hence, interconnect traffic initiated by one core in order to process an instruction is composed of *data traffic*, *eviction traffic*, and *coherency traffic*. Accessing the processor's interconnect to aquire some data may be impossible at a time due to interconnection traffic initiated by other cores. Thus, predicting the timing behavior of multi-cores without enforcing privatization as proposed by PROMPT

(PRedictability Of Multi-Processor Timing, [WFC+09]) still remains an open research issue. [KSP+12a, KSP+12b] describe an experiment showing that a task's average runtime on a multi-core increases tremendously, if other tasks run on other cores, even if spatial isolation is guaranteed.

4.4 Timing Anomalies

Especially the interaction of several features in a pipeline and also the interaction with the caches may be hard to predict and lead to some counterintuitive behavior of the whole system: the case, where a locally faster execution time (e.g., a cache hit) leads to an increase in the overall execution time of a program compared to the locally worst execution time (e.g., the cache miss), is called a *timing anomaly*. This effect was originally stated in [Lun02].

To compute safe timing guarantees, a static timing analysis has to consider all possible execution paths caused by any non-determinism in the abstract hardware model. Due to the loss of predictability, the static analysis of architectures featuring timing anomalies requires much more effort in terms of computational power and memory consumption.

[RWT+06] classify timing anomalies into three different classes:

- *Scheduling timing anomalies* are the most common ones. Most timing anomalies found in the literature correspond to this class. Figure 4.8 on the next page shows one instance. Depending on the execution time of the task A, a faster execution might lead to a globally longer schedule. This kind of anomaly is well-known in the scheduling domain.
- A *speculation timing anomaly* is shown in Figure 4.9 on the facing page. An initial cache hit allows the pipeline to speculatively prefetch an instruction that is not cached leading to an overall longer execution.
- *Cache timing anomalies* are caused by some non-LRU replacement strategies (cf. [Ber06, Geb10]).

A system is said to have *domino effects* [LS99a] if there are at least two hardware states s_1 and s_2 such that the difference in execution time of the same program starting in s_1 and s_2 respectively may be arbitrarily high, i.e. cannot be bounded by a constant. Otherwise, if the effect of a timing anomaly can be bounded by a factor k, this is called a *k-bounded timing anomaly*.

The existence of domino effects is crucial for timing analysis. Unfortunately, domino effects show up in real hardware. [Sch03] describes a domino effect in the pipeline of the Freescale PowerPC MPC755. Another example basing on

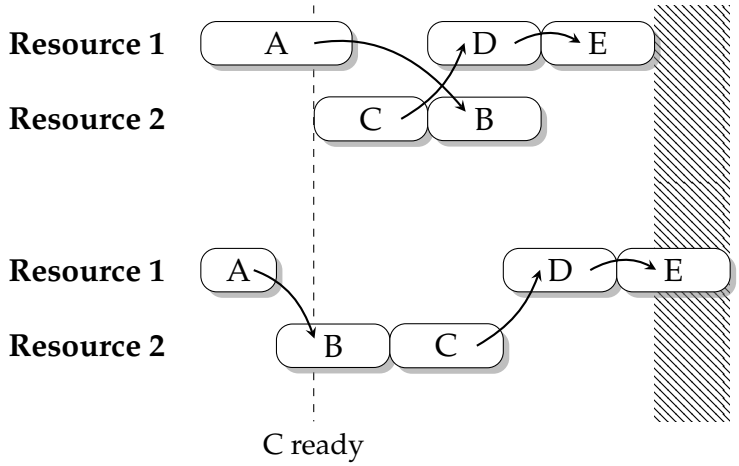


Figure 4.8 – Scheduling timing anomaly.

the PLRU replacement policy of caches is given in [Ber06]. [Geb10] exhibits that timing anomalies are not limited to complex hardware architectures.

In general, architectures can be classified into three categories basing on the presence of timing anomalies and domino effects [WGR+09]:

- *Fully timing compositional architectures* do not exhibit timing anomalies. Hence, a static WCET analysis can safely follow local worst-case decisions. One example for this class is the ARM7 processor core [ARM04].

- *Compositional architectures with k-bounded effects* 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 decisions by adding a constant number of cycles to the local worst-case decision. The Infineon TriCore [Inf02] is assumed, but not formally proven, to belong to this class.

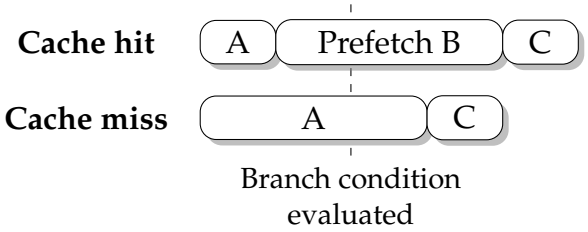


Figure 4.9 – Speculation timing anomaly.

- *Non-compositional architectures* exhibit domino effects and timing anomalies. For such architectures timing analyses always have to cover the whole search space (i.e. must follow each possible decision) since a local effect may influence the future execution arbitrarily. A domino effect within the pipeline of the Freescale PowerPC MPC755 [Fre01] has been shown in [Sch03]. Furthermore, the Freescale PowerPC MPC7448 [Fre05a] is assumed to belong to this class.

4.5 VHSIC Hardware Description Language

Modeling modern processors combining features described in the previous sections is simply not possible for a single person. Large development teams work together to design modern digital electronic systems. Shorter and shorter development cycles, a reduced time to market and reliability requirements become important goals. Thereby, simulation and verification are crucial issues. In 1994, a faulty behavior within the floating point unit of the Intel Pentium [Hal95] led to a recall of a huge charge of processors. The use of formal hardware description languages, designed to support the design process as well as the simulation and verification process becomes more and more important in the development of digital electronic systems.

When talking about digital electronic systems, several definitions exist. They range from single VLSI circuits[4] to complete systems including peripherals. Due to their complexity, it is not possible to comprehend the complexity of the systems in their entirety. Thus, there is a need for finding methods of dealing with the complexity to be able (at least with some confidence) to design components and systems meeting their requirements. To ease the communication and understanding and to enhance the interoperability, languages for describing digital electronic systems, called *hardware description languages* (HDLs), have been developed.

One of the most common languages for describing digital electronic systems is VHDL. It arose from the US Department of Defense *Very High Speed Integrated Circuits* (VHSIC) program and was originally developed as an alternative to huge, complex manuals which were subject to implementation-specific details. Due to the need for a standard language describing the structure and function of integrated circuits, the *VHSIC Hardware Description Language* (VHDL) was developed. Concepts and syntax of VHDL are very similar to the Ada programming language [Bar95] since the US government wants to avoid re-inventing concepts that had already been thoroughly tested in the development of Ada.

[4] *Very-large-scale integration* (VLSI) is the process of creating integrated circuits by combining thousands of transistors into a single chip.

Subsequent development of VHDL is made under the auspices of the Institute of Electrical and Electronic Engineers (IEEE), a first version of the IEEE standard 1076 [IEE87] was published in 1987. Like all IEEE standards, also the VHDL standard is subject to review every five years.

VHDL defines a formal notation intended for use in all phases of the creation of electronic systems [Ash01]. Thus, VHDL allows to describe the structure of a system, how it is decomposed into subsystems and how these subsystems are interconnected. Furthermore, it allows the specification of the function of a system using a familiar programming language form. Hardware prototyping for testing a system is expensive and very time-consuming, so VHDL was also designed to simulate a whole system before being manufactured. Additionally, design synthesis allows the designers to focus on more strategic design decisions.

VHDL allows designers to specify digital electronic systems in a high-level human-readable notation, and allows for both, *simulation* and *sythesis* of the model. Therefore, the focus of the language ranges from specifying circuits at wavefront level to describing large system behaviors with high-level constructs. As a result, the standard is huge resulting in a more restricted subset for automatic design synthesis [IEE99].

In June 2006, VHDL technical committee of Accellera released the so-called draft 3.0 of VHDL-2006 standard. Key changes include incorporation of child standards (1164, 1076.2, 1076.3) into the main 1076 standard, and extend the set of operators with the goal to improve the quality of synthesizable VHDL code.

In September 2008, VHDL 4.0 also informally known as VHDL-2008 has been released as IEEE 1076-2008 addressing more than 90 issues discovered during the trial period for version 3.0 and including enhanced generic types.

4.5.1 Modeling Digital Systems

Modeling a digital system requires a systematic methodology of design. Starting with a requirements document, an abstract structure meeting these requirements has to be designed. This abstract structure can be decomposed into a collection of components that interact to perform the function that is required. The process of decomposition can be iterated arbitrarily until a level is reached, where some ready-made, primitive components performing the required function exists.

The advantage of this methodology is that each component of the system can be designed independently. The use of subcomponents thus can be viewed as a kind of *black-boxing* abstracting away details about their composition and implementation. Often, such subcomponents are part of the *intellectual property* (IP) of one party and can be licensed to another party. In each particular stage of the design process, only a small amount of information that is relevant for the

current design focus must be present and the designers are not overwhelmed by masses of details. Thus, the result of this design methodology is a *hierarchically composed system*.

In the following, the term model will be used for the understanding of a system. A *model* represents the information which is relevant and abstracts away all irrelevant details. Consequently, there might exist more than one model for the same system or component since depending on the context, relevant information might differ. E.g., one model focuses on the function of a system, whereas another model concentrates on the composition of the system from subsystems.

[Ash01] classifies models into three domains:

- *Functional models* focus on the operations performed by the system.
- *Structural models* reveal details on the composition of interconnected systems.
- *Geometric models* deal with the layout of a system in physical space.

Each domain can be divided into different levels of abstractions (cf. Figure 4.10 on the next page by [GK83]). The top level (i.e. the outer circle) provides an overview on function, structure and geometry of the system. Lower levels successively introduce more and more details.

At the most abstract functional level, the function of a system is described in terms of an algorithm, comparable to an algorithm for a computer program. This is often called *behavioral modeling*. At the same abstract level, the structure of a system may often be described as an interconnection of components such as a processor, different kinds of memories, sensors and actors. Within the structural domain, this is often called *processor-memory-switch*. Within the geometric domain, the system to be implemented as a VLSI circuit is often described using a floor plan showing the components of a system and their arrangement on a silicon die.

The next level of abstraction, depicted by the second ring in Figure 4.10 on the facing page, describes a system in terms of units of data storage and transformations. Within the structural domain, this is called *register-transfer* composed of a data path and a control section. The *data path* contains the description of registers, data between these registers is transferred through transformation units. The *control section* sequences operations of the data path components. In the functional domain, this level is often described using a *register-transfer language*, storage is represented using variables, and transformations are represented by arithmetic and logical operators. Within the geometric domain, this level is often associated with standard library cells used to implement registers and transformations which have to be placed on the physical die according to the floor plan.

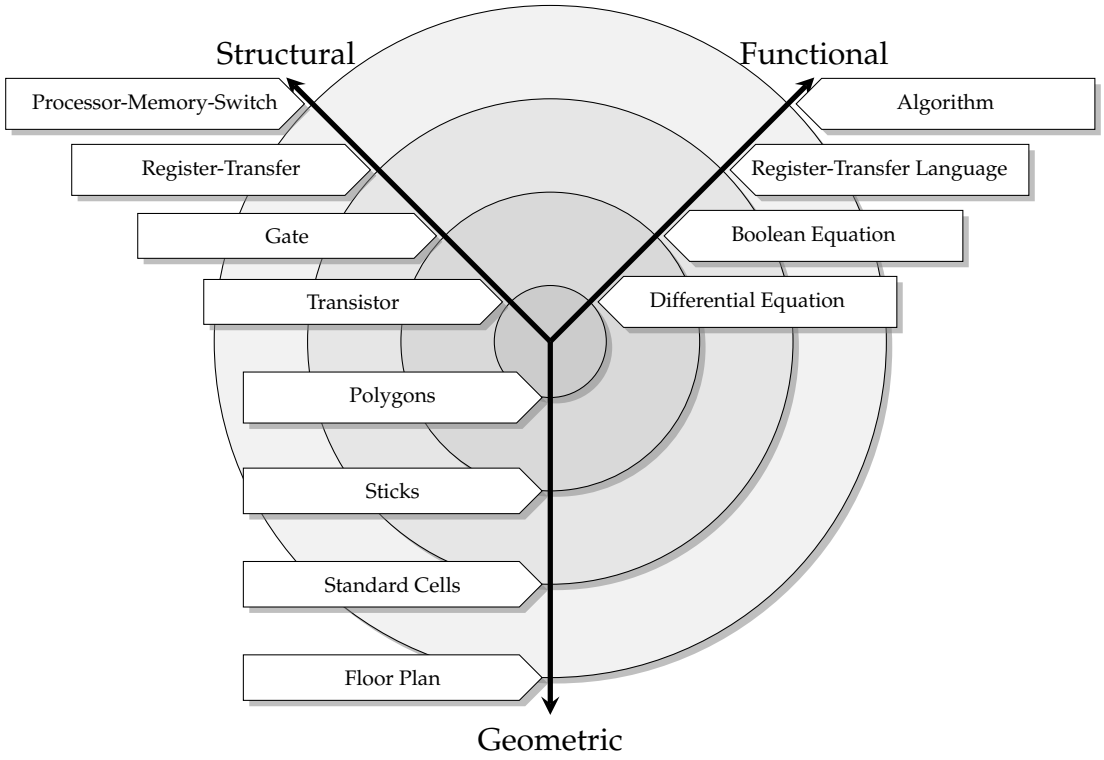


Figure 4.10 – Domains and levels of abstraction (cf. [GK83]). The axes show the different domains of modeling. The concentric rings show the different levels of abstraction – more abstract levels on the outside, more detailed ones towards the center.

The third level shown in Figure 4.10 is the conventional logic level. Structure is modeled using interconnections of gates, functions are modeled by boolean equations, and the geometric domain uses the notation of virtual grids.

At the most detailed level, structure is modeled using individual transistors, functions using differential equations that relate current and voltage in the circuit, and the geometry deals with polygons for each mask layer of an integrated circuit. Due to the availability of design tools (e.g., Synopsys [Syn]), designers do not need to work on those detailed levels. Translation from higher levels of abstraction, namely from register-transfer level, is automated.

Models of every domain and of different abstraction level can be described in many ways. E.g., a structural model can be expressed using a circuit schematic. Graphical symbols are used to express subsystems (or components), and instances of these components are connected using lines that represent wires. The same information can also be represented in a textual manner, e.g., in form of a net list. Within the functional domain, representation usually is based on formal mathematical methods that ease testing and verification of a system and are

typically based on programming languages. This supports reliability and thus reduces the time to market.

VHDL as a modeling language includes facilities for describing structure and function at a number of abstraction levels, from the most abstract level to the gate level. Due to an attribute mechanism, it can further be used to annotate a model with information from the geometric domain. Thus, VHDL explicitly offers the possibility to create arbitrary models for each domain. VHDL is intended as a modeling language for specification, simulation, and, when restricted to the synthesizable substandard, for automatic hardware synthesis.

In the following, this thesis focuses on the second abstraction layer, namely the register-transfer level, and the synthesizable IEEE substandard 1076.6. Commercial synthesis tools like Synopsys also work on this level and allow use of powerful standard libraries providing implementations of widely used components such as flip-flops, etc. Furthermore, this thesis is only concerned with synchronous models, i.e. models that are synchronized to an external clock signal.

4.5.2 Register-Transfer Level

Models described at the level of register transfer are more or less a collection of process statements. Each process is a collection of actions that are executed in sequence. These actions are called *sequential statements* and are comparable to conventional programming languages. The types of sequential statements to be performed include assignments to either signals or variables, expression evaluation, conditional execution (if- and switch-statements), repeated execution (for- and while-loops), and subprogram execution (function and procedure calls). Additionally, the sequential execution can be *suspended* by dedicated wait statements.

Listing 4.2 on the facing page shows the specification of a simple 3-bit counter in VHDL. The description of the circuit consists of an interface declaration defining the in- and output signals of the circuit and of one or more implementation(s). In VHDL, the interface declaration is called an *entity*, the implementation an *architecture*.

The entity declaration defines the input ports of the circuit (`clk` and `rst`). The counter is designed as a *synchronous circuit*, i.e. all computations are synchronized on the transitions of a global signal. This signal is referred to as the *global clock* (`clk` in the sample circuit). 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 a *process* (`P1`). A process executes its code, whenever one of the *signals* contained in the process' *sensitivity list* (`clk`

```
entity counter is
    port (clk : in std_logic;
          rst : in std_logic;
          val : out std_logic_vector (2 downto 0));
end entity;

architecture rtl of counter is
    signal cnt : std_logic_vector (2 downto 0);
begin

 P1: process (clk, rst) is
     begin
        if (rst = '1') then
            cnt <= "000";
        elsif (rising_edge (clk)) then
           if (cnt < "111") then
              cnt <= cnt + '1';
           else
              cnt <= "000";
           end if;
           val <= cnt;
        end if;
     end process;
end;
```

Listing 4.2 – 3-bit counter in VHDL.

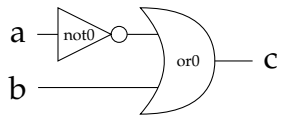
and `rst`) changes its value. Thus, the sensitivity list of a process is an implicit wait-statement at its end.[5] After execution of all statements, execution suspends until another change of at least one signal's value. Within an architectural body, many processes may be used to implement the desired behavior. All of these processes (besides some other language constructs) are called *concurrent statements* since they all run in parallel.

A VHDL process consists of a set of local *variables* that are only accessible from inside the process. By contrast, local *signals* can be accessed by more than one process, but only one process is allowed to drive the value of a signal.[6] Within a process, execution of statements is done sequentially.

VHDL makes a distinction between assignments to a variable and to a signal. Assigning a value to a variable takes effect immediately (i.e. the next reference

[5] In VHDL, the use of explicit wait-statements and sensitivity lists is exclusive.

[6] In full VHDL, *resolution functions* can be used for value computation of signals being driven by two or more processes.




```
entity implies is
    port (a, b: in std_logic;
          c: out std_logic);
end entity;

architecture struct of implies is
    signal int_neg: std_logic;
begin
    not0: entity invert
          port map (a, int_neg);

    or0:  entity or2bit
          port map (int_neg, b, c)
             ;
end;
```

Figure 4.11 – Composition of VHDL components.

of 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). E.g., in Listing 4.2 on the previous page, the signal assignment `cnt <= cnt + '1';` schedules the next value of `cnt` to be `cnt` plus one, but the next reference `val <= cnt;` 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.

VHDL also supports component-based circuit specifications. Figure 4.11 gives an example for hierarchical composition. Here, a circuit for the logical implication $a \rightarrow b$ for two inputs `a` and `b` and the output `c` is built from a logical-or gate `or2bit` and a negation gate `invert`, implementing the implication by the formula $c = \neg a \vee b$. Note that `or0` is an instance of the generic entity `or2bit`, as `not0` is one of the entity `invert`. In VHDL, use of predefined entities is called *component instantiation*. Having a hierarchically composed specification of a circuit, elaboration has to be performed in order to get a flat definition of it.

4.5.3 Elaboration and Synthesis

Once a hierarchical composed system has been modeled, *elaboration* has to be performed. Elaboration does all the required renaming for unifying names, wires all structural descriptions, etc. The result of the elaboration process is a collection of processes interconnected by *nets*.

Elaboration starts with the topmost entity of a design hierarchy. Every component instantiation within the implementation is substituted with the contents of an architecture body of the corresponding entity. Since many architecture can exist for a component, the implementation to be used must be explicitly chosen. After substitution, the architecture body is elaborated by first instantiating all signals it declares, then by elaborating each concurrent statement in its body.

This procedure is repeated recursively until all component instantiations are replaced by their corresponding implementation and all signals are created. The result is a flattened system model consisting of a collection of nets comprising signals and ports, and processes that sense and drive the nets.

After elaboration, the system is ready to be simulated, and, if no faulty behavior can be detected by simulation, the system can be synthesized for usage on a FPGA[7] or an ASIC[8]. This is called *hardware synthesis*. The idea behind this is to allow the designer to model the system in abstract terms, i.e. the designer must not think about how best to implement the design in hardware logic. This is automated by a synthesis tool converting the abstract description into a structural description at a lower level of abstraction.

Not all constructs of VHDL are suitable for synthesis. Most constructs that explicitly deal with timing such as `wait for 5 ns;` are not synthesizable despite being valid for simulation. Thus, the IEEE standard 1076.6 [IEE99] – the synthesizable subset of VHDL – defines constructs and idioms map into common hardware for hardware synthesis. There exists a wide range of synthesis tools, all covering different sets of language constructs. Thus, it is generally considered a "best practice" to write very idiomatic code for synthesis as results can be incorrect or suboptimal for non-standard constructs.

4.5.4 Semantics

Since VHDL explicitly allows for simulation, every model after elaboration can be *executed*. The underlying semantics and thus the operation of a system can be seen as a chronological sequence of events. This is also called *discrete-event simulation*. At a point in time, a process can be simulated by changing the value of a signal to which the process is sensitive. The process is resumed and may schedule new values to be given to signals at some later time, which is called *scheduling a transaction* on a signal. If the newly assigned value differs from the previous value, an *event* occurs, and all other processes being sensitive to this signal must be resumed.

[7] A *field-programmable gate array* (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing.

[8] An *application-specific integrated circuit* (ASIC) is an integrated circuit customized for a particular use, rather than intended for general-purpose use.

```
architecture rtl of counter is
    signal cnt : std_logic_vector (2 downto 0);
begin
 P1: process (clk, rst) is
     begin
        if (rst = '1') then
            cnt <= "000";
        elsif (rising_edge (clk)) then
           if (cnt < "111") then
              cnt <= cnt + '1';
           else
              cnt <= "000";
           end if;
        end if;
     end process;

 P2: process (cnt) is
     begin
        val <= cnt;
     end process;
end;
```

Listing 4.3 – Alternative implementation for the 3-bit counter.

Listing 4.3 shows a different implementation of the 3-bit counter example shown in Listing 4.2 on page 67. The implementation is given in form of two processes. The semantics of this VHDL model is depicted in Figure 4.12 on the facing page. Execution of the model starts with an *initialization phase*, where each process of the elaborated model – `P1` and `P2` in this example – runs until it suspends. During this phase, each signal (and variable) of the system is given an initial value. Simulation time is set to zero, each process is activated and its sequential statements are executed. In general, every process will include at least one signal assignment in its body to schedule a transaction at a later execution step. Sequential execution resumes until a wait-statement is reached causing the process to suspend. In this example, `val` and `cnt` are both set to `0`. Since the synthesizable VHDL subset does not allow the specification of clocks[9], the clock signal is external and is changed from the outside.

After the initialization phase, the *activation phase* starts. This phase repeatedly executes the *simulation cycle* which can be described as follows:

1. Execute processes until they suspend.

[9]The synthesizable subset of VHDL does not allow a process to sense a signal it drives. Also use of timeouts is not allowed, thus, a frequent change of a signal's value cannot be modeled.

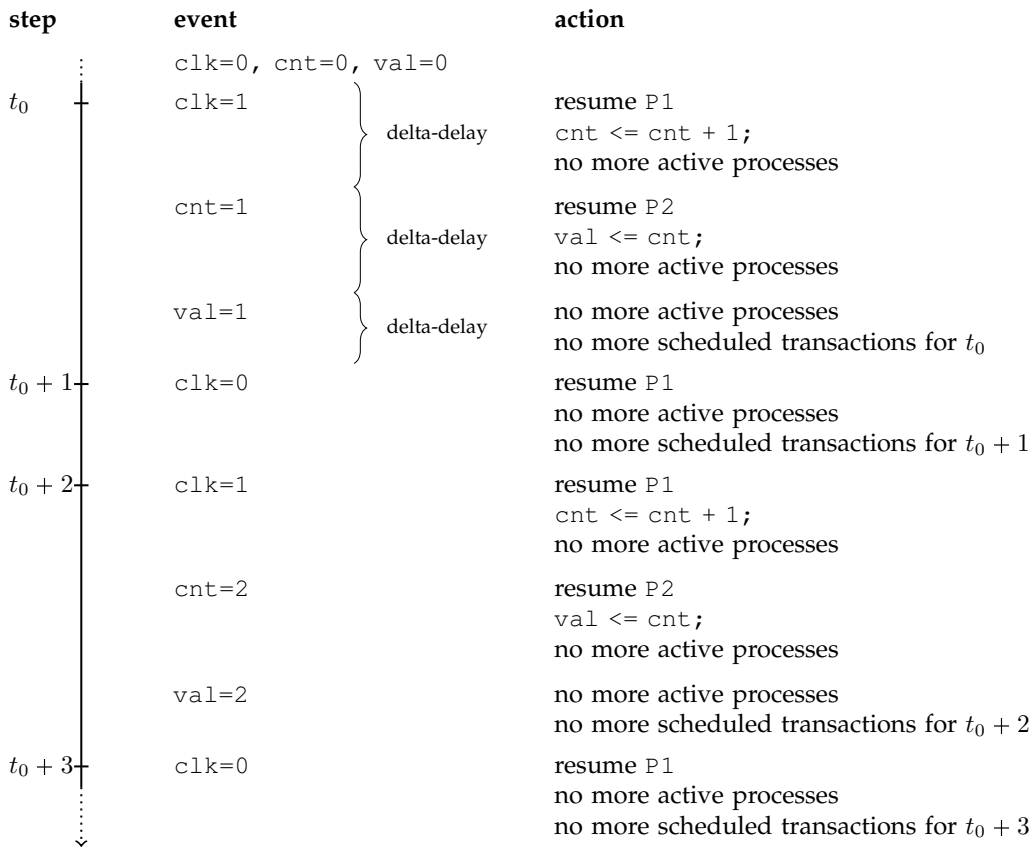


Figure 4.12 – Execution of a VHDL model.

2. If all processes are suspended, make all scheduled signal assignments visible at once.

3. If there is a process being sensitive on a signal having changed its value, resume this process and go to step 1.

4. Otherwise, an external signal must change its value (e.g., the external clock signal). If this happens, resume all processes waiting for this signal and go to step 1.

In the example shown in Listing 4.3 on the preceding page, all processes have suspended after the initialization phase. Since there is no scheduled transaction, nothing happens until the change of the external clock signal `clk <= 1`. This change causes `P1` to resume and triggers the transaction `cnt <= cnt + 1;` to be scheduled. Afterwards, `P1` suspends, and no other process is actually running (cf. Figure 4.12). All scheduled transactions are now made visible at once. The time until this happens is called a delta-delay within VHDL.

A *delta-delay* is an infinitesimally small delay that separates events occurring in successive simulation cycles but at the same simulation time.

The change of the signal `cnt`, as scheduled by `P1`, leads to a reactivation of process `P2`. This process schedules the transaction `val <= cnt;`. After the next delta-delay making this transaction visible, no process has to be reactivated. So, simulation time needs to be advanced to the change of an external signal, which happens at timestamp $t_0 + 1$ in this example. Since the specified circuit is synchronized to rising clock events, nothing happens here and the simulation time is advanced again. At the next rising clock edge, at simulation cycle $t_0 + 2$, the same actions occur as described for simulation cycle t_0.

The semantics of VHDL can be seen as a two-level semantics: sequential process execution at its first, signal update and process reevaluation at its second level. The sequential execution rules for a VHDL process are depicted in Figure 4.13 on the next page. In order to formally define the semantics of a VHDL process, it is necessary to define, what action is performed by it (cf. [MPS09]).

Definition 4.5.1 (Program, program counter)
A *program* Π_p is the list of sequential statements of a process p. The current statement of a program is denoted by the *program counter* ζ_p. The function $next(\zeta_p)$ returns the address of the next statement of a program Π_p. If the current statement is the last one in a program, $next(\zeta_p)$ returns the special program counter ζ_{sus} indicating that the process is suspended. The function $start(b)$ takes a list of statements b and returns the address of the first statement in b.

In other words, each process in VHDL is comparable to programs in other imperative programming languages.

Definition 4.5.2 (Environment)
The *environment* $\Theta \in Env$ of a VHDL process p is a mapping from logical names to values. A *logical name* in VHDL can be either a variable v, a signal s, or a scheduled signal $\bar{s}$.

Definition 4.5.3 (Process context)
The *context* of a process p is the tuple (Θ, ζ_p, Π_p) where

- Θ is the *environment* of process p,
- ζ_p is the program counter of process p, and
- Π_p is the program of p.

The inference rule (I) on the facing page covers variable assignments. W. l. o. g., evaluation of VHDL operations is embedded into the function $eval$, inference rules describing the semantics of VHDL expressions are detailed in [Hym03]. Note that during evaluation, $eval$ always works on the current values of signals,

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

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

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

$$fcond\ \frac{\Pi[\zeta] \Rightarrow \text{if } expr \text{ then } b_1 \text{ else } b_2 \text{ end if};\ \ \Theta \vdash eval(expr) = \text{false}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b_2), \Pi)} \quad \text{(IV)}$$

$$tloop\ \frac{\Pi[\zeta] \Rightarrow \text{while } expr \text{ loop } b \text{ end loop};\ \ \Theta \vdash eval(expr) = \text{true}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, start(b), \Pi)} \quad \text{(V)}$$

$$floop\ \frac{\Pi[\zeta] \Rightarrow \text{while } expr \text{ loop } b \text{ end loop};\ \ \Theta \vdash eval(expr) = \text{false}}{(\Theta, \zeta, \Pi) \rightharpoonup (\Theta, next(\zeta), \Pi)} \quad \text{(VI)}$$

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

Figure 4.13 – VHDL sequential execution semantics.

never on scheduled ones. The process environment is updated with the new value assigned to variable v. Inference rule (II) covers signal assignments. While in case of a variable assignment, the environment is directly updated with the new value, this is not the case for signal assignments. Here, the new value is a scheduled transaction and must be seen as the future value.

The behavior of conditional statements is specified in the inference rules (III) and (IV). When the condition expression evaluates to true, control is transfered to the first instruction of the subsequent block b_1. Otherwise, control is transfered to the first instruction in the block b_2. Obviously, switch-statements can be easily

$$exec\ \frac{\begin{array}{c}\exists l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_l, \Pi_l, \omega_l) \wedge \zeta_l \neq \zeta_{sus} \\ (\Theta, \zeta_l, \Pi_l) \rightharpoonup_{seq} (\Theta', \zeta_l', \Pi_l) \\ \boldsymbol{\rho}' = \lambda(t \in \mathbb{L}).\begin{cases}(\zeta_l', \Pi_l, \omega_l) & \text{if } t = l, \\ \boldsymbol{\rho}(t) & \text{otherwise.}\end{cases}\end{array}}{(\Theta, \boldsymbol{\rho}) \rightharpoonup (\Theta', \boldsymbol{\rho}')} \tag{VIII}$$

$$delta\ \frac{\begin{array}{c}\forall l \in \mathbb{L} : \boldsymbol{\rho}(l) = (\zeta_{sus}, \Pi_l, \omega_l) \quad \Theta' = \lambda t.\begin{cases}\Theta(\bar{s}) & \text{if } t = s, \\ \Theta(\bar{s}) & \text{if } t = \bar{s}, \\ \Theta(t) & \text{otherwise.}\end{cases} \\ \boldsymbol{\rho}' = \lambda(l \in \mathbb{L}).\begin{cases}(start(\Pi_l), \Pi_l, \omega_l) & \text{if } \exists s \in \omega_l : \Theta(s) \neq \Theta(\bar{s}), \\ (\zeta_{sus}, \Pi_l, \omega_l) & \text{otherwise.}\end{cases}\end{array}}{(\Theta, \boldsymbol{\rho}) \rightharpoonup (\Theta', \boldsymbol{\rho}')} \tag{IX}$$

Figure 4.14 – VHDL simulation semantics.

transformed into conditional cascades, so the rules for these statements are left out in Figure 4.13 on the preceding page.

Although the synthesizable VHDL subset only defines for-loops, they can be easily transformed to while-loops. Their semantics is specified in the inference rules (V) and (VI). Similar to conditional statements, control is transfered to the loop body, when the conditional expression evaluates to $true$. Otherwise, control is transfered to the first statement after the loop body.[10]

Inference rule (VII) on the previous page describes the semantics of empty blocks, e.g., an empty else-part within a conditional statement. The environment remains unchanged, only the program counter is updated to the next statement.

Process termination in the inference rules in Figure 4.13 on the preceding page is guaranteed by the definition of the function $next(\zeta)$. A VHDL process terminates (i.e. suspends execution) when reaching the special program counter ζ_{sus} as defined above.

The global simulation semantics, i.e. the level of processes reactivation, for a VHDL model is shown in Figure 4.14 working on a set of processes.

[10] As this thesis focuses on the synthesizable substandard of VHDL, all loops are assumed to be bounded, i.e. no infinite loops are part of the models.

Definition 4.5.4 (Process)
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 as defined above, and ω_l is a set of signal names representing the sensitivity list of $\boldsymbol{\rho}(l)$.

Definition 4.5.5 (Global execution context)
The *global execution context* of a VHDL simulator is the tuple $(\Theta, \boldsymbol{\rho})$ where

- Θ is the environment as defined above.
- $\boldsymbol{\rho}$ is a map from *process labels* $\mathbb{L}$ to processes.

Sequential execution of all running (i.e. not suspended) processes is specified in inference rule (VIII) on the facing page. The sequential execution rules shown in Figure 4.13 on page 73 are embedded in the function $\rightharpoonup_{seq}$. This rule covers step 1 in the description of the VHDL simulation cycle.

Transaction handling (i.e. handling of scheduled signal assignments) and process reactivation is covered by inference rule (IX) on the facing page. For each process p having at least one signal in its sensitivity list ω_p that actually has changed its value the program counter ζ_p is set to the first instruction in the program. This corresponds to the steps 2 and 3 of the simulation cycle.

As stated earlier, it is not possible to model the frequent change of a clock signal within the synthesizable subset of VHDL. Thus, the rules given in Figure 4.14 do not describe, how simulation time advances. Simulation proceeds when all scheduled transaction have been processed and no further update occurs. The advance in simulation time (cf. step 4 of the simulation cycle) must be handled externally. The combination of inference rules given in Figure 4.13 on page 73 and Figure 4.14 on the facing page specifies the semantics of the synthesizable subset ofVHDL and can be used to generate a cycle-accurate simulator for a given VHDL model.

4.5.5 Related Languages

Besides VHDL, a variety of other hardware description languages have been developed. The first languages were ISPS [BBCS77] developed at Carnegie Mellon University, and KARL, developed at University of Kaiserslautern, in 1977. Whereas ISPS does only offer the possibility to simulate a design but not to synthesize it, KARL also includes capabilities for supporting VLSI chip floorplanning and structured hardware design.

The first modern HDL was Verilog [TM02], developed in 1985. As for VHDL, Verilog was initially developed to document and simulate system models to enable engineers to work at a higher level of abstraction. With the introduction

of logic-synthesis supporting the automatic compilation of the description into transitor-level netlist descriptions, modern HDLs come to the fore regarding the design of integrated circuits.

As VHDL, Verilog also offers support to describe state-machines and temporal dependencies, the syntax was designed to be related to the C programming language [KR88]. A Verilog design consists of a hierarchy of modules, modules communicate through a set of declared input, output, and bidirectional ports. As in VHDL, a module consists of a set of concurrent blocks, each block is a list of statements that is executed sequentially. The blocks are executed concurrently.

Also VHDL and Verilog have many things in common, both HDLs differ in some points: Whereas VHDL is patterned after Ada, Verilog is patterned after C. Basically, this means that Verilog is terse, and type-checking is made very airy, whereas VHDL is verbose and strongly typed. Another difference is that Verilog is a case sensitive programming language, whereas VHDL is case insensitive.

Within a few years, both, VHDL and Verilog, emerged as the dominant HDLs in the electronics industry, while older HDLs disappeared step by step from use. Anyhow, VHDL and Verilog share the limitation that both are not suitable for analog/mixed-signal circuit simulation. Specialized hardware description languages such as Confluence [Haw03] were introduced with the goal to explicitly fixing this limitation, though none were ever intended to replace VHDL or Verilog.

5

Timing Model Derivation

> Nothing is more powerful than an idea whose time has come.
>
> *(Victor Hugo)*

Chapter 3 on page 27 already introduced methods for estimating the WCET of tasks. The most prominent and successful tool, the *aiT* WCET analyzer framework, developed by Saarland University and AbsInt Angewandte Informatik GmbH, now serves as the basis for this thesis. They key concept underlying the approach of *aiT* is abstract interpretation of a timing model of a processor and its periphery. A *timing model* used in the micro-architectural analysis of the *aiT* framework thereby is a reduced description of a processor only describing its timing behavior.

Currently, the micro-architectural analysis and the underlying timing model are hand-crafted by human experts using publicly available processor documentation and measurements as input. Thus, the process of developing a micro-architectural analysis for a new target processor is time-consuming and also error-prone. Faults are caused by several factors:

- Wrong documentation – the public available processor documentation not necessarily reflects the real behavior of a processor.
- Missing documentation – especially corner cases are often not or only partly described in the publicly available documentation.
- Ambiguity of measurement results – often, the result of a program snippet allows multiple interpretations and thus not yield the desired information.

- Limitations in inspecting processor internals – unfortunately, execution unit details and internal buses are not accessible from the outside complicating understanding corner cases.
- Human factors in the implementation of the micro-architectural analysis – human involvement in implementation always causes a big source of errors.

The increasing complexity and features used in modern hardware makes model derivation even harder.

The previous Chapter 4 on page 41 introduces VHDL, one of the most prominent hardware description languages. Modern processors and system controllers are automatically synthesized out of these formal hardware specifications. The use of HDLs eases design, simulation and verification of new processors. Besides the systems functional behavior, such specifications provide all information needed for the creation of a timing model (cf. Section 4.5.4 on page 69).

Since the final hardware (i.e. the processor that can be bought) is automatically synthesized out of the formal hardware model (normally given in form of a register-transfer level description), hardware and specification are equivalent. Thus, it seems reasonable to use the specification directly to derive timing models usable for their operation in WCET analysis. Currently, faulty documentation is directly reflected in the timing analysis and can cause faulty WCET estimates at worst. Using the same sources for timing analysis as for hardware synthesis thus eliminates an important source of errors. Often, hardware does not behave as described in the reference manuals due to specification errors for some corner cases. For timing analysis, such a behavior can only be covered with difficulty. However, using the same source of information as for synthesis would also solve this problem.

Due to size and complexity, manually examining the formal specification sources is even more complex than only looking at the processor manuals. Moreover, this would not reduce the effort nor the probability of implementation errors. Since a processor description in a hardware specification language also includes all functional details, it cannot be used directly for a micro-architectural analysis due to space limitations. Thus, there is a need for a methodology to derive timing models from a processor specification. Besides all issues discussed above, this would also reduce the effort for creating a micro-architectural analysis from months to weeks.

This chapter describes a semi-automatic method to derive timing models suitable for the usage within the micro-architectural analysis (cache/pipeline analysis) of the *aiT* WCET analyzer framework. First, transformations reducing size and complexity of a processor specification given in VHDL will be introduced. A new methodology for the derivation of timing models will be described in Section 5.2 on page 87. Section 5.3 on page 91 describes the framework supporting the

semi-automatic derivation of timing models and their automatic translation into a micro-architectural analysis that can be directly used in the *aiT* framework.

5.1 Transformations of VHDL

A VHDL specification of a complete processor is quite large, e.g., around 70000 lines of code for the LEON2 processor [Gai05]. Due to elaboration (cf. Section 4.5.3 on page 68) – a necessary process before hardware synthesis – the size of the processor model is even larger. Directly using such a model for timing analysis, i.e. within the micro-architectural analysis is not feasible in terms of computational effort and resource consumption, and thus would render timing analysis infeasible.

Nowadays processors offer a variety of different options optimizing them for usage in different fields of application. Within the area of embedded systems, and in contrast to the usage within a normal desktop-oriented area, only a small and fixed amount of options is in use. Thus, for timing analysis of embedded software, large parts of a VHDL model are not relevant due to the restricted usage. As a result, the uninteresting parts within a processor specification can be removed for timing analysis reducing size and complexity of the specification. In the following, some size and complexity-reducing transformations will be presented. More details on these transformations can be found in [Pis12].

5.1.1 Environmental Constraints

Current embedded hardware architectures offer a huge variability concerning their configurability. E.g., caches can be configured to be used as scratchpads, partial cache locking can be enabled, unified caches can be used for instructions or data only, memory regions can either use a write-back or a write-through write policy, branches can be predicted by only using static information or using dynamic information taken from large branch history tables, or the system bus can be used in a pipelined or a non-pipelined mode. All of these settings have one thing in common: they are fixed during runtime, thus rendering some parts of a hardware specification given in VHDL obsolete for timing analysis. Removing the unused parts of the specification directly contributes to the reduction in size and complexity, thereby directly reducing the complexity of a timing analysis.

Besides these configurable features, there exist a variety of events that are either asynchronous (e.g., DMA, or interrupts) or simply not predictable in their occurrence (e.g., ECC errors). A timing analysis won't deal with these events,

thus parts in the VHDL specification dealing with these events can be safely purged for timing analysis.

In order to make static timing analysis still applicable, it is necessary to make assumptions about the processor's environment. An 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 is safe to assume that those events do not happen. Based on this assumption, some signals in the VHDL description can be assumed to never change their value, so they can be *hardwired* to their default value (i.e. the value after system reset). As a result, parts of the VHDL become unreachable (i.e. control never reaches these statements), and thus, dead-code elimination can be used to remove those parts from the VHDL specification.

Asynchronous events that occur frequently (e.g., SDRAM refreshes) can be handled in a similar way. As stated in Section 4.3 on page 56, these events can be handled by statistical means. Within a VHDL specification, an occurrence of such an event triggers an input signal, which also can be viewed as hardwired for timing analysis, and thus can be treated in the same manner as non-predictable events.

The specialized operational area of processors within the field of embedded systems and the (non-)predictability of some hardware features and events offer possibilities to restrict the VHDL model of the used architecture without changing its semantics within the concrete field of application.

5.1.2 Domain Abstraction

Many modern processors offer a wide range of registers used for storing data processed by the CPU. In the field of timing analysis, the concrete value of a register at a certain program point (i.e. during execution) is often not important. E.g., for timing analysis, the resulting value of an addition is not important but the time spent to add the two involved registers. [Sic97] describes a method for separating value computations from other processor activities, thus, data paths within a VHDL model can be excluded for timing analysis. This directly implies the possibility to abstract the contents of registers in order to reduce size and complexity of the timing model.

A rather similar argumentation is valid for different memory parts, e.g., a fast and a slow memory. Due to the presence of timing anomalies (cf. Section 4.4 on page 60), memory might not be abstracted in total. But in the specialized field of application in which a processor is used, it might be sufficient for timing analysis to remember for each access, which type of memory will be accessed. Instead of storing a concrete address, an index denoting the type of memory is satisfactory for WCET analysis.

Related abstractions are also applicable to queues used in the processor (e.g., prefetch queues), where not the contents of the several queue entries is interesting for timing analysis, but the current filling level in order to determine, if there are currently any vacant slots or not.

In case of homogeneous execution units offered by many modern processors, a reduction in size and complexity of the timing model can also be achieved by this kind of abstraction. E.g., Freescale's PowerPC MPC7448 offers three identical simple fixed-point units used for simple arithmetic operations (like additions, subtractions and compare operations, [Fre05a]). For timing analysis, it might not be interesting, which concrete instruction is currently running on which concrete instance of the homogeneous execution units, instead, it is sufficient to know the amount of free/occupied units. Using this kind of abstraction requires the hardware to have no irregularities. Typical characteristics of irregular hardware architectures are constraints on instruction-level parallelism and the interconnectivity of functional units [Kä00].

All kind of transformations discussed above can be called *domain abstractions*. At the VHDL level, they can be viewed as a change of type of signals and variables in the model. Changing a domain type thus implies also a change of functors used in the model in order to preserve both, the syntactical as well as the semantical correctness of the model. The functors to be adopted need to work on the abstracted domains as well. An example is shown in Listing 5.1 on the next page showing an implementation of a simple memory controller and its access latencies for two types of memory. Changing the domain of a concrete address to address intervals implies that the compare operators used in the sample code must be adopted to return three values: yes, no and perhaps. The new implementation of the compare operators thus introduce non-determinism into the given VHDL model.

In general, this kind of transformation might introduce non-determinism into the timing model, and it is the task of the timing analysis to cope with the variety of possibilities (cf. Section 3.3 on page 39).

5.1.3 Process Substitution

For timing analysis, the internal implementation of many components is uninteresting. As stated before, correctness of the VHDL specification is assumed, and the result of some computational unit is not of interest for timing analysis and can be viewed as a kind of black box.

In terms of VHDL, the running parts within a specification are processes. Processes drive signals containing the result of its computational task (e.g., the

```
entity mem_ctrl is
    port (addr : in integer; wait_states : out integer);
end entity;

architecture arch of mem_ctrl is
    function access_time (a: integer) returns integer is
    begin
        if a >= X"0000" and a < X"1000" then --slow memory
            return 15;
        else --fast memory
            return 5;
        end if;
    end function;
begin
 P: process (addr) is
        variable latency: integer;
    begin
        latency := access_time (addr);
        if addr mod 4 /= 0 then --address not aligned
            latency = latency * 2;
        end if;
        wait_states <= latency;
    end process;
end architecture;
```

Listing 5.1 – Simple memory controller in VHDL.

address of next instructions to be fetched from memory) and activate other processes that sense at least one of those signals.

For timing analysis, the details on how a process derives its result often is uninteresting. *Process substitution* allows for replacing a concrete VHDL process implementation by a custom implementation modeling less details or using abstractions. It can further be used, if a modeling of all details results in a too large timing model rendering the resulting timing analysis computational infeasible. Thus, this transformation can be viewed as a kind of black boxing with respect to the timing behavior of the process.

This kind of transformation has been successfully applied to caches and is known as *cache abstraction* [FMWA99]. The concrete implementation of the cache is replaced by an abstracted cache domain storing only the maximal ages of all lines that are definitively in the cache (cf. Section 4.1.1 on page 43).

As for domain abstractions, depending on the implementation chosen, this kind of transformation might introduce non-determinism into the timing model,

which must be handled by the resulting timing analysis.

5.1.4 Memory Abstraction

Code and data processed by a CPU are kept in memory. For execution, instructions need to be fetched from main memory, decoded, executed and the results are written back to memory [HPG03].

Large memory arrays like the main memory blow up a timing model. Moreover, only small portions of these arrays are normally used by a program or task, for which timing analysis is made. Content of registers, addresses of memory cells being accessed by a program, and also their content can be statically computed without modeling the details of processor pipelines. A cycle-wise simulation of the processor's behavior is not necessary for the determination of these values, only the instruction semantics is needed [Sic97]. Within the *aiT* timing analyzer framework, this phase is called value analysis, which has been described in Section 3.2 on page 35.

Thus, and to make timing analysis feasible, large memory areas need to be abstracted. Otherwise, the space consumption of a resulting timing analysis will be too large. The removal of memory arrays requires changes in the VHDL model since instructions to be executed are fetched from main memory. Within the *aiT* framework, analyses are based on a common control-flow graph of the program to be executed augmented with additional informations (cf. Section 3.2 on page 35). This is also valid for the micro-architectural analysis modeling the processor behavior. In order to derive this analysis from a timing model, instructions have to be inserted into the model using the common control-flow graph. Thus, a new interface for the insertion of instructions into the timing model is needed enabling the removal of the main memory from the model. In order to achieve this, a new VHDL process is to be inserted into the model serving as interface to the control-flow graph.

Due to the value analysis, data paths can be abstracted for timing analysis and thus reduce the complexity of the timing model. The removal of these paths also requires the introduction of an interface to the annotated control-flow graph to obtain information computed by the value analysis, whenever such an information is required. E.g., for load or store instructions, the information, which address is to be accessed, requires the connection to the control-flow graph.

The removal of memory arrays and data paths from a VHDL model, and the introduction of new VHDL processes acting as interfaces to the control-flow graph is called *memory abstraction*. Due to domain abstraction and process substitution, and also due to static analysis, data access addresses of instructions

$$var\ \frac{\begin{array}{ll}\Pi[\zeta] \Rightarrow \nu := expr; & \Theta \vdash eval(expr) = u_1.\cdots.u_n \\ \Theta' = \lambda t.\begin{cases} u_1 & \text{if } t = \nu, \\ \Theta(t) & \text{otherwise.}\end{cases} & \Gamma' = \Gamma.\langle \nu \leftarrow u_1 \rangle \\ \zeta' = next(\zeta) & a_i = \langle \Gamma, \zeta', \nu \leftarrow u_i \rangle\end{array}}{(\Theta, \mathbf{\Gamma}.\Gamma, \Sigma, \zeta, \Pi) \rightharpoonup (\Theta', \mathbf{\Gamma}.\Gamma', a_2.\cdots.a_n.\Sigma, \zeta', \Pi)} \quad \text{(I)}$$

$$sig\ \frac{\begin{array}{ll}\Pi[\zeta] \Rightarrow s <= expr; & \Theta \vdash eval(expr) = u_1.\cdots.u_n \\ \Theta' = \lambda t.\begin{cases} u_1 & \text{if } t = \bar{s}, \\ \Theta(t) & \text{otherwise.}\end{cases} & \Gamma' = \Gamma.\langle \bar{s} \leftarrow u_1 \rangle \\ \zeta' = next(\zeta) & a_i = \langle \Gamma, \zeta', \bar{s} \leftarrow u_i \rangle\end{array}}{(\Theta, \mathbf{\Gamma}.\Gamma, \Sigma, \zeta, \Pi) \rightharpoonup (\Theta', \mathbf{\Gamma}.\Gamma', a_2.\cdots.a_n.\Sigma, \zeta', \Pi)} \quad \text{(II)}$$

$$bktrk\ \frac{\Sigma = \langle \Gamma, \zeta', \delta \rangle.\Sigma'\ \ \Theta' = update(\Theta^s, \Gamma.\delta)}{(\Theta, \mathbf{\Gamma}, \Sigma, \zeta_{end}, \Pi) \rightharpoonup (\Theta', \mathbf{\Gamma}.[\Gamma.\delta], \Sigma', \zeta', \Pi)} \quad \text{(III)}$$

$$stop\ \frac{}{(\Theta, \mathbf{\Gamma}, [\,], \zeta_{end}, \Pi) \rightharpoonup (\Theta, \mathbf{\Gamma}, [\,], \zeta_{sus}, \Pi)} \quad \text{(IV)}$$

Figure 5.1 – Abstract VHDL sequential execution semantics.

might only be known as safe intervals, not as single addresses. Thus, the VHDL design must be adapted to utilize the information from the value analysis instead of the real computation of addresses. For this, all places where data access addresses are generated by instructions have to be identified. At these places pseudo VHDL processes have to be added that interface with the value analysis to retrieve the previously computed intervals. Consequently, addresses must be abstracted to address intervals.

5.1.5 Abstract VHDL Semantics

Abstractions introduced into a timing model must not affect the global simulation semantics of VHDL (i.e. the process reactivation level) since this would cause a severe loss of precision [MPS09].

W. l. o. g., but for simplification, it is assumed that the VHDL model has been preprocessed avoiding any non-determinism 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. Thus, only variable and signal assignment rules need to be adjusted.

The sequential execution semantics for abstracted VHDL processes is shown in Figure 5.1 on the facing page. For simulation of these processes, it is necessary to remember points, where non-determinism was encountered. This can be achieved by managing a list of updates being applied over the initial process environment and a stack managing the decision points.

Definition 5.1.1 (Assignment)
An *assignment* $\delta \in \mathcal{D}$ with $\delta \equiv n \leftarrow v$ associates a logical name n to a value v.

Definition 5.1.2 (Process update)
A *process update* $\Gamma \in \mathit{list}(\mathcal{D})$ is a list of assignments, which is to be applied to the environment Θ of a process p (cf. Definition 4.5.2 on page 72). Θ^{s_p} denotes the environment at the start of process p.

Definition 5.1.3 (Stack)
A *stack element* is a triple $\langle \Gamma, \zeta, \delta \rangle$, where Γ is the update to be applied on the initial environment Θ^{s_p} to restore the environment before the program point where non-determinism was encountered, denoted by the program counter ζ. δ is the assignment to be applied to the resulting environment Θ to proceed from the split point on. A list of stack elements is called *stack*. [ ] is used to denote an empty stack.

Using these definitions, the context of an abstracted VHDL process can be defined.

Definition 5.1.4 (Process context)
The *context* of an abstracted process p is defined as the tuple $(\Theta, \mathbf{\Gamma}, \Sigma, \zeta_p, \Pi_p)$ where

- Θ is the environment of process p.
- $\mathbf{\Gamma}$ is a list of updates $\Gamma_1, \Gamma_2 \ldots \Gamma_n$. The last element Γ in a list $\mathbf{\Gamma}$ is denoted as $\mathbf{\Gamma}.\Gamma$.
- Σ is the stack as defined above.
- ζ_p is the program counter as defined in Definition 4.5.1 on page 72. ζ_{end} is a special address denoting that the current simulation path is at its end (i.e. process suspends here).
- Π_p is the program as defined in Definition 4.5.1.

The evaluation function $eval$ that embeds evaluation of VHDL expression, must be extended to return a list of result values in order to support non-determinism encountering the evaluation of expressions.

Signal and variable assignments are shown in the transformation rules (II) and (I), respectively. Only one of the values obtained from the evaluation function is directly used for further updates, the remaining results (if any) are pushed on the stack along with the current program counter ζ' and the update Γ that has to be applied over the initial environment Θ^{s_p} of the process p to restore the state before the actual statement.

Having processed the last statement of a process, the special program counter ζ_{end} is reached. A non-empty stack indicates that there are still execution paths that are not yet covered by the simulation. Thus, the topmost element (Γ, ζ, δ) is popped from the stack Σ and used to restore the process environment after the non-deterministic statement right before ζ. This is called *backtracking* and is depicted in inference rule (III) on page 84. $update$ is a function applying a list of assignments to an environment. Applying Γ to the initial environment Θ^{s_p} of a process p thus restores the environment *before* the point, where non-determinism encounters. Additionally applying δ to this restored environment results in the environment *after* the program point causing the split (which is exactly the program point in the process, ζ points to).

When reaching the last statement (denoted by ζ_{end}) with an empty stack $[\,]$, all possible simulation paths have been visited, i.e. simulation is done. This is covered by inference rule (IV) on page 84 and the program counter is set to ζ_{sus}.

Embedding the rules from Figure 5.1 on page 84 and Figure 4.13 on page 73 into the simulation function $\rightharpoonup_{seq}$, the transitive closure of this function

$$(\Theta, [\,], [\,], start(\Pi_p), \Pi_p) \rightharpoonup_{seq}^{*} (\Theta', \mathbf{\Gamma}, [\,], \zeta_{sus}, \Pi_p)$$

describes all execution paths through a process p.

Applying each update Γ in the list of updates $\mathbf{\Gamma}$ to the initial environment yields an environment at a suspend state (i.e. a state, where $\zeta_p = \zeta_{end}$) of the abstract execution tree (cf. Section 3.3 on page 39). The union over all resulting environments forms the input for the next process update. In order to simulate abstracted VHDL processes, the simulation rules depicted in Figure 4.14 on page 74 must be adopted to cope with sets of environments (see [Pis12, MPS09] for more details).

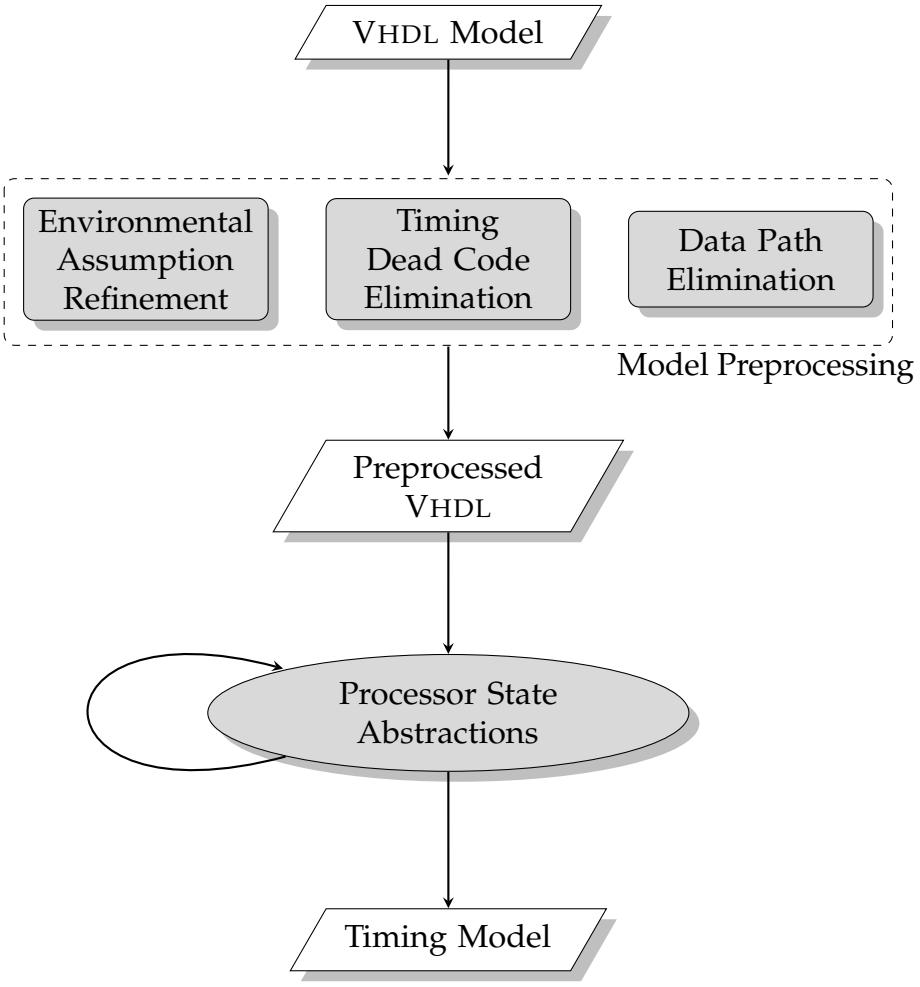


Figure 5.2 – Timing model derivation process – Methodology overview.

5.2 Derivation Cycle

As shown previously, a VHDL specification of a processor also describes its timing behavior, but it also contains all implementational details. In order to make timing analysis feasible, a VHDL specification needs to be reduced in both, size and complexity. Within the previous Section 5.1 on page 79, some transformations of VHDL have been introduced that can be used to simplify a given VHDL specification. Within this section, a methodology describing the systematic derivation of a timing model from a VHDL specification will be introduced. The resulting timing model can be directly used within the *aiT* WCET analyzer framework.

Figure 5.2 depicts the general process flow. Starting with a full VHDL specification, a systematic reduction in size and complexity must be achieved. Model

preprocessing uses statically available information on the system's environment and common abstractions to reduce the complexity of the specification. Basing on the resulting model, further processor state abstractions have to be applied iteratively until the resulting timing model is applicable within the specific field of application. Finding the balance between precision and analysis speed (i.e. applying an abstraction/transformation introduces non-knowledge into the model, which results in a more pessimistic WCET bound) thereby is the task of an advanced software engineer.

In following, the phases of the derivation cycle are briefly described, more details can be found in [Pis12, SP10].

5.2.1 Model Preprocessing

Model preprocessing is the task of reducing the size of the input VHDL model by removing parts in it that are irrelevant for the timing behavior of the system. As stated earlier, embedded systems are very specific, both in terms of hardware and software components. Mostly, some parts of a processor are not used or only used in a limited way by the application running on the system (e.g., no floating point instructions are used by the application).

Processors nowadays used in the area of embedded systems especially allow configuration of many features supported by the hardware. In a VHDL model, these configurable parts are often guarded by control signals enabling or disabling a dedicated hardware feature. Since these control signals are assigned their values during the system's initialization, remaining accesses of them are read accesses. Thus, these signals can be viewed as constant from the point of view of a timing analysis. The default values of these control signals within a processor and its corresponding VHDL model are set during system reset. Analyzing the reset behavior of a VHDL model thus leads to the initial values of these control signals. Changes of these signals can be found in the system's startup code. Using the information obtained via these two sources directly reveals, which components might be unused within a VHDL model and are thus not interesting for timing analysis. The removal of parts not reachable under a fixed mapping of control signals is called *environmental assumption refinement* (cf. Figure 5.2 on the previous page). Those parts can be safely pruned from the VHDL model and do not affect the system's behavior nor its timing within the specific filed of application.

Another optimization also aiming at reducing the size of the original VHDL model is *timing dead code elimination*. This optimization removes parts within a specification that do not contribute to the system's timing behavior and thus are not interesting for timing analysis. Timing dead code elimination (in contrast to environmental assumption refinement) is not restricted to inactive parts

or modules within a VHDL specification. E.g., a specification of a pipelined multiplier unit within VHDL describes the function of this unit in every detail including the algorithmic implementation of the unit, and also the control logic and data flow through the several pipeline stages. But for timing analysis, the algorithmic implementation and the data flow through the unit are uninteresting. Only the logic controlling the residence time of instructions in the pipeline stages, or in other words, the time spent in each stage is interesting for timing analysis.

Thus, the goal of timing dead code elimination is to restrict the VHDL model to *timing-alive* code portions. In order to achieve this, all points in the VHDL model must be identified, where an instruction can leave the processor pipeline (i.e. an instruction retires). Due to the complexity of modern architectures with branch prediction, branch folding and other early-out conditions for some instruction (cf. Section 4.1 on page 43), there might exist several locations in the model, where instruction retirement can happen. Instructions leaving the processor pipeline at the same place in the VHDL model form a so-called *instruction class*. After having identified all of these retirement locations, a backward slice (see Section 6.4 on page 140 for more details) for each instruction class can be computed. The resulting slice yields all VHDL statements influencing the instruction retirement within the model and therefore contributing to the instruction flow through the pipeline w.r.t. the instruction class. All VHDL statements not being part of this slice do obviously not influence the behavior of this instruction class for which the slice is computed. All VHDL code that is not part in the union over all these slices is said to be *timing dead* and thus can be safely removed from the VHDL model reducing its size and complexity.

The focus of timing model derivation presented in this thesis is to derive models suitable for the usage in the *aiT* WCET analyzer framework (cf. Section 3.2 on page 35). [Sic97] has shown that data paths within a processor core can be factored out. The removal of these data paths from a given VHDL model is called *data path elimination*. Fortunately, the latency of instructions is normally not influenced by the content of registers and/or memory cells they use.[1] Therefore, all data paths within a VHDL model can be removed, the values of registers and memory cells are completely computed by an external value analysis. Whenever information about the content of registers is necessary, the external analysis must be queried for this information. In contrast to the real address computation, the value analysis relies on a different domain, i.e. an interval domain representing address intervals. Due to the lack of information that is statically available, and due to the nature of abstract interpretation (cf. Section 2.3 on page 14), the removal of data paths introduces non-determinism. Consequently, this factorization of address/value information implies the need of domain abstractions from addresses to address intervals.

[1] There exists some exceptions, e.g., some early-out conditions in the multiply unit, where instructions are faster, when one of the operands contains the value 0.

5.2.2 Processor State Abstractions

Model preprocessing as described above reduces size and complexity of a given VHDL model and makes the timing model computationally feasible. The removal of timing dead code and environmental assumption refinement do not cause any non-determinism into the model, whereas the elimination of data paths causes splits due to the interconnection to the external value analysis and the accompanying switch form concrete addresses to address intervals. However, depending on the complexity of the target processor providing many performance improving features like branch speculation, speculative execution, etc., the reduction in size might not be sufficient.

Parts of the VHDL model removed so far did not contribute to the timing behavior of the processor at all. A further reduction in size can only be achieved by abstracting parts of the model relevant for its timing behavior. These parts have to be approximated rather than precisely modeled. In principle, a lack of information about the system state results in a loss of precision of the computed WCET bound. Fortunately, timing is often not affected by these abstractions, since many units work independent from any concrete input. E.g., the latency of an addition is not affected by the concrete operands' values, so simply counting remaining cycles until finished suffices for timing analysis. In contrast to this, multiplications on a multiply unit can be finished much faster, if one argument has many leading zero bits (cf. [Fre05a], Table 6-5, Integer Unit Execution Latencies). Due to the presence of timing anomalies (cf. Section 4.4 on page 60) in many modern processor cores, a lack of information here induces non-determinism into the timing model leading to a larger search space in order to guarantee a safe WCET bound. If multiplications are rare in the domain for which the timing analysis is developed for, this might be acceptable.

Section 5.1 on page 79 has already introduced transformations that can be used to further abstract the state representation of the processor. Whereas process substitution and domain abstraction can be iteratively applied until the resource consumption of the resulting micro-architectural analysis is acceptable, memory abstraction can only be applied once.

The kind in which an abstraction is to be used to derive a timing model suitable for usage in the *aiT* WCET analyzer framework cannot be described in general. This is and will ever be an engineering problem. For rather simple architectures like the ARM7 [ARM04], most parts can be modeled precisely without getting computational trouble, whereas for more complex architectures, more abstractions have to be used. Anyhow, some abstractions are widely used and are applied to nearly every processor:

- *Cache abstraction*: Process substitution is used to replace the concrete implementation of a cache by an abstract one. Details on this abstraction and

its domain can be found in [FMWA99].

- *Address abstraction*: Domain abstraction replacing concrete/precise addresses by address intervals. The use of this abstraction is mainly implied by the removal of data paths and the interconnection to an external value analysis. Switching from concrete addresses to intervals for e.g., load instructions in the load/store unit of a processor directly implies an adjustment of the interface of the bus interface unit and the memory controller unit.

Processor state abstractions certainly must not change the timing behavior of the VHDL specification. Moreover, their usage is very architecture specific in terms of applicability and necessity. As stated previously, their introduction leads to several possibilities in the simulation/behavior of a process. If the number of possibilities at a certain point in simulation is too high, the computational complexity and resource consumption might increase and timing analysis may still be infeasible. This is called *state explosion* in terms of an micro-architectural analysis [Wil12], i.e. the number of possible successor states is too high. Thus, processor state abstraction must be chosen very carefully.

The introduction of processor state abstractions leads to a loss of precise information. Whenever precise information is required, but not available for further process simulation (e.g., for the latency check in Listing 5.1 on page 82), this yields to multiple successor states. The abstract VHDL simulation rules introduced in Section 5.1.5 on page 84 deal with this non-determinism.

5.3 Derivation Framework

Previous sections already gave an overview of possible transformations on VHDL and have also proposed a workflow to derive models suitable for usage in a timing analyzer framework. Performing these steps by hand is a very time-consuming process on the one hand [The06], and error-prone on the other hand. In order to eliminate these harms, this section introduces a framework enabling the semi-automatic derivation of timing models given a formal VHDL specification of a processor.

All parts of model preprocessing as well as the processor state abstractions are based on techniques that are well known from the area of compiler construction, namely static analyses and source-to-source transformations [SWH12].

This sections describes, how to use static analyses and transformations to automate parts of the methodology introduced in the last section and to systematically derive timing models suitable for the usage in the *aiT* WCET analyzer framework. The general workflow is depicted in Figure 5.3. Starting with a

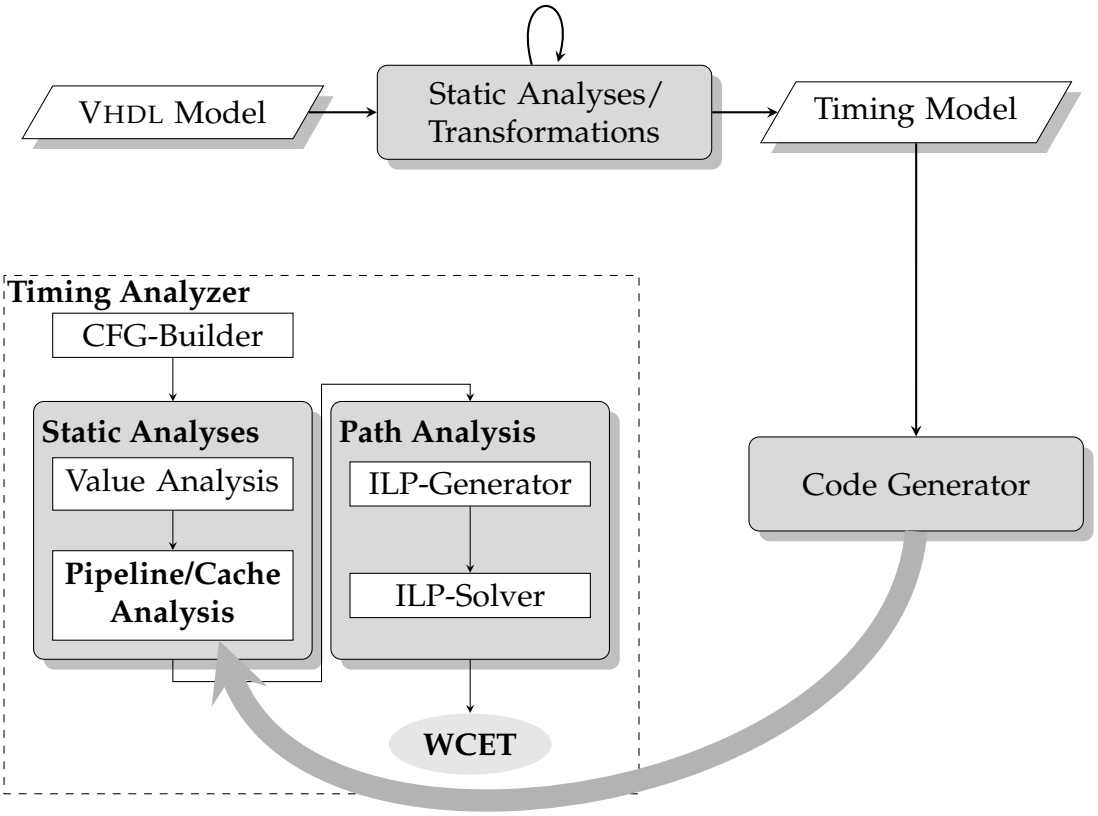


Figure 5.3 – Derivation process automation – Overview.

processor model specified in VHDL, a set of static analyses and transformations can be applied to derive the timing model. Afterwards, simulation code for the combined cache/pipeline analysis can be automatically produced using a code generator, that also respects the simulation rules of abstracted VHDL. A more detailed view on the derivation workflow can be found in [Pis12].

5.3.1 Analyses and Transformations

The methodology introduced in the last section describes several processes to derive a timing model usable for timing analysis. In order to make this process easier and more efficient, methods from the compiler design domain can be used. This section introduces these processes, a detailed view on the new VHDL analysis framework that eases the development of VHDL analyzers is given in the next Chapter 6 on page 97.

In order to simplify a processor specification given in VHDL, assignment statements to either signals or variables need to be removed. If this results in a specification containing only read-references of these signals (or variables), constant values are read. These *constant* values are the *initial values* assigned during system reset.

Within VHDL, the system reset normally is controlled externally by an input signal, called reset signal. The assertion of this signal causes the system's init-code to be executed.

Identification of default (or initial) values assigned during system reset can be automated by evaluating the code segments that are active during reset. The contents of signals/variables after these code segments have finished are the initial values. Please note that modern hardware often uses *activation chains* to initialize subcomponents, thus, the desired initial values are often not obtainable by looking at the one clock cycle, where the reset signal is asserted. E.g., memory controllers often make use of long initialization sequences to initialize the several memory chips being attached to them.

Obtaining the initial values of a system's signals and variables can be easily automated by a constant propagation analysis on those parts of the VHDL model that are active during the system's reset. A *constant propagation analysis* determines for each program point, whether or not a signal/variable has a constant value whenever execution reaches that point [NNH99]. The execution context thus has to be restricted to the active parts under system reset. Furthermore, the special VHDL signal semantics, process reactivations and simulation cycles need to be taken into account.

In the following , we will refer to this context-sensitive static analysis as *reset analysis*. For the sample implementation of a 3-bit counter given in Listing 4.3 on page 70, the reset analysis would determine the value "`000`" as the initial value for the signals `cnt` and `val`.

Timing analysis is always performed for a special field of application. The hardware used in this field is configured to exactly fit into the needs of the applications running on it. Further restrictions allow the applications to make only use of dedicated hardware features. In contrast to this, modern hardware offers a variety of features (as described in Chapter 4 on page 41). For the specialized use within embedded systems, many features are disabled (like dynamic branch prediction), or the use of them brings the whole system into a state, where guaranteeing the WCET is no longer of interest (e.g., interrupt handling in case of an error).

The assumptions on the environment of the embedded system can be automatically incorporated into the VHDL model to reduce its size. Evaluating assumptions result in rendering statements in the VHDL model as *timing dead*.

Definition 5.3.1 (Timing dead)
A statement in VHDL is called *timing dead*, if it does not contribute to the timing behavior of the processor specification.

Doing a constant propagation analysis using the externally specified assumptions yields all signals and variables that become constant under a certain assumption. Moreover, assumptions may restrict the domains of signals and variables to smaller co-domains. Evaluating conditional expressions under the co-domains can also mark code sequences as timing dead, since under a given set of assumptions, it can be guaranteed that control never reaches these program points. This process can be automated by a static analysis. In following, this analysis is referred to as the *assumption evaluation*.

Both, reset analysis and assumption evaluation, can be coupled and support the process of environmental assumption refinement. The initial values of signals and variables are determined by the reset analysis. The output of this analysis then can be used while analyzing the effect of environmental assumptions. Due to the absence of a system reset for timing analysis[2], specification parts concerned with system reset can also be removed to derive a timing model. Please not that this corresponds to the assumption "*The reset is never triggered*" and thus can also be automatically purged from the VHDL specification by using the assumption evaluator.

Reset analysis and assumption evaluation support the process of environmental assumption refinement. Also the removal of timing-dead code can be automated by means of backward slicing as defined in [Wei81]. [Sch05] describes how to derive slices using data-flow analyses.

Computing backward slices for the criterion defined in Section 5.2 on page 87, i.e. 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 behavior of the processor, and thus, can be viewed as timing dead. For timing model derivation, these statements can safely be purged from the specification.

Analyses described so far are used for supporting the model preprocessing steps shown in Figure 5.2 on page 87. Supporting the data-path elimination and the processor state abstractions requires the introduction of abstractions into the VHDL model. The process of inserting abstractions can also be supported by transformations tools.

Changing the type of a signal or variable to an abstract domain is a very common abstraction used for introducing non-knowledge into a VHDL model, e.g., for

[2]For timing analysis, the absence of faults is mandatory, since every failure results in a system state, where timing bounds are no longer of interest.

imprecise or unknown addresses, address abstraction is used. Naturally, changing the domain (i.e. the type) of a signal/variable also induces a change of all functors used by the signal/variable (cf. Section 5.1.2). A similar transformation is the process substitution as described in Section 5.1.3 that replaces a dedicated process by a more abstract implementation which has to be provided by the user.

Both transformations can be viewed as *source-to-source* transformations. More details on these transformations and their implementations can be found in [Pis12].

Combining these analyses and transformations allows for automating the process of timing model derivation in order to obtain a model that is feasible for timing analysis. In general, the iterative workflow to reduce the complexity of a VHDL model can be viewed as:

1. Exploration of the VHDL model: an engineer determines a possible abstraction.
2. Abstraction-specific analyzes: according to the chosen abstraction, some specific analyzes might be necessary in order to safely transform the model.
3. Model transformation: induce the chosen abstraction into the model.
4. Timing dead code elimination: Many transformations and analyzes result in dead code parts within the model that can be purged from it.

Applying these steps iteratively, the code generator described in the next section can be used to generate a combined cache/pipeline analysis suitable for the usage within the *aiT* WCET analyzer framework.

5.3.2 Code Generation

After applying a set of transformations and abstractions to the initial VHDL model, a timing model is derived. This timing model can be used to directly generate C++-code respecting the abstract simulation rules described in Section 5.1.5 on page 84. The generated code provides simulation code for exactly one simulation cycle (cf. Section 4.5.4 on page 69), i.e. exactly one clock cycle. Process reactivations resulting from VHDL delta cycles are unrolled and therewith respected by the code generator.

Due to abstractions induced into the model, the simulation code provides a set of resulting processor states based on one initial state. This is shown in Figure 5.4 on the next page showing the evolvement of final states for the simple memory controller depicted in Listing 5.1 on page 82. If the input address that is to be checked is unknown, four possible successor states exists. The edges in

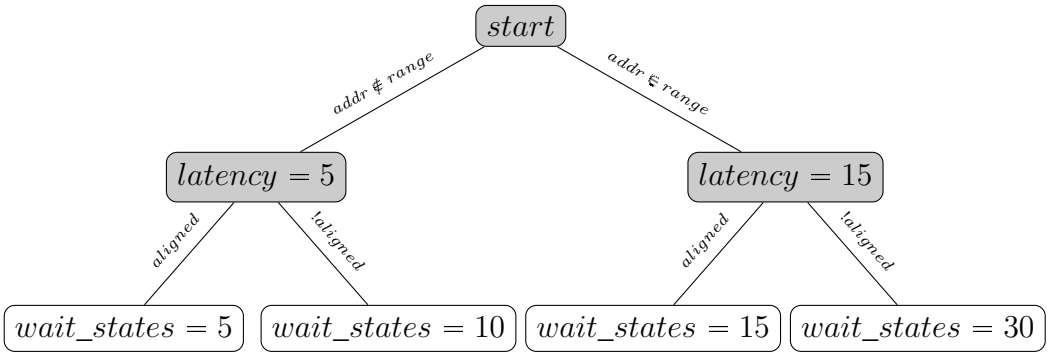


Figure 5.4 – State altering of the sample memory controller from Listing 5.1 on page 82 and unknown input.

the figure are labeled with the basis of decision. The tree-like representation is called *abstract computation tree* (cf. [Sch98]).

Due to the special two-level semantics of VHDL, code generation has to cover both, process execution as well as the re-evaluation level. In general, every VHDL model can be directly simulated. Due to certain loss of information (some information is statically not available), the resulting simulation code won't be feasible in terms of computational effort.

[Mak07, MPS09] describes the implementation of a code generator for abstracted VHDL models. Besides the simulation cycle code allowing for a cycle-wise update of the model, additional constructs are generated supporting a seamless integration into the *aiT* WCET framework. In order to enable code generation for a given VHDL model, processor state abstractions are not mandatory. Only memory abstraction as described in Section 5.1.4 is mandatory due to the insertion of pseudo processes serving a the interface to *aiT*'s shared control-flow graph (cf. Section 3.2).

Since the generated simulation code automatically follows all possible paths within the execution tree, timing anomalies are safely found and the resulting WCET is guaranteed to be safe. Due to abstractions and missing knowledge on the inputs, not necessarily all paths within an execution tree need to correspond to execution traces being observable in reality. E.g., if there are some exclusive conditional paths in the analyzed application and due to some unknown memory accesses, all paths are combined although some of them are mutually exclusive. Such situations can lead to overestimations of the real WCET if such a combined path leads to the highest WCET. [Ste06] describes a method that allows for precise analysis of exclusive paths.

6

Static Analysis of VHDL

> The man with a new idea is a crank until the idea succeeds.
>
> *(Mark Twain)*

This chapter describes the transformation of an architecture design given in VHDL into a semantically equivalent sequential program. The transformation is embedded into a framework supporting the specification of data-flow analyzers. Due to the theory of abstract interpretation underlying these data-flow analyzers, their results can be proven to be correct. Also the soundness of the transformation to obtain the sequentialized program is shown.

The design of the proposed framework focuses on simplicity, efficiency, flexibility and reliability. The utilizability of the framework is shown by different analyses, which are bundled to support the process of timing model derivation as described in the previous chapter. In detail, the following parts of the derivation framework will be presented:

- *reset analysis*,
- *assumption evaluation*, and
- *static backward slicing*.

Also the correctness of these analyses will be proved.

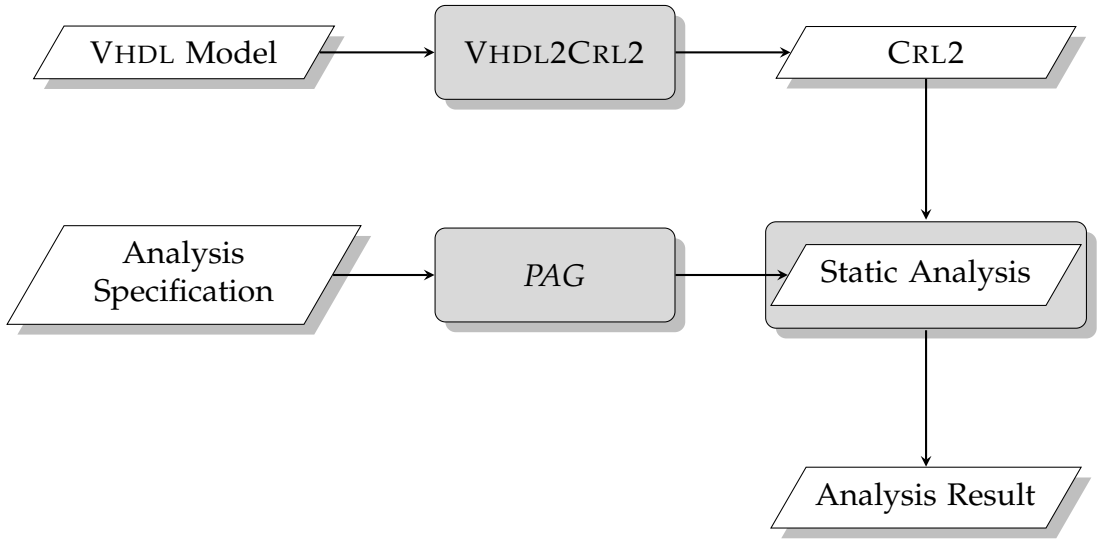


Figure 6.1 – VHDL analysis framework – Structure.

6.1 Analysis Framework

This section introduces a new framework allowing for static analyses of VHDL using abstract interpretation. The overall structure of the framework is depicted in Figure 6.1. Starting with an architecture specification given in VHDL, the model is transformed into an equivalent intermediate format, called CRL2. The transformation tool named VHDL2CRL2 performs all the required renaming for unifying names, instantiates referenced components, and wires structural descriptions as described in Section 4.5.3. In order to support the process of timing model derivation (cf. Section 5.3 on page 91), the framework focuses on efficient and reliable analyses of VHDL [SP07]. Thus, *PAG* is used to generate efficient static analyzers from concise high-level specifications that work on an intermediate format, called CRL2. The analyzer framework offers the possibility of analyzing open, as well as closed designs[1]. The generated analyzers are so called data-flow analyzers that rely on a control-flow graph given in an intermediate format.

In the following, *PAG* and the intermediate format are described in more detail. Then, the principal mapping of VHDL constructs to CRL2 primitives is detailed. Section 6.1.4 describes how the special two-level semantics of VHDL can be transformed into a one-level semantics, which enables the use of *PAG* to automatically generate program analyzers from high-level specifications. Moreover, additional constructs introduced to support analysis of open designs as well as synchronous designs (including multiple clock domains) are described.

[1] A design is called *open* (or *non-closed*) if the design refers to some external input signals.

```
SET
    numlist = list (unum)

DOMAIN
    numberset = set (unum)
    numbers = lift (numberset)
```

Listing 6.1 – Example of set and lattice specifications in *DATLA*.

6.1.1 Program Analyzer Generator

Analogous to *flex* and *bison* used to generate lexers and parsers, the program analyzer generator (*PAG*) is used to automatically generate data-flow analyzers from high-level specifications. Its theoretical background is detailed in [Mar95, Mar99]. The generated analyzers are efficient allowing not only prototyping of analyses, but also using them within the final software [AM95, Mar98].

The input for *PAG* is the specification of an analysis given in high-level specification languages, namely *DATLA* and *FULA*. The specified analysis thereafter is available via a C-function, whose input is only the control-flow graph of the program to be analyzed. The result of the analysis is a data structure mapping the nodes of the control-flow graph to the computed data-flow value. Computed information may then be used for optimizations.

An analysis specification for *PAG* consists of two parts: the first part contains the definition of sets and lattices, the second part contains the problem specification, transfer functions and support routines. For the specification of sets and lattices, *DATLA* is used. An example specification is given in Listing 6.1. Here, a set of lists over unsigned numbers is defined. Within the `DOMAIN` section, a power set of the unsigned numbers with inclusion ordering is defined using $\varnothing$ as bottom and the set of unsigned numbers as top element for the defined lattice. `lift` extends the argument lattice by additional top and bottom elements while preserving the ordering of the argument lattice.

Listing 6.2 on the following page lists an example problem specification given in *FULA*. Besides the direction of the data-flow analysis (i.e. forward or backward), the analysis carrier denoting the underlaying lattice of the analysis, as well as the initial values need to be specified. Moreover, a functor to combine two data-flow values at control-flow joins, a widening operator (cf. Section 2.4.1 on page 23) and a functor for checking the equality of two data-flow values need to be given. Both specification languages, DATLA and FULA, are detailed in [TMAL98].

Analyzers generated by *PAG* are data-flow analyzers, thus, all computations are based on a control-flow graph. Therefore, *PAG* is not restricted to any input

```
PROBLEM Dominators
    direction:  forward
    carrier:    numbers
    init:       bot
    init_start: lift(bot)
    combine:    comb
    widening:   wid
    equal:      eq

RETURN
    ...

TRANSFER
    ...

SUPPORT
    ...
```

Listing 6.2 – Example of a problem specification in *FULA*.

language, but the input programs need to be translated into a control-flow graph. Within *PAG*, this translation is made within the so-called *frontends*. *PAG* already supports a couple of frontends, but a user may also write his own frontend according to his wishes and demands. The most widely used frontend is the CRL2 frontend, which also serves as the internal exchange format for the *aiT* toolchain (cf. Section 3.2 on page 35). A more detailed view on CRL2 will be given in the next section.

Analyzes generated by *PAG* support interprocedural analysis based on an interprocedural control-flow graph, the supergraph (cf. Definition 2.4.7 on page 26). The underlying principle of constructing a supergraph from several control-flow graphs by adding call and return nodes and connecting these special nodes via special edges was introduced by [SP81]. Listing 6.3 on the facing page gives a recursive implementation of the factorial function, the corresponding supergraph is shown in Figure 6.2 on page 102. At return nodes in the combined supergraph, a special function $\mathit{return}\colon L \times L \to L$ is used instead of the usual combine operator defined on a lattice L. Within *PAG*, this function is called to combine data-flow information over the local edge and the return edge.

To keep different call paths apart, each node within the supergraph is annotated with an array of data-flow elements from the analysis domain; the size of each array (and thus the number of disjoint contexts) is fixed. *PAG* calculates a *mapping*, i.e. a set of functions describing for each call within a supergraph how the data-flow elements of the calling procedure are connected to the locations of the

```
int fac (int n)
{
    if (n < 2)
        return 1;
    else
        return n * fac (n - 1);
}

void main (int argc, char** argv)
{
    int n, res;
    scanf ("number: %d", &n);
    res = fac (n);
    printf ("result: %d\n", res);
}
```

Listing 6.3 – Recursive implementation of the factorial function.

called procedure. Each field in the array is called a *context*. Within a procedure, this mapping is fixed, i.e. context is not switched here. The number of contexts (and therewith the size of the data-flow arrays) is not fixed by *PAG* and can be varied by setting the maximum length for the call strings. The sample supergraph shown in Figure 6.2 on the following page uses a maximum call string length of one, i.e. only the two calls of the factorial function can be distinguished, all recursive calls are mapped into a collecting context. More details on call strings and the static call-graph approach can be found in [TMAL98].

6.1.2 Control-flow Representation Language

The control-flow representation language (CRL2) was developed as an intermediate format simplifying analyzes and optimizations on a control-flow graph [Lan98]. Originally, it was developed at Saarland University and AbsInt Angewandte Informatik GmbH and is for now totally maintained by AbsInt.

Based on the assumption that the control-flow graph of a program is always well-formed, a CRL2 description is text-based and its structure is hierarchically organized into operations, instructions, blocks and routines.

Listing 6.4 on the next page shows a concurrent signal assignment statement taken from a VHDL processor specification. The corresponding CRL2 description is depicted in Listing 6.5, and the control-flow graph is shown in Figure 6.3 on page 104. Operations are grouped to instructions, which in turn are grouped to basic blocks. Basic blocks are connected via edges of different types to describe

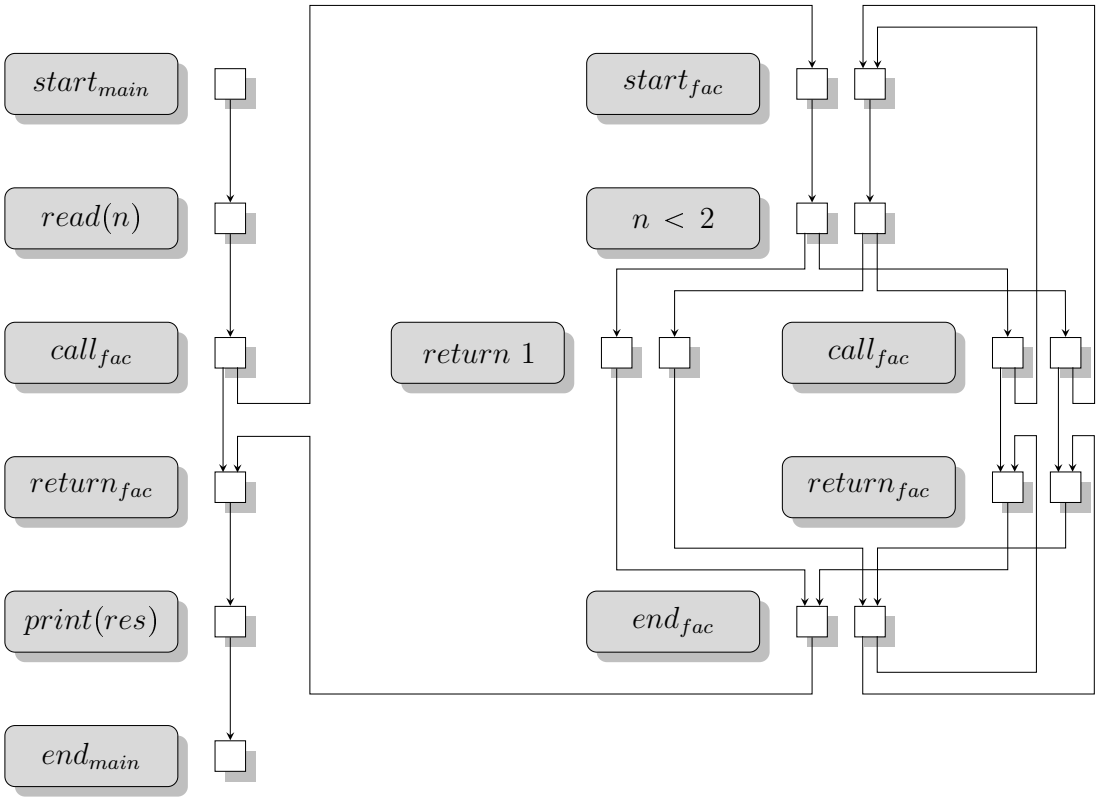


Figure 6.2 – Interprocedural supergraph.

the structure of programs. Beside *normal* edges, also *true* and *false* edges exist to describe the successors of conditional expressions. Edges between two routines indicate a call dependency. Each call is represented through *call/return* blocks in the calling routine. Use of these call/return blocks and the corresponding *call/return* edges ease interprocedural analyzes [SP81] and can be viewed as place holders for the branch to the called routine.

Nevertheless, CRL2 is a flexible language by virtue of an attribute-value concept: each element of a CRL2 description can be easily extended by attributes coding arbitrary information. The language provides a set of predefined advanced data structures like key-value-maps, hashes, lists and sets that can be further

```
IF_InstrRegA_Input <=
    IF_InstrRegB when IF_LoadStageA_WithStageB = '1' else
    IC_DataOut( 31 downto  0 ) when IF_InstrCounterReg( 2 ) = '1'
       else
    IC_DataOut( 63 downto 32 );
```

Listing 6.4 – Sample VHDL code snippet.

```
routine r4: file="sources/Dlx.vhd", line=0x346, name="process_5" {
 block b8 (start) {
  edge e830 -> b833;
 }
 block b9 (end);
 block b833: bbid=0xe {
  edge e834 (true) -> b834;
  edge e838 (false) -> b835;
  instruction i831 {
   operation o832 "IF_LoadStageA_WithStageB = \'1\'":
     cat=1*{ifstatement}, defs=0*{}, op_id=0x2f,
     uses=1*{'IF_LoadStageA_WithStageB'};
   }
 }
 block b834: bbid=0xf {
  edge e849 -> b9;
  instruction i835 {
   operation o836 "IF_InstrRegA_Input <= IF_InstrRegB":
     cat=1*{signalassignment}, def=1*{'IF_InstrRegA_Input'},
     op_id=0x1d, uses=1*{'IF_InstrRegB'};
  }
 }
 block b835: bbid=0x10 {
  edge e842 (true) -> b836;
  edge e846 (false) -> b837;
  instruction i839 {
   operation o840 "IF_InstrCounterReg(2) = \'1\'":
     cat=1*{ifstatement}, defs=0*{}, op_id=0x2f,
     uses=1*{'IF_InstrCounterReg'};
  }
 }
 block b836: bbid=0x11 {
  edge e850 -> b9;
  instruction i843 {
   operation o844 "IF_InstrRegA_Input <= IC_DataOut(31 downto 0)":
     cat=1*{signalassignment}, def=1*{'IF_InstrRegA_Input'},
     op_id=0x1d, uses=1*{'IC_DataOut'};
  }
 }
 block b837: bbid=0x12 {
  edge e851 -> b9;
  instruction i847 {
   operation o848 "IF_InstrRegA_Input <= IC_DataOut(63 downto 32)":
     cat=1*{signalassignment}, def=1*{'IF_InstrRegA_Input'},
     op_id=0x1d, uses=1*{'IC_DataOut'};
  }
 }
}
```

Listing 6.5 – CRL2 representation of Listing 6.4 on the facing page.

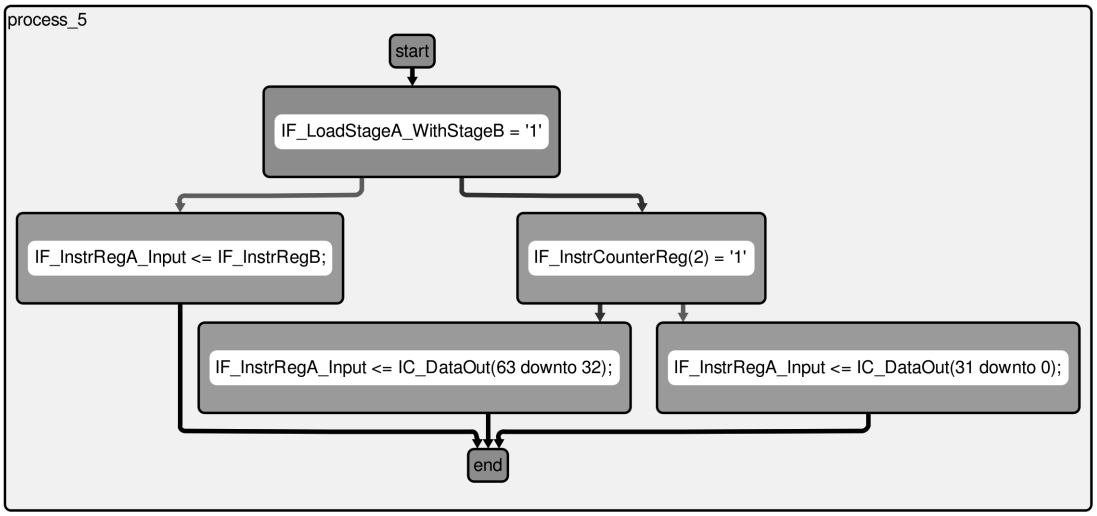


Figure 6.3 – Control-flow graph of the CRL2 example from Listing 6.5.

nested. To process programs represented in CRL2, AbsInt offers a huge library providing a large interface. Also a frontend for *PAG* (cf. Section 6.1.1 on page 99) is available.

6.1.3 Basic Mapping

As stated before, the static analysis framework for analysis of VHDL models makes use of *PAG* to efficiently generate program analyzers from high-level specifications. The analyzers rely on a control-flow description of the program to be analyzed given in CRL2. Thus, a VHDL specification needs to be expressed in CRL2 while preserving the semantics of VHDL. This section gives the basic mapping of VHDL constructs to constructs from CRL2.

The basic mapping of VHDL constructs is shown in Table 6.1 on the facing page. For sequential statements that form the body of processes, functions and procedures, the control-flow reconstruction rules described in [CFS94] and [NNH99] can be easily applied. Each statement is mapped to a single operation within a CRL2 instruction. Change-of-flow constructs (i.e. if statements, switch statements, etc.) form the basic block structure of the CRL2 representation.

Consequentially, processes are mapped to routines, as well as functions and procedures. Each routine is extended by unique start and end nodes marking the entry and the exit of the routine, respectively. For concurrent signal assignments and concurrent procedure calls, their transformation into fully qualified processes is described in the IEEE VHDL standard [IEE87]. Thus, both constructs are also mapped to CRL2 routines.

VHDL element	CRL2 element
Sequential statement	Operation + instruction
Process, Function, Procedure, Concurrent signal assignment, Concurrent procedure call	Routine
Function call, Procedure call	Routine call
Loop	Loop transformation + call

Table 6.1 – Mapping of VHDL constructs to CRL2 constructs.

To enhance the analyzability of loops [MAWF98], and thus to achieve more precise results, loops have to be transformed into recursive procedures. In order to achieve this, for-loops have to be converted into semantically equivalent while-loops. The resulting while-loop can be transformed into the desired recursive procedure by applying the loop transformation as described in Section 3.2 on page 35.

Function and procedure calls are represented via call and return nodes in CRL2. Additionally, a call edge connecting the caller's call node and the callee's start node is inserted. Also, a return edge from the callee's end node to the caller's return node is added to ease analysis of interprocedural control-flow [SP81].

This basic mapping of constructs suffices to transform every process in VHDL to its semantically equivalent representation in CRL2. To further enhance the analyzability of the VHDL model, additional transformations can be applied (while keeping the semantic equivalence). Thus, switch statements within the body of processes, functions or procedures are transformed into equivalent if-then-else cascades.

To support and ease analyses, additional attributes can be added to the CRL2 description of a VHDL model. These attributes also code information that is missing due to limitations of the available CRL2 constructs. The basic mapping rules map processes, procedures, as well as functions to routines in CRL2. When only looking at the new representation, the correlation to the corresponding VHDL part is missing. Thus, an attribute is added to mark if a routine represents a process, a procedure or a function. Attributes are further used to ease access

to used and defined variables, signals and constants at a certain statement. Also the type of a statement (i.e. a classification if the statement is a signal assignment, a variable assignment, a change-of-flow statement, etc.) is expressed as an attribute in CRL2. Use of attributes in CRL2 is not limited to hierarchical constructs, but can also be used to classify edges connecting nodes. Edges are thus classified as normal, true, false, call and return edges.

Using attributes to express certain properties of statements and constructs of VHDL allows static analyzers not to rely on parsing VHDL itself in order to obtain provable correct results. E.g., the attributes coding the information on used and defined identifiers dramatically ease the development of a reaching definition analysis. Paired with the statement classification attribute mentioned above, a reaching definition analysis can be implemented without any further knowledge of the semantics of VHDL. An example showing some of the attributes mentioned so far is shown in Listing 6.5 on page 103. The statement `IF_InstrRegA_Input <= IF_InstrRegB;` in line 18 is categorized as a signal assignment using the identifier `IF_InstrRegB` and defining the signal `IF_InstrRegA_Input`. In order to ease the implementation of static analyzers like an available expression analysis and to not rely these analyzers being able to parse VHDL expressions with their language specific operator precedence, an attribute can also store the prefix notation of the underlying expression.

Whereas this section has described the basic mapping of constructs from one language to another one being more suitable for static analyses, the scope of mapping rules is restricted to the process execution level with its sequential semantics. The next section details on how to deal with the two-level semantics specific for hardware description languages.

6.1.4 Transformed VHDL Simulation Semantics

The semantics of VHDL is special compared to other widely spread languages. Compared with imperative languages, the VHDL semantics is two-staged (cf. Section 4.5 on page 62). At the first level, there is the parallel execution of processes, and the second level is signal updating and process reactivation.

In order to enable use of *PAG* and CRL2, which are both designed to analyze and represent structured control-flow based on imperative languages, the two semantical levels of VHDL need to be joined. This section describes, which semantic properties allow joining both semantic levels to one level, and thus, enable use of *PAG* and CRL2 to implement efficient static analyzers on VHDL.

The subsequent definitions describe the semantics of constructs used to join the two semantical levels of VHDL. The definitions are based on the semantic description of VHDL that is given in Section 4.5.

First, it is necessary to extend the definition of an environment for the simulation of VHDL models. The standard offers a variety of functions for accessing the history of transactions of signals connecting the different processes. Since this history is relevant for process reactivation, and thus relevant for the behavior of a VHDL model, the definition of environments is to be updated as follows.

Definition 6.1.1 (Environment)
The environment $\Theta \in Env$ for the transformed VHDL semantics is a mapping from logical names to values. A logical name for the transformed simulation semantics can be either a variable v, a signal s, a scheduled signal $\bar{s}$, or a preceding signal $\underline{s}$.

Thus, access to the previous value assigned to a signal s is possible via $\underline{s}$. Using this special name allows for access of the transaction history of a signal, and thus allows for checking the occurrence of a new event. This is necessary for modeling the simulation cycle that advances simulation time for a given VHDL model. Please note that for the remainder of this thesis, we will use $identifier$ to denote the set of logical names.

Joining the two semantical levels requires the following property of the VHDL language: Variables in a VHDL model are process-local, and thus visible solely in the defining process. All processes in a model run in parallel. Any assignment to a variable immediately takes effect. In contrast, signal assignments are delayed to the point, where all processes have finished their execution. Only signals are used to interconnect processes in a model. These semantics directly induce that there are no side effects in the execution of different VHDL processes in a model. Thus, it is possible to serialize the execution of processes without changing the semantics of the whole VHDL model. As a result, it is possible to choose an arbitrary execution order among all processes, and sequentially execute the processes in this chosen order.

Therefore, a special simulation process, called $simul$, has to be added to the given model. The task of this special process is to model the execution order that has been chosen for process execution. Therefore, a special reactivation statement with its semantics has to be defined.

Definition 6.1.2 (Reactivation statement Υ)
The semantics of a *reactivation statement* $\Upsilon : \mathbb{L} \rightarrow Env \rightarrow Env$ for a process label $l \in \mathbb{L}$, with $\boldsymbol{\rho}(l) = (\zeta_l, \Pi_l, \omega_l)$ and an environment $\Theta \in Env$ is defined as

$$\Upsilon(l)(\Theta) = \Theta', \text{ with } (\Theta, start(\Pi_l), \Pi_l) \rightharpoonup_{seq}^{*} (\Theta', \zeta_{sus}, \Pi_l)$$

The reactivation statement resets the program counter of a given process to its first statement, and thus reinvokes the given process. This is safe since the models of interest for timing model derivation are restricted to have their

wait-statement (i.e. the statement that suspends a process) at the end of their sequential code.

Use of reactivation statements sequentializes the process execution and allows for choosing a fixed process ordering. To model the change of signal values, a special synchronize statement has to be introduced. This statement models the part where scheduled transactions become visible in a VHDL simulation.

Definition 6.1.3 (Synchronize statement Ω)
The semantics of the *synchronize statement* $\Omega\colon Env \to Env$ for a given environment $\Theta \in Env$ is defined as:

$$\Omega(\Theta) = \lambda t. \begin{cases} \Theta(\bar{s}) & \text{if } t = s, \\ \Theta(\bar{s}) & \text{if } t = \bar{s}, \\ \Theta(s) & \text{if } t = \underline{s}, \\ \Theta(t) & \text{otherwise.} \end{cases}$$

Use of reactivation statements and the use of the synchronize statement allow for joining process reactivation and signal updating to one semantic level, which already can be expressed in CRL2 using the basic mapping rules from Section 6.1.3 on page 104. The missing part in the VHDL semantics is the level of process reactivation.

A process is to be reinvoked if, and only if, at least one of the signals being part of the its sensitivity list changes its value. Within the terminology of VHDL, assigning a value to a signal s is called *scheduling a transaction*. Please note that also the assignment of an identical value schedules a transaction on the signal, but for process reactivation, the value assigned to a signal must have changed. This is called an *event*. In VHDL, the presence of transactions, events, and other properties of signals can be checked using signal attributes. In order to check if there is a pending event for any signal s in the current simulation cycle (i.e. under a certain environment), the attribute $s'event$ is to be used. This denotes the presence or absence of an event. Beside this attribute, several other attributes exist (see [Ash01, chap. 5] for a complete listing of available attributes). The semantics of the event attribute can be defined as follows:

Definition 6.1.4 ($event$ attribute)
The signal attribute $event$ for a given signal s under the environment Θ is defined as

$$s'event = \begin{cases} true & \text{if } \Theta(s) \neq \Theta(\underline{s}), \\ false & \text{otherwise.} \end{cases}$$

Using this attribute allows to express process reactivation as the disjunction of events on the signals of a process' sensitivity list. The reactivation of a process

```
//create initial starting label
create ("label start:")

// create if-guard and process revocation statement
forall l ∈ 𝕃, with ρ(l) = (ζ_l, Π_l, ω_l)
    create ("if (Ξ(l)(Θ) = true) then")
    create ("    Θ = ϒ(l)(Θ);")
    create ("end if;")

// create synchronization statement and delta delay
create ("Θ = Ω(Θ);")
create ("if ((⋁_{l∈𝕃} Ξ(l)(Θ)) = true) then")
create ("    goto start;")
create ("end if;")
```

Listing 6.6 – Construction algorithm of the simulation routine.

(via the special reactivation statement) can now be guarded by the following condition.

Definition 6.1.5 (If-guard statement Ξ)
Given a process label $l \in \mathbb{L}$, with $\boldsymbol{\rho}(l) = (\zeta_l, \Pi_l, \omega_l)$, the *if-guard statement* $\Xi\colon \mathbb{L} \to Env \to \mathbb{B}$ for the process $\boldsymbol{\rho}(l)$ is constructed as logical disjunction of the $event$ attribute of each signal contained in ω_l:

$$\Xi(l)(\Theta) \equiv \Theta \vdash \bigvee_{s \in \omega_l} s'event$$

The combination of process reactivation statements, if-guard statements and the synchronize statement can be used to express the two semantic levels of VHDL with only one level, which enables a direct use of *PAG* to derive static analyzers from high-level specifications.

The construction algorithm of the special simulation process $simul$ for a given environment Θ is given in Listing 6.6. First, a start label is created, which is part of modeling the delta-delay. Then, for each process label l in the set of process labels $\mathbb{L}$, an if-statement with the special if-guard is created. The process $\boldsymbol{\rho}(l)$ is only to be reinvoked if at least one signal s contained in the sensitivity list ω_l changes its value (i.e. an event on at least one signal must be pending). If this is the case, the process labeled l must be reinvoked, so the program counter ζ_l has to be changed to the start of the process' statement list Π_l and the process has to be executed once again. Otherwise, no signal of the process' sensitivity list ω_l has been changed, so the process is not to be reinvoked and the corresponding program counter ζ_l remains unchanged (i.e. $\zeta_l = \zeta_{sus}$).

By this code, a sequential update order and the conditions for process reactivation are already modeled. Making scheduled transaction visible is done by the synchronize statement Ω, but modeling the VHDL delta-delay (cf. Section 4.5.4 on page 69) requires at least one event in the set of all signals at least one process is sensitive to. This set can be easily constructed via the disjunction

$$\bigvee_{l \in \mathbb{L}} \Xi(l)(\Theta)$$

If this logic expression under the updated environment $\Theta' \equiv \Omega(\Theta)$ evaluates to $true$, at least one process must be reinvoked. This can be guaranteed by creating a *goto* statement jumping to the beginning of the created simulation process.

The result of applying this construction algorithm to the sample VHDL code shown in Listing 4.3 on page 70 is depicted in Listing 6.7 on the facing page. The VHDL model consists of two processes, `P1` and `P2`. The first process senses the clock signal `clk` and the reset signal `rst`, and thus, the if-guard statement for the first process must check these two signals for containing an event in the current simulation cycle. If one of these signals actually have changed its value, `P1` has to be reinvoked. Analogously, process `P2` senses the signal driving the counter value `cnt`, so this process must be reinvoked if the value of the counter changes.

If at least one of the signals mentioned changes its value, another delta cycle within the current simulation cycle is necessary. So, if, after making all scheduled transactions visible, one of the signals `clk`, `rst` or `cnt` currently contains an event, the goto statement is taken to reinvoke the required processes.

Modeling Simulation Time

The focus of this thesis is the static analysis of synchronous VHDL circuits in order to ease the process of deriving timing models usable for timing analysis. In a synchronous design, a clock signal triggers the integrated circuit to go from a well defined and stable state to the next one. On an event of the clock signal, all signals must reside stable in either the high or low state. Between two consecutive events of the clock, the signals are allowed to change and may take any intermediate state. Thus, especially the advance of simulation time and the state of the model at a certain simulation cycle is interesting. Modeling the frequent change of a clock signal is not possible using the synthesizable subset of VHDL, and therefore, the clock signal (and with it the simulation time) is an external signal.

Nevertheless, timing is relevant for static analyses, and thus, the frequent change must be part of the static analysis framework presented here. Hence, the clock signal of the synchronous circuit is to be modeled via a special clock process modeling rising and falling edges of the clock signal.

```
label start:

if (Θ(clk) ≠ Θ(clk)||Θ(rst) ≠ Θ(rst)) then
    Θ = ϒ(P1)(Θ);
end if;

if (Θ(cnt) ≠ Θ(cnt)) then
    Θ = ϒ(P2)(Θ);
end if;

Θ = Ω(Θ);
if (Θ(clk) ≠ Θ(clk)
    ||Θ(rst) ≠ Θ(rst)
    ||Θ(cnt) ≠ Θ(cnt)) then
    goto start;
end if;
```

Listing 6.7 – Simulation routine for 3-bit counter example from Listing 4.3.

Definition 6.1.6 (Rising clock $\Gamma^{\uparrow}$, falling clock $\Gamma^{\downarrow}$)
Given the name of a clock signal c. The semantics of a *rising clock* update $\Gamma^{\uparrow}\colon \mathit{identifier} \to \mathit{Env} \to \mathit{Env}$ is defined as

$$\Gamma^{\uparrow}(c)(\Theta) = \lambda t. \begin{cases} 1 & \text{if } t = c, \\ 1 & \text{if } t = \bar{c}, \\ 0 & \text{if } t = \underline{c}, \\ \Theta(t) & \text{otherwise.} \end{cases}$$

Analogously, the semantics of a *falling clock* update $\Gamma^{\downarrow}\colon \mathit{identifier} \to \mathit{Env} \to \mathit{Env}$ is defined as

$$\Gamma^{\downarrow}(c)(\Theta) = \lambda t. \begin{cases} 0 & \text{if } t = c, \\ 0 & \text{if } t = \bar{c}, \\ 1 & \text{if } t = \underline{c}, \\ \Theta(t) & \text{otherwise.} \end{cases}$$

Using these definitions, we can define the clock process for the static analysis framework.

Definition 6.1.7 (Clock process κ)
Given the simulation process simul, the semantics of the recursive *clock process* $\kappa\colon \mathit{Env} \to \mathit{Env}$ is defined as

$$\Upsilon(\kappa) = \big(\Upsilon(\kappa) \circ \Upsilon(\mathit{simul}) \circ \Gamma^{\downarrow}(\mathit{clk}) \circ \Upsilon(\mathit{simul}) \circ \Gamma^{\uparrow}(\mathit{clk})\big)$$

The clock process is modeled as a recursive process. This allows static analyses based on this framework to make use of *PAG*s static call-string approach (cf. Section 6.1.1 on page 99). Use of the call-string approach allows for separating several clock cycles via different contexts that can be controlled by the static analyzers. So, a cycle-wise analysis of the behavior of a VHDL model is possible.

Modeling Open Designs

So far, the analysis framework presented in this section can only cope with closed designs. To analyze open designs (i.e. designs where transactions of at least one signal besides the clock signal are scheduled from outside the VHDL model), the framework has to be extended. Therefore, a new process called *environment* will be introduced. Signal transactions that are not part of the analyzed VHDL model have to be handled in this process.

Definition 6.1.8 (Environment process ϖ)
The semantics of the *environment process* $\varpi\colon list(\mathcal{D}) \to Env \to Env$ for a given list of assignments $\Gamma \in list(\mathcal{D})$ and an environment $\Theta \in Env$ is defined as:

$$\varpi(\Gamma)(\Theta) = update(\Theta, \Gamma)$$

The list of assignments applied to the environment must be given externally. Listing 6.8 on the facing page gives the pseudo code for generating the simulation process for open VHDL designs. The process reactivation statement for the environment process ϖ must not be guarded with $\Xi(\varpi)$, since, from the design's point of view, external signals may change their value in each delta cycle.

Using this construction algorithm and the basic mapping rules from the previous section allow for expressing any synchronous VHDL model with constructs of the CRL2 language in a semantically equivalent way. The semantics of the additional processes was given in this section. Applied to the example given in Listing 4.3 on page 70, the equivalent sequentialized representation as a control-flow graph is shown in Figure 6.4 on page 114.

The CRL2 representation of the VHDL model can be directly processed by static analyzers created by *PAG*. In order to show the effectiveness and the correctness of the presented static analysis framework, static analyses in forward and backward direction as well as the combination of analyses will be presented in the next sections.

```
//create initial starting label
create ("label start:")

// create if-guard and process revocation statement
forall l ∈ 𝕃, with ρ(l) = (ζ_l, Π_l, ω_l)
    create ("if (Ξ(l)(Θ) = true) then")
    create ("    Θ = ϒ(l)(Θ);")
    create ("end if;")

// create process revocation for environment
create ("Θ = ϒ(ϖ)(Θ);")

// create synchronization statement and delta cycle
create ("Θ = Ω(Θ);")
create ("if ((⋁_{l∈𝕃} Ξ(l)(Θ)) = true) then")
create ("    goto start;")
create ("end if;")
```

Listing 6.8 – Construction algorithm for open designs.

6.2 Reset Analysis

The last section has introduced the new framework for static analysis of hardware description languages. This section now introduces a first data-flow problem (cf. Section 2.4 on page 19) making use of the new framework to solve it.

Chapter 5 has introduced the workflow to derive a timing model suitable for static timing analysis from a hardware specification given in VHDL. Also the need for tools supporting the derivation process was described there. The process that has been introduced consists of two main phases, namely the model preprocessing and the processor state abstraction phases. Model preprocessing reduces the size and complexity of a processor specification by information that is statically available. To gain the best return from environmental assumption refinement, where the processor is "instantiated" according to the constraints introduced by a certain application context, knowing the initial values of signals and variables within a system is important.

According to the IEEE VHDL standard [IEE87], the initial value assigned to a signal or variable is defined as the "leftmost" value of the type of the identifier. Obviously, this information can be directly extracted from a given hardware specification. But most of these standard-defined initial values will be altered during the system's boot-up phase. Whereas the standard only defines the behavior of a system when it is powered on, a system needs to sustain a stable

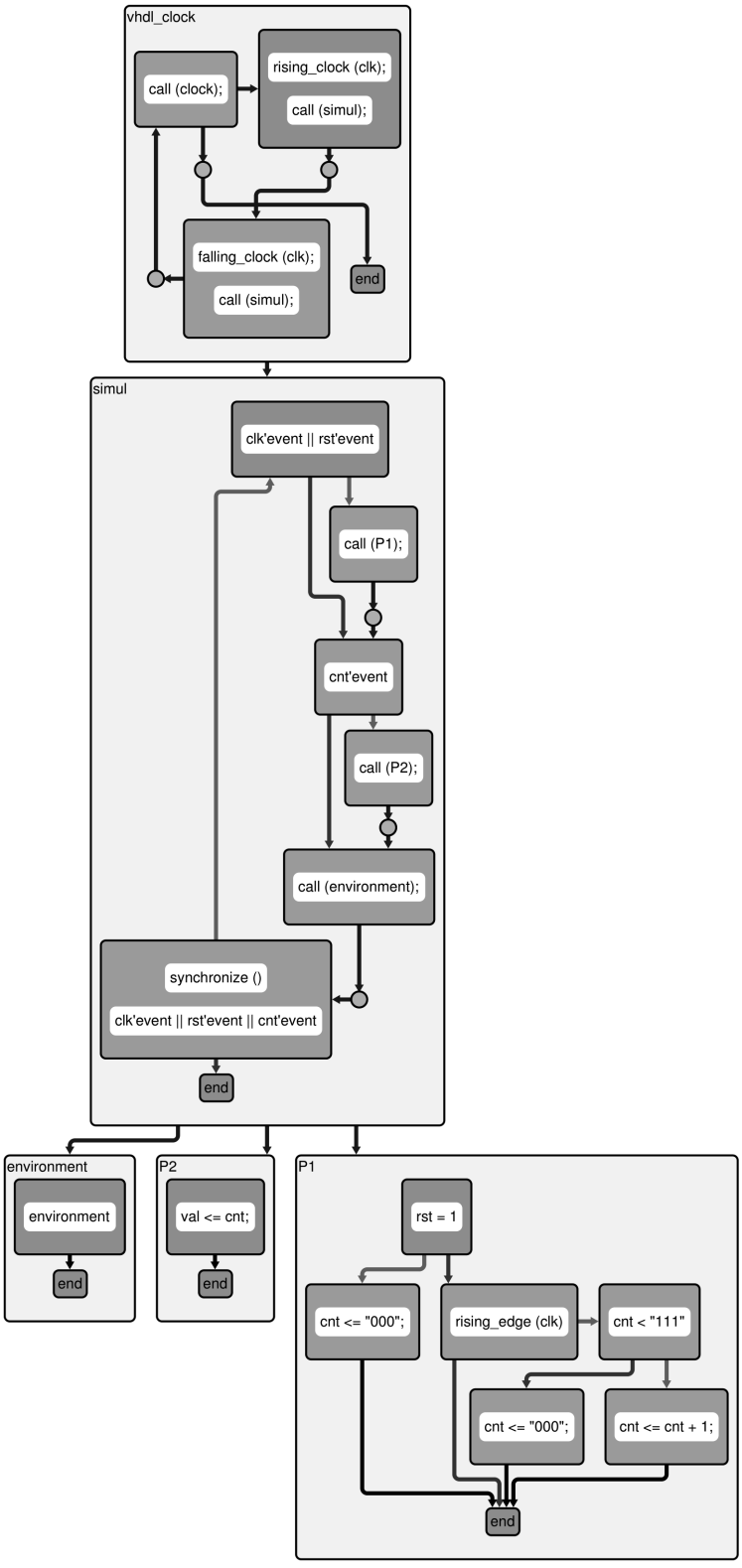


Figure 6.4 – Sequentialized representation of the 3-bit counter from Listing 4.3.

```
1   entity delayed is
        port (clk : in bit; rst : in bit;);
3   end delayed;

5   architecture rtl of delayed is
        signal d0, d1, d2, d3 : bit;
7   begin
        process (clk, rst)
9       begin
            if (rst = '1') then
11              d0 <= '1';
            end if;
13
            if rising_edge (clk) then
15              if (d0 = '1') then
                    d1 <= '1';
17              end if;

19              if (d1 = '1') then
                    d2 <= '1';
21              end if;

23              if (d2 = '1') then
                    d3 <= '1';
25              end if;
            end if;
27      end process;
    end rtl;
```

Listing 6.9 – Simplified activation chain.

and reproducible state also after a reset. Thus, the initial state (and also the initial values of signals and variables) is assigned to the system during reset handling.

Reset handling is mostly not finished within one clock cycle. In a complex computer system with several components connected via a system bus, and different peripheral devices assigning the initial values to signals and variables might take several clock ticks. The sequence of assigning initial (and stable) values to the signals of a component is called an *activation chain*.

The stable state of a system assigned during reset is apparently not visible in a hardware specification. It is the goal of the reset analysis to statically determine the initial values assigned to signals during the system's reset phase.

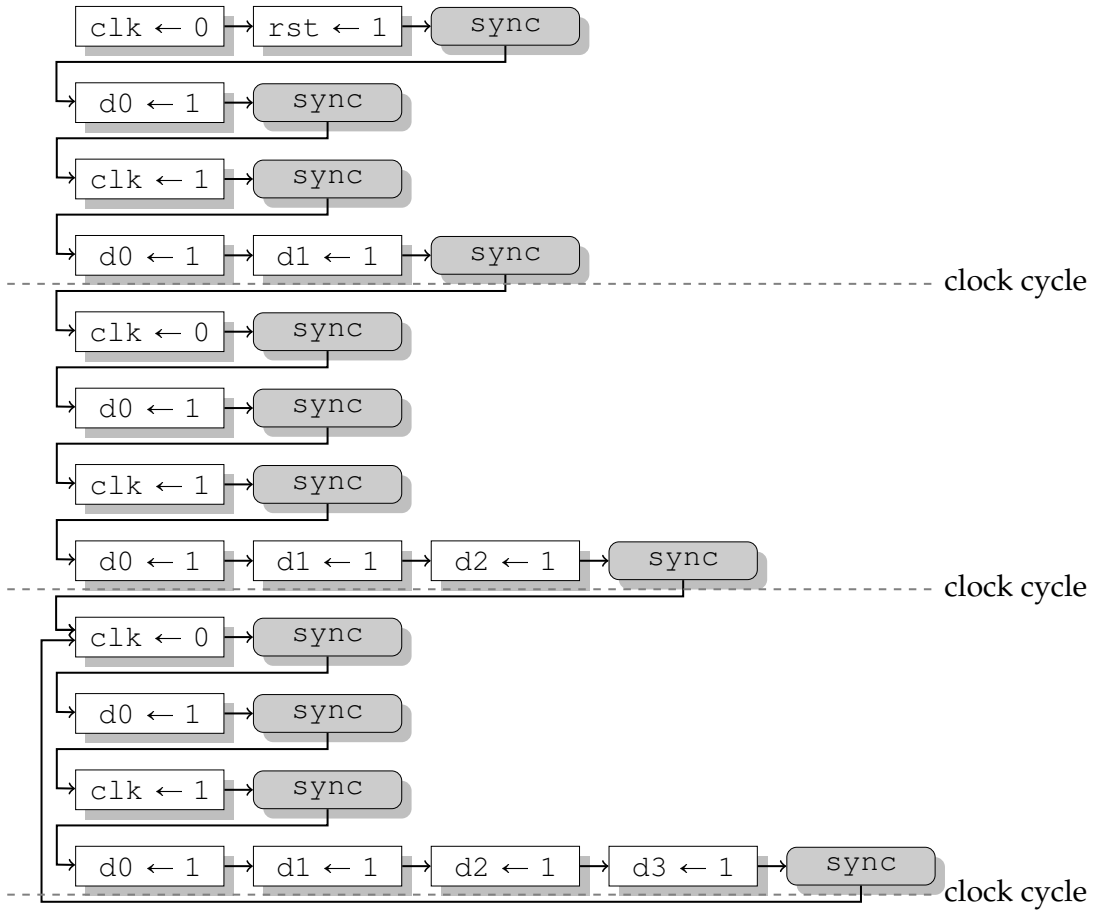


Figure 6.5 – Signal assignment trace of the VHDL example shown in Listing 6.9 on the preceding page.

Listing 6.9 on the preceding page shows a simplified activation chain. Obviously, all signals `d0`, `d1`, `d2`, and `d3` are set to `'1'` after reset, but it takes 3 ticks of the external clock to achieve the stable state, i.e. it takes 3 clock cycles to finish the activation chain.

While the example VHDL code seems artificial, activation chains are widespread when considering a computer system as a whole. During the initialization phase of memory controllers with different supported types of memory, several bridges (e.g., north and south bridges of modern chipsets), and buses with different clock domains connecting the peripheral devices, it normally takes several cycles until the system has been initialized.

Figure 6.5 depicts the trace of signal assignments for an activated reset signal on the simplified activation chain example from above. Each dashed line indicates the end/start of a clock cycle. The second semantics level of VHDL, namely

process reactivation and the waiting for a change of the environment, is depicted by gray synchronization nodes. At these points, scheduled signal assignments become visible. Within the notation of the trace semantics, the actual value of a signal can be obtained by traversing the edges backwards until the last synchronization node, and from that point on by searching backwards until the last assignment of the signal being in focus of interest. In contrast, assignments to variables are directly visible, and thus, directly searching the last assignment contained in the trace yields the desired value.

The signal assignment trace also depicts that after the third clock cycle, signals (besides the frequent tick of the externally driven clock signal) become stable, thus, the initialization phase ends. In general, only stable signals and variables are interesting for reset analysis.

From the area of compiler construction, we already know the *constant propagation analysis* that yields for each program point, whether or not a variable has a constant value whenever execution reaches that point.

Using the static analysis framework as described in Section 6.1 on page 98, this well-known analysis can be easily extended to cope with the semantical characteristics of hardware description languages.

In the following, the lattice and transfer functions will be described that, together with the control-flow graph obtained from the analysis framework, form a data-flow problem whose maximal fixed-point solution yields the constant signals under the assumption that the reset signal is triggered.

As the domain underlying the analysis, the environment as defined in Definition 6.1.1 can be used, but it needs to be extended to cope with unknown and undefined (i.e. not yet defined) values. Thus, the new domain V_{ext} is defined by

$$V_{ext} \equiv Value \cup \{\bot, \top\}$$

ordered such that $\bot \sqsubseteq_{V_{ext}} v \sqsubseteq_{V_{ext}} \top$ for all $v \in Value$, and no other ordering relationship holds. Defining V_{ext} as a *flat* domain is sufficient, since for reset analysis, we are only interested in signals becoming stable during the reset phase.

The functors $\sqcup_{V_{ext}}$ and $\sqcap_{V_{ext}}$ for $a, b \in V_{ext}$ are defined as follows:

$$a \sqcup_{V_{ext}} b \equiv \begin{cases} a & \text{if } a = b, \\ a & \text{if } b = \bot, \\ b & \text{if } a = \bot, \\ \top & \text{otherwise.} \end{cases}$$

$$a \sqcap_{V_{ext}} b \equiv \begin{cases} \bot & \text{if } a = \bot \vee b = \bot, \\ a & \text{if } b = \top, \\ b & \text{if } a = \top, \\ a & \text{if } a = b, \\ \bot & \text{otherwise.} \end{cases}$$

Using the extended value domain V_{ext}, it is possible to define the domain for the analysis mapping identifiers to a value $v \in V_{ext}$.

$$D_{cp} \equiv \{f \mid f\colon identifier \to V_{ext}\} \cup \{\bot, \top\}$$

with $f \sqsubseteq_{D_{cp}} g$ for $f, g \in D_{cp}$ defined as:

$$f = \bot \vee g = \top \vee (f, g \notin \{\bot, \top\} \wedge \forall i \in identifier\colon f(i) \sqsubseteq_{V_{ext}} g(i))$$

The combine $\sqcup_{D_{cp}}$ and meet functors $\sqcap_{D_{cp}}$ for $f, g \in D_{cp}$ are defined as:

$$f \sqcup_{D_{cp}} g \equiv \begin{cases} \top & \text{if } f = \top \vee g = \top, \\ f & \text{if } g = \bot, \\ g & \text{if } f = \bot, \\ \lambda x \in identifier.\ f(x) \sqcup_{V_{ext}} g(x) & \text{otherwise.} \end{cases}$$

$$f \sqcap_{D_{cp}} g \equiv \begin{cases} \bot & \text{if } f = \bot \vee g = \bot, \\ f & \text{if } g = \top, \\ g & \text{if } f = \top, \\ \lambda x \in identifier.\ f(x) \sqcap_{V_{ext}} g(x) & \text{otherwise.} \end{cases}$$

The lattice for the data-flow problem to analyze the reset behavior of a system thus can be defined as $(D_{cp}, \sqsubseteq_{D_{cp}}, \sqcup_{D_{cp}}, \sqcap_{D_{cp}}, \bot, \top)$, and will be denoted by D_{cp} for short. In order to define a data-flow problem, transfer-functions $f\colon E \to D_{cp} \to D_{cp}$ for the edges in a given control-flow graph $G = (V, E, s, x)$ need to be defined.

The nodes of a control-flow graph resulting from the construction algorithm shown in Listing 6.8 on page 113 can be grouped into several disjoint sets making the definition of suitable transfer functions more convenient:

$$V = V_{framework} \cup V_{\text{VHDL}}$$

V_{VHDL} corresponds to the straight-forward mapping of VHDL constructs to a control-flow representation as described in Section 6.1.3 on page 104. The nodes from the analysis framework $V_{framework}$ can be further categorized as:

$$V_{framework} = V_{clock} \cup V_{simul} \cup V_{env}$$

The classification of nodes of the control-flow graph eases the definition of transfer functions to determine constant signals in a given VHDL model. Especially for the analysis framework nodes, some special rules need to be defined in order to correctly analyze any VHDL input. In the following, the transfer function for VHDL statements will be given first, followed by the special functions handling the framework nodes.

For the definition of the transfer functions, the attribute-value concept of CRL2 will be used. Values of attributes are accessible via dedicated functions:

Definition 6.2.1 (CRL2 attributes)
Given a control-flow graph $G = (V, E, s, x)$ constructed from a CRL2 description, the value of an attribute $attr$ for a node $v \in V$ is written as $attr(v)$. Analogously, the value of an attribute $attr$ for an edge $e \in E$ is denoted by $attr(e)$.

As stated before, the straight-forward mapping of VHDL constructs to CRL2 constructs already adds attributes to the CRL2 description denoting used and defined identifiers ($uses$ and $defs$ attributes), as well as a statement and edge type classification (cat attribute). This classification can further be used to define the desired transfer functions. For the definition, the evaluation function $eval$ that has been introduced in the last section is extended to use the domain of the data-flow analysis as the environment for evaluation.

Thus, the update function $transfer_{variable}\colon D_{cp} \times V_{\text{VHDL}} \to D_{cp}$ for a VHDL variable assignment $v := expr;$ represented via its corresponding CRL2 construct $w \in V_{\text{VHDL}}$ on a lattice element $l \in D_{cp}$ is defined as:

$$transfer_{variable}(l, w) = \begin{cases} l[v \leftarrow \top] & \text{if } l \vdash eval(expr) = u_1.\cdots.u_n \wedge n > 1, \\ l[v \leftarrow u_1] & \text{if } l \vdash eval(expr) = u_1. \end{cases}$$

Whenever evaluation of the right-hand side of the variable assignment yields a list of possible values, the result is non-deterministic, and thus, the value assigned to the variable cannot statically proven to be constant. Please note that also in the case of a deterministic return value of the function $eval$, the returned value might be $\top$.

Analogously to variable assignments, the transfer function $transfer_{signal}\colon D_{cp} \times V_{\text{VHDL}} \to D_{cp}$ for a VHDL signal assignment $s <= expr;$ represented via its corresponding CRL2 construct $w \in V_{\text{VHDL}}$ on a lattice element $l \in D_{cp}$ is defined as:

$$transfer_{signal}(l, w) = \begin{cases} l[\bar{s} \leftarrow \top] & \text{if } l \vdash eval(expr) = u_1.\cdots.u_n \wedge n > 1, \\ l[\bar{s} \leftarrow u_1] & \text{if } l \vdash eval(expr) = u_1. \end{cases}$$

Update functions for signal and variable assignment directly follow the definition of the abstract simulation rules for VHDL. In order to compute precise results, if-statements in the VHDL model are of interest.

The transfer function $transfer_{if}\colon D_{cp} \times E_{\text{VHDL}} \to D_{cp}$, with $E_{\text{VHDL}} = \{e \mid e = (m,n) \in E \wedge m,n \in V_{\text{VHDL}}\}$, for a VHDL if-statement $expr$ represented via its corresponding CRL2 construct $m \in V_{\text{VHDL}}$ for an edge $e = (m,n)$ is defined as:

$$transfer_{if}(l,e) = \begin{cases} \bot & \text{if } l = \bot, \\ \top & \text{if } l = \top, \\ l & \text{if } l \vdash eval(expr) = u_1.\cdots.u_n \wedge n > 1, \\ l & \text{if } l \vdash eval(expr) = u_1 \wedge u_1 = \top, \\ \bot & \text{if } l \vdash eval(expr) = u_1 \wedge u_1 \neq cat(e), \\ l & \text{otherwise.} \end{cases}$$

Whenever the evaluation of the condition is non-deterministic, the current data-flow value is propagated to the destination of the given edge. In case of a deterministic evaluation, i.e. $n = 1$, the current data-flow value is only propagated over the edge matching the result of the expression evaluation, i.e. if the evaluation of the expression returns true, but the categorization of the current edge marks it as a false-edge, $\bot$ is returned. This transfer function allows modeling of the sequential VHDL simulation rules. Combined with the transfer functions for signal and variable assignments, the update function for VHDL-statements can be defined.

The update function $transfer_{\text{VHDL}}\colon D_{cp} \times E_{\text{VHDL}} \to D_{cp}$ for an edge $e = (m,n) \in E_{\text{VHDL}}$ can be defined as:

$$transfer_{\text{VHDL}}(l,e) = \begin{cases} transfer_{variable}(l,m) & \text{if } cat(m) = variableassignment, \\ transfer_{signal}(l,m) & \text{if } cat(m) = signalassignment, \\ transfer_{if}(l,e) & \text{if } cat(m) = ifstatement, \\ l & \text{otherwise.} \end{cases}$$

The transfer function mentioned above only deals with statements being directly derived from the original VHDL model. In order to define a data-flow problem, also updates for the routines derived from the analysis framework need to be given. As mentioned before, the framework helpers can be distinguished into a clock simulation routine, the process reactivation simulation routine and the environment routine. In the following, the transfer functions for these routines will be given.

Given the identifier of the reset signal rst and its activation value $v \in Value$, the transfer function for the environment routine $transfer_{env}\colon D_{cp} \times V_{env} \to D_{cp}$ for

a node $w \in V_{env}$ on a data-flow element $l \in D_{cp}$ is defined as:

$$transfer_{env}(l, w) = \lambda i \in identifier. \begin{cases} v & \text{if } i = rst \vee i = \overline{rst} \vee i = \underline{rst}, \\ l(i) & \text{otherwise.} \end{cases}$$

The environment process thus is used to ensure that the reset signal remains activated during the whole analysis. In general, the reset signal of a processor is not updated within its specification, i.e. the signal normally is used as an input signal which has to be triggered by the environment, namely the user pushing the reset button. Nevertheless, in the area of embedded systems, a processor is also reset in case of an uncorrectable error.

The transfer function for the nodes of the process reactivation simulation routine $transfer_{simul}\colon D_{cp} \times E_{simul} \rightarrow D_{cp}$ for an edge $e \in E_{simul} \equiv \{e \mid e = (m, n) \in E \wedge m, n \in V_{simul}\}$ on a given control-flow graph $G = (V, E, s, x)$ and a data-flow value $l \in D_{cp}$ is defined as:

$$transfer_{simul}(l, e) = \begin{cases} \top & \text{if } l = \top, \\ \bot & \text{if } l = \bot, \\ l & \text{if } cat(m) = ifstatement \wedge l \vdash \Xi(m) = \top, \\ \bot & \text{if } cat(m) = ifstatement \wedge l \vdash \Xi(m) \neq cat(e), \\ \Omega(l) & \text{if } cat(m) = syncstatement, \\ l & \text{otherwise.} \end{cases}$$

The update rules for the simulation process use the definitions of the if-guard statement Ξ and the synchronize statement Ω as defined in Definitions 6.1.3 and 6.1.5. Also the $event$-attribute must be extended to cope with elements of V_{ext} instead of $Value$. This also implies that Ξ returns a tristate value $true$, $false$ and $\top$. Whenever it is statically undecidable if a process statement has to be reinvoked, the data-flow value is propagated over both edges of the if-guard statement. At the synchronization statement, all scheduled transactions become visible, which is ensured by the Ω update. The delta-delay (required for process reactivation) is represented via another if-statement in the analysis framework, and thus, the rules for the if-guard statements also apply to it.

The clock routine of the analysis framework is used to model the frequent change of a clock signal. Since the reset analysis shall determine constant signals during system reset, modeling the frequent change of the clock is also important for the analyzer. Most current systems are triggered on the rising edge of a clock signal. An example of such a system is shown in Listing 6.9. As stated above, this system finishes its initialization phase after 3 clock cycles. In general, the duration of an initialization phase in an arbitrary model must be bounded, but is not limited to a certain value. Thus, the reset analyzer needs to compute the fixed point over the process executions in order to determine the desired stable signal values.

Given the name of an external clock signal clk, the transfer function $transfer_{clock}$: $D_{cp} \times V_{clock} \rightarrow D_{cp}$ for the clock routine of the analysis framework for a node $v \in V_{clock}$ on a data-flow value $l \in D_{cp}$ is defined as:

$$transfer_{clock}(l, v) = \begin{cases} \Gamma^{\uparrow}(clk)(l) & \text{if } cat(v) = risingedge, \\ \Gamma^{\downarrow}(clk)(l) & \text{if } cat(v) = fallingedge, \\ l & \text{otherwise.} \end{cases}$$

Given a control-flow graph $G = (V, E, s, x)$. The update functions defined above can now be combined to a transfer function $transfer_{cp,e} \colon D_{cp} \rightarrow D_{cp}$ for an edge $e = (m, n) \in E$ on a lattice element $l \in D_{cp}$:

$$transfer_{cp,e}(l) = \begin{cases} transfer_{\text{VHDL}}(l, e) & \text{if } e \in E_{\text{VHDL}}, \\ transfer_{clock}(l, m) & \text{if } e = (m, n) \in E_{clock}, \\ transfer_{simul}(l, e) & \text{if } e \in E_{simul}, \\ transfer_{env}(l, m) & \text{if } e = (m, n) \in E_{env}, \\ l & \text{otherwise.} \end{cases}$$

The return function $return \colon D_{cp} \times D_{cp} \rightarrow D_{cp}$ combining the data-flow values coming from the local edge (a) and the return edge (b) at return nodes is defined as:

$$return(a, b) = b$$

Now, it is possible to define a function $\mathfrak{f}_{cp} \colon E \rightarrow (D_{cp} \rightarrow D_{cp})$ for an edge $e \in E$ as:

$$\mathfrak{f}_{cp}(e) = transfer_{cp,e}$$

This yields the data-flow problem $dfp_{cp} = (G, D_{cp}, \mathfrak{f}_{cp})$, with $G = (V, E, s, x)$. Given the name of the reset signal rst, and its activation value $v \in V_{ext}$, the initial state ι is defined as:

$$\iota \equiv \lambda i \in identifier. \begin{cases} v & \text{if } i = rst \vee i = \overline{rst} \vee i = \underline{rst}, \\ \top & \text{otherwise.} \end{cases}$$

The data-flow problem describes a forward analysis. At control-flow joins, the union of the incoming information is formed. It can now be solved using one the algorithms described in Section 2.4.1 on page 21 leading to the maximal fixed-point solution MFP_{cp}. The maximal fixed-point is computed for every node $v \in V$ of the given control-flow graph.

Reset analysis shall determine the set of stable identifiers under the assumption that the reset is triggered. A stable identifier is not allowed to change its value

until the reset signal is asserted anymore. Furthermore, the value of a stable identifier must not flip depending on the current state of the external clock signal. Thus, the set of stable identifiers during the system initialization phase can be determined by combining the maximal fixed-point solutions of the data-flow problem dfp_{cp} for the rising and falling clock nodes of the clock process κ.

We define two functors $\tilde{\sqcap}_{V_{ext}}$ and $\tilde{\sqcap}_{D_{cp}}$ for $a, b \in V_{ext}$ and $f, g \in D_{cp}$ helping in determining the set of stable identifiers:

$$a \;\tilde{\sqcap}_{V_{ext}}\; b \quad \equiv \begin{cases} \top & \text{if } a = \top \vee b = \top, \\ a & \text{if } a = b, \\ \bot & \text{if } a \neq b, \\ \top & \text{otherwise.} \end{cases}$$

$$f \;\tilde{\sqcap}_{D_{cp}}\; g \quad \equiv \begin{cases} \bot & \text{if } f = \bot \vee g = \bot, \\ f & \text{if } g = \top, \\ g & \text{if } f = \top, \\ \lambda x \in \mathit{identifier}.\; f(x) \;\tilde{\sqcap}_{V_{ext}}\; g(x) & \text{otherwise.} \end{cases}$$

Let $v_1, v_2 \in V_{clock} \subset V$, with $cat(v_1) = risingedge$ and $cat(v_2) = fallingedge$. The set of stable identifiers during system reset is defined as:

$$\begin{aligned} S_{stable_{cp}} = \{ s \mid\; & s \in \mathit{identifier} \\ & \wedge\, (MFP_{cp}(v_1) \;\tilde{\sqcap}_{D_{cp}}\; MFP_{cp}(v_2))(s) \neq \top \\ & \wedge\, (MFP_{cp}(v_1) \;\tilde{\sqcap}_{D_{cp}}\; MFP_{cp}(v_2))(s) \neq \bot \} \end{aligned}$$

The definition of the function lattice D_{cp} and the functor $\tilde{\sqcap}_{cp}$ ensure that identifiers that are mapped to different values depending on the state of the external clock signal are not stable. Moreover, the initial element maps all identifiers (except the reset signal) to $\top$, thus the definition of $\tilde{\sqcap}_{D_{cp}}$ also guarantees that for two elements $a, b \in V_{ext}$, it holds that:

$$a \;\tilde{\sqcap}_{V_{ext}}\; b = \bot \implies a \neq \top \wedge b \neq \top \wedge (a \neq b \vee (a = \bot \wedge b = \bot))$$

So, whenever a signal identifier s is mapped to $\bot$ when applying $\tilde{\sqcap}_{D_{cp}}$ to the falling and rising data-flow values, it can be guaranteed that it has been assigned different stable values depending on the external clock signal. The signal s is said to define a different *clock domain*. Listing 6.10 on the following page gives a small excerpt of the register transfer-level description of the superscalar DLX processor (cf. Section 7.1.2). The external clock signal is named `IncomingClock`, but processor updates are triggered on the rising edge of an internally defined clock signal `Clock`, which is derived from the external signal by conjunction with the negation of an internal halt signal.

```
entity Dlx is
    port (IncomingClock : in bit;
          Reset : in bit;);
end Dlx;

architecture arch of Dlx is
    signal Clock : bit;
    signal DP_HaltFlag : bit;
begin
    P1: Clock <= IncomingClock and not DP_HaltFlag;

    P2: process(Clock, IncomingClock)
        begin
            if Reset = '1' then
                DP_HaltFlag <= '0';
            end if;

            if (rising_edge (Clock)) then
                # processor update logic
                ...
            end if;
        end process;
end;
```

Listing 6.10 – Simplified example of a clock domain.

Thus, a side-product of the reset analysis is the set of possible clock domains S_{pclk}, which can be defined as follows. Let $v_1, v_2 \in V_{clock} \subset V$ be the nodes representing the rising and falling clock updates of the clock simulation routine, i.e. $cat(v_1) = risingedge$ and $cat(v_2) = fallingedge$, S_{pclk} is defined as

$$S_{pclk} = \{s \mid s \in identifier \wedge (MFP_{cp}(v_1) \mathbin{\tilde{\sqcap}_{D_{cp}}} MFP_{cp}(v_2))(s) = \bot\}$$

The knowledge of different clock domains is very useful in the process of deriving a timing model from a formal hardware specification, since it enables a human expert to find more suitable abstractions in order to derive a small, but also precise timing model. More details on abstractions of hardware description languages can be found in [Pis12].

6.2.1 Analysis of Functions and Procedures

This section describes an enhancement to the reset analysis presented so far. VHDL, and many other hardware description languages, also provide functions

and procedures to implement functionality that will be often reused. The distinction between a function and a procedure is limited to the kind of parameter passing: function calls transfer their parameters by value, whereas procedure calls use parameter passing by reference. Furthermore, procedures do not return a value.

Procedures and functions are widely used from within VHDL specifications, thus extending the data-flow problem dfp_{cp} as presented before to cope with functions and procedures will result in more precise analysis results. The solution presented in this section makes use of the call/return approach mentioned before. Every function and procedure call statement within a processor specification is represented in the CRL2 description via a pair of call and return nodes connected to the routine representing the called function/procedure (cf. Figure 6.2 on page 102).

The parameters occurring in the heads of function and procedure declarations are named *formal parameters*, the identifiers used at caller side to call the function (or procedure) are called *actual parameters*. To enhance the reset analysis, mapping actual to formal parameters is necessary.

As before, modeling function- and procedure-calls uses the flexible attribute-value concept of CRL2. The function $formal$ returns for a node $v \in V_{\text{VHDL}}$ of a control-flow graph $G = (V, E, s, x)$ a list of formal parameter names of the function, v is contained in. If v does not belong to a function or procedure, the empty list is returned. Analogously, the function $actual$ can be defined returning the list of actual parameter names for a node v. If v does not represent a call or a return node (i.e. $cat(v) \neq callnode \wedge cat(v) \neq returnnode$), the empty list is returned. Using these two functions, the update function $transfer_{call}\colon D_{cp} \times E_{\text{VHDL}} \to D_{cp}$ for an edge $e = (m, n) \in E_{\text{VHDL}}$, with $cat(e) = calledge$, on a lattice element $l \in D_{cp}$ can be defined. Let $formal(n) = p_0.\cdots.p_s$ and $actual(m) = a_0.\cdots.a_s$, with $s \geqslant 0$, $transfer_{call}(l, e)$ is defined as:

$$\forall i \in [0, s]\colon l(p_i) = l(a_i)$$

Please note that static type checking of VHDL already guarantees that functions and procedures are called with the correct number of parameters. Thus, the lengths of the lists returned from $actual$ and $formal$ must be equal.

Whereas the update function for call statements establishes a mapping from actual to formal parameters, this mapping needs to be destroyed after the call, since formal parameter names are not live after the function (or procedure) call. For procedure calls, where parameters are passed by reference, also the modified values must be updated. Thus, the update function $transfer_{return}\colon D_{cp} \times E_{\text{VHDL}} \to D_{cp}$ for an edge $e = (m, n) \in E_{\text{VHDL}}$ with $cat(e) = returnedge$ on a lattice element $l \in D_{cp}$ can be defined. Let $apply\colon D_{cp} \times identifier \times V_{ext} \to D_{cp}$ be a function

defined by

$$apply(l, i, v) = \lambda j \in identifier. \begin{cases} v & \text{if } j = i, \\ l(j) & \text{otherwise.} \end{cases}$$

Let $formal(m) = p_0. \cdots .p_s$ and $actual(n) = a_0. \cdots .a_s$, with $s \geqslant 0$, $apply$ can be used to define $transfer_{return}(l, e)$ as follows:

$$\forall i \in \{0, \ldots, s\} : l := \begin{cases} apply(apply(l, a_i, l(p_i)), p_i, \top) & \text{if } is\_procedure(m) = true, \\ apply(l, p_i, \top) & \text{otherwise.} \end{cases}$$

Please note that due to elaboration, identifiers used in the resulting VHDL model are unique, i.e. they are not overloaded. This also applies to new naming scopes introduced by function calls, etc. Thus, identifiers used at callee side can never be modified at the caller side.

Both functions now have to be integrated into the update function for VHDL statements, resulting in the update function $transfer_{\text{VHDL}} : D_{cp} \times E_{\text{VHDL}} \to D_{cp}$ for an edge $e = (m, n) \in E_{\text{VHDL}}$:

$$transfer_{\text{VHDL}}(l, e) = \begin{cases} transfer_{variable}(l, m) & \text{if } cat(m) = variableassignment, \\ transfer_{signal}(l, m) & \text{if } cat(m) = signalassignment, \\ transfer_{if}(l, e) & \text{if } cat(m) = ifstatement, \\ transfer_{call}(l, e) & \text{if } cat(e) = calledge, \\ transfer_{return}(l, e) & \text{if } cat(e) = returnedge, \\ l & \text{otherwise.} \end{cases}$$

Use of this function in the data-flow problem dfp_{cp} as defined above enables analysis of procedure and function calls in order to enhance the results.

6.3 Assumption-based Model Refinement

This section introduces the assumption-based model refinement analysis, or assumption evaluation for short. The goal of the assumption evaluation analysis is to support the process of environmental assumption refinement as introduced in Chapter 5 on page 77.

Modern processors offer a huge variety in configurable components (and features) in order to make them universally applicable. This configurability compared to the restricted usage within a specific embedded area offers the possibility of instantiating a timing model for timing analysis for a dedicated field of application. E.g., interrupt handling often brings a system to a state, where a timing bound for the task currently being executed is no longer of interest.

Thus, the parts in a specification being concerned with the implementation of the processor's interrupt handling can be safely removed for timing analysis.

The specific field of application of a processor within the embedded area thus leads to a set of assumptions on the physical environment. Moreover, asynchronous events like DMA, or erroneous events like transfer errors due to alpha collisions leading to bit flips cannot be covered by any static analysis, but by statistical means. Events that can be covered by statistical means can also be used to extend the set of assumptions on the specific use and the behavior of a processor for timing analysis. These assumptions render parts of a processor specification obsolete. Parts of the specification can be safely marked as timing dead. Signals and variables that are modified within this timing-dead code might become stable, or their co-domains might be restricted to a smaller subset. Knowing these restricted values, other parts within a VHDL processor specification might also become timing dead.

Furthermore, if a signal (or variable) is known to always be of a constant value, assignments to this signal (or variable) can be removed from the specification in order to reduce its size. Remaining read references then access constant values, which may also be inlined.

Identification of these signals can be automated by extending a constant propagation analysis to deal with an interval domain. Using the set of environmental assumptions, all signals that become constant under these assumptions can be identified. Moreover, evaluating conditional expressions under the newly computed restricted co-domains for signals and variables allows marking of code sequences (namely the then- or else-branch) as timing dead, since under the given set of assumptions, it can statically be guaranteed that control never reaches these program points. In this case, also the conditional statement can be safely removed from the processor specification.

For analysis, the environment has to be extended to use a pair of values $(a, b) \in \mathit{Value} \times \mathit{Value}$ representing the set $\{x \mid x \in \mathit{Value} \wedge a \leqslant x \leqslant b\}$.[2] In the following, $[a, b]$ is used to denote the set of values contained in the interval from a till b. For convenience, the left and right end point of an interval $X \in \mathit{Value} \times \mathit{Value}$ will be denoted by $\underline{X}$ and $\overline{X}$, respectively.

$$X = [\underline{X}, \overline{X}]$$

The Cartesian product domain representing intervals is extended by additional top and bottom elements to ease data-flow analyses. So, the domain used within the environment is defined by

$$V_{Int} = (\mathit{Value} \times \mathit{Value}) \cup \{\bot, \top\}$$

[2] Type-domains within VHDL must be mappable to basic partially ordered types, thus, w. l. o. g., the domain Value is a partially ordered set with the ordering relation $\leqslant$.

partially ordered such that $\bot \sqsubseteq_{V_{Int}} X \sqsubseteq_{V_{Int}} \top$ for all $X \in \mathit{Value} \times \mathit{Value}$, and for $X, Y \in \mathit{Value} \times \mathit{Value}: \underline{Y} \leqslant \underline{X} \wedge \overline{X} \leqslant \overline{Y} \implies X \sqsubseteq_{V_{Int}} Y$.

The functors $\sqcup_{V_{Int}}$ and $\sqcap_{V_{Int}}$ for $X, Y \in V_{Int}$ are defined as follows:

$$X \sqcup_{V_{Int}} Y \equiv \begin{cases} \top & \text{if } X = \top \vee Y = \top, \\ X & \text{if } Y = \bot, \\ Y & \text{if } X = \bot, \\ [min(\underline{X}, \underline{Y}), max(\overline{X}, \overline{Y})] & \text{otherwise.} \end{cases}$$

$$X \sqcap_{V_{Int}} Y \equiv \begin{cases} \bot & \text{if } X = \bot \vee Y = \bot, \\ X & \text{if } Y = \top, \\ Y & \text{if } X = \top, \\ \bot & \text{if } \overline{Y} < \underline{X} \vee \overline{X} < \underline{Y}, \\ [max(\underline{X}, \underline{Y}), min(\overline{X}, \overline{Y})] & \text{otherwise.} \end{cases}$$

Please note that $\sqcup_{V_{Int}}$ computes the *interval hull* of two intervals, since in general, the union of two intervals is not an interval.

Using this interval domain, the analysis domain for the assumption-based model refinement is defined as

$$D_{Int} = \{f \mid f: \mathit{identifier} \to V_{Int}\} \cup \{\bot, \top\}$$

The ordering relation $f \sqsubseteq_{D_{Int}} g$ for two elements $f, g \in D_{Int}$ is given by

$$f = \bot \vee g = \top \vee (f, g \notin \{\bot, \top\} \wedge \forall i \in \mathit{identifier}: f(i) \sqsubseteq_{V_{Int}} g(i))$$

The combine and meet functors $\sqcup_{D_{Int}}$ and $\sqcap_{D_{Int}}$ for $f, g \in D_{Int}$ are defined as:

$$f \sqcup_{D_{Int}} g \equiv \begin{cases} \top & \text{if } f = \top \vee g = \top, \\ f & \text{if } g = \bot, \\ g & \text{if } f = \bot, \\ \lambda x \in \mathit{identifier}.\ f(x) \sqcup_{V_{Int}} g(x) & \text{otherwise.} \end{cases}$$

$$f \sqcap_{D_{Int}} g \equiv \begin{cases} \bot & \text{if } f = \bot \vee g = \bot, \\ f & \text{if } g = \top, \\ g & \text{if } f = \top, \\ \lambda x \in \mathit{identifier}.\ f(x) \sqcap_{V_{Int}} g(x) & \text{otherwise.} \end{cases}$$

The domain D_{Int} paired with the ordering relation $\sqsubseteq_{D_{Int}}$, the join and meet functors $\sqcup_{D_{Int}}$ and $\sqcap_{D_{Int}}$, and the least and greatest elements $\bot$ and $\top$ forms a lattice used as the domain for the data-flow problem. In the following, the adaptations to the transfer functions presented in the previous section in order to define the reset analysis will be described. These updated transfer functions

then form a data-flow problem, whose fixed-point solution can be used to identify smaller co-domains for signals and variables and also to mark parts of a processor specification as timing-dead.

In order to cope with intervals, the evaluation function $eval$ for arithmetic expression evaluation of VHDL statements has to be extended according to the interval arithmetic rules described in [MKC66]. The return value of the evaluation function is no longer a list of possible results, but the interval hull enclosing all result values.

Using the extended version, the update function for variable assignment statements $transfer_{variable} : D_{Int} \times V_{\text{VHDL}} \rightarrow D_{Int}$ for a variable assignment $v := expr;$ represented by a node $w \in V_{\text{VHDL}}$ on a lattice element $l \in D_{Int}$ can be defined as

$$transfer_{variable}(l, w) = l[v \leftarrow U]$$

with $l \vdash eval(expr) = U$.

Analogously, the update function for signal assignment statements $transfer_{signal} : D_{Int} \times V_{\text{VHDL}} \rightarrow D_{Int}$ for a signal assignment $s <= expr;$ represented by a node $v \in V_{\text{VHDL}}$ on a lattice element $l \in D_{Int}$ can be defined as

$$transfer_{signal}(l, v) = l[\overline{s} \leftarrow U]$$

with $l \vdash eval(expr) = U$.

To enhance the quality of the analysis results, also conditional statements must be analyzed. Whenever the conditional expression can be statically determined to be true or false in every case, the then- or else-branch can be rendered as infeasible. Thus, statements contained in the infeasible branch are known to be never executed and therefore do not contribute to the values computed by the program.

In order to model this behavior, the update function $transfer_{if} : D_{Int} \times E_{\text{VHDL}} \rightarrow D_{Int}$ on a lattice element l and an edge $e = (m, n)$ for a VHDL if-statement represented via its corresponding CRL2 construct $m \in V_{\text{VHDL}}$ is defined as:

$$transfer_{if}(l, e) = \begin{cases} \bot & \text{if } l = \bot, \\ \top & \text{if } l = \top, \\ l & \text{if } l \vdash eval(expr(m)) = U \wedge (U = \top \vee U = cat(e)) \\ \bot & \text{if } U \neq cat(e), \\ l & \text{otherwise.} \end{cases}$$

where the CRL2-attribute $expr$ is used to access the VHDL-mnemonic of conditionals and the right-hand side of assignment statements, i.e. for a node n representing the statement $s <= a + b;$, $expr(n) = a + b$.

Intuitively, the current data-flow value is propagated over an edge of a conditional statement if the condition can statically be evaluated to match the type of the edge, i.e. if the condition is evaluated to true, the data-flow value is propagated over the true, but not over the false edge. Whenever the condition cannot be deterministically evaluated, the data-flow value is propagated over the true as well as the false edge.

Analysis of procedure and function calls directly follows the definitions presented in Section 6.2.1 on page 124 and is therefore omitted here. Using these update functions, the transfer function for $transfer_{\text{VHDL}} : D_{Int} \times E_{\text{VHDL}} \to D_{Int}$ for an edge $e = (m, n) \in E_{\text{VHDL}}$ can be easily defined:

$$transfer_{\text{VHDL}}(l, e) = \begin{cases} transfer_{variable}(l, m) & \text{if } cat(m) = variableassignment, \\ transfer_{signal}(l, m) & \text{if } cat(m) = signalassignment, \\ transfer_{if}(l, e) & \text{if } cat(m) = ifstatement, \\ transfer_{call}(l, e) & \text{if } cat(e) = calledge, \\ transfer_{return}(l, e) & \text{if } cat(e) = returnedge, \\ l & \text{otherwise.} \end{cases}$$

The analysis framework presented in this chapter has introduced a process handling the environment of the VHDL specification to be analyzed. This process can now be used to instantiate the assumptions provided by the user. Thereby, it does not matter whether a specific assumption is concerned with an external signal of the specification or with an internal one since the user guarantees that the given signal does not change its value for the rest of the analysis time. One of the simplest assumptions used for timing model derivation is the assumption "The reset is never triggered". By this, all initialization code in a given specification is pruned out.

Given a mapping $assume : identifier \to V_{Int}$ from identifiers to value ranges representing the assumption provided by the user, where for all signal identifiers s it holds that $assume(s) = assume(\overline{s}) = assume(\underline{s})$, the update function for the environment process $transfer_{env} : D_{Int} \times V_{env} \to D_{Int}$ for a node $n \in V_{env}$ and a data-flow element $l \in D_{Int}$ is defined as:

$$transfer_{env}(l, v) = \lambda i \in identifier. \begin{cases} assume(i) & \text{if } assume(i) \neq \bot \\ & \quad \wedge\ assume(i) \neq \top, \\ l(i) & \text{otherwise.} \end{cases}$$

In order to define the data-flow problem of the assumption evaluator, also the update functions for the process reactivation simulation routine and the clock simulation routine need to be provided. Fortunately, both update functions equal the definitions provided for the reset analysis, but their domains (i.e. the

environment) have to be adapted. They can be analogously extended to cope with intervals instead of single values, and thus, their definitions are omitted here.

Given a control-flow graph $G = (V, E, s, x)$. The update functions defined above can now be combined to a transfer function $transfer_{Int,e} \colon D_{Int} \to D_{Int}$ for an edge $e = (m, n) \in E$ on a lattice element $l \in D_{Int}$:

$$transfer_{Int,e}(l) = \begin{cases} transfer_{\text{VHDL}}(l, e) & \text{if } e \in E_{\text{VHDL}}, \\ transfer_{clock}(l, m) & \text{if } e = (m, n) \in E_{clock}, \\ transfer_{simul}(l, e) & \text{if } e \in E_{simul}, \\ transfer_{env}(l, m) & \text{if } e = (m, n) \in E_{env}, \\ l & \text{otherwise.} \end{cases}$$

The return function $return \colon D_{Int} \times D_{Int} \to D_{Int}$ combining the data-flow values coming from the local edge (a) and the return edge (b) at return nodes is defined as:

$$return(a, b) = b$$

To obtain the desired data-flow problem, the function $\mathfrak{f}_{Int} \colon E \to (D_{Int} \to D_{Int})$ for an edge $e \in E$ is to be defined as:

$$\mathfrak{f}_{Int}(e) = transfer_{Int,e}$$

Using this function, $dfp_{Int} = (G, D_{Int}, \mathfrak{f}_{Int})$, with $G = (V, E, s, x)$ forms a data-flow problem. Given the mapping $assume$ of user-assumptions as mentioned above, the maximal fixed-point solution of the reset analysis MFP_{cp}, and the set of stable identifiers $S_{stable_{cp}}$ derived from the reset analysis, the initial state ι used for solving this data-flow problem is defined as:

$$\iota \equiv \lambda i \in identifier. \begin{cases} (MFP_{cp}(x)(i), MFP_{cp}(x)(i)) & \text{if } i \in S_{stable_{cp}}, \\ assume(i) & \text{otherwise.} \end{cases}$$

Illustratively, the initial element used for solving the data-flow problem uses the results from a preceding reset analysis in order to enhance the analysis results. Furthermore, the assumptions about the behavior of dedicated signals and variables are also integrated into the starting value.

The data-flow problem describes a forward analysis requiring the union of incoming information at control-flow joins. It can now be solved using one the algorithms described in Section 2.4.1 on page 21 leading to the maximal fixed-point solution MFP_{Int}. The maximal fixed-point thereby is computed for every node $v \in V$ of the given control-flow graph.

The result of the assumption evaluation analysis is the set of newly stable identifiers within a VHDL specification based on assumptions on the behavior of the processor within a specific embedded environment. So, let $v_1, v_2 \in V_{clock} \subset V$, with $cat(v_1) = risingedge$ and $cat(v_2) = fallingedge$, this set is defined as

$$\begin{aligned} S_{stable_{Int}} = \{s \mid\ & s \in identifier \\ & \wedge\ (MFP_{Int}(v_1) \sqcap_{D_{Int}} MFP_{Int}(v_2))(s) \neq \top \\ & \wedge\ (MFP_{Int}(v_1) \sqcap_{D_{Int}} MFP_{Int}(v_2))(s) \neq \bot \\ & \wedge\ MFP_{Int}(v_1)(s) = MFP_{Int}(v_2)(s) = X \wedge \underline{X} = \overline{X}\} \end{aligned}$$

The knowledge on stable identifiers and their specific values now can be used to reduce the size of the VHDL specification by pruning out assignments to identifiers known to be stable and inlining their use within other expressions.

Therefore, new CRL2-attributes $timing\_dead\colon V \to \mathbb{B}$ and $timing\_dead\colon E \to \mathbb{B}$ are introduced marking statements or edges within a control-flow graph as timing dead (cf. Definition 5.3.1 on page 94). A timing-dead mark for a statement semantically implies that the statement shall be ignored for further analyses, and the statement can be safely removed from the control-flow graph, and therewith, also from the specification. A timing-dead mark at an edge implies that every path $\pi = (v_1, \ldots, v_k, v_l, \ldots, v_n)$ starting from the unique start node s of a control-flow graph $G = (V, E, s, x)$, i.e. $v_1 = s$, containing an edge $e = (v_k, v_l)$, with $timing\_dead(e) = true$ has to be viewed as unreachable under the given set of assumptions. A node $v \in V$ that is only accessible via an edge that has been marked as timing dead (i.e. all paths to v starting at s contain at least one edge e, where $timing\_dead(e) = true$), can also be removed for timing model derivation.

Timing-dead marks for statements and edges can now be automatically computed using the set of stable identifiers $S_{stable_{Int}}$ and the result of the assumption evaluation analysis MFP_{Int}. Thus, it is possible to mark signal assignment statements of a VHDL specification as timing dead if the signal to be assigned is known to be stable:

$$\begin{aligned} & \forall n \in V_{\text{VHDL}}\colon cat(n) = signalassignment \wedge def(n) \in S_{stable_{Int}} \\ & \implies timing\_dead(n) = true \end{aligned}$$

Removing signal assignment statements of signals that are known to be stable under a set of assumptions also directly implies that remaining read references of a stable signal $s \in S_{stable_{Int}}$ access a constant value, namely $MFP_{Int}(x)(s)$. This constant value can also be inlined in the right-hand side expression of VHDL statements.

The design of the assumption evaluation analysis dfp_{Int} to use an interval domain being capable to not only identify constant signals, but also more restricted

co-domains for identifiers additionally allows for identification of paths in a control-flow graph that cannot be reached under a certain set of assumptions. Whenever an expression of a conditional statement can be statically evaluated to be always true or false (i.e. the result of the conditional expression becomes stable), the then- or else-branch can be rendered obsolete.

$$\begin{aligned}&\forall e = (m,n) \in E_{\text{VHDL}}, \text{ with } cat(e) = true \vee cat(e) = false:\\ &MFP_{Int}(m) \vdash eval(expr(m)) \neq cat(e)\\ &\wedge MFP_{Int}(m) \vdash eval(expr(m)) \notin \{\bot, \top\}\\ &\implies timing\_dead(e) = true\end{aligned}$$

Furthermore, every conditional statement whose outcome can be statically determined to be constant can also be marked as timing-dead. By this, a further reduction in size of a given VHDL specification can be achieved.

$$\begin{aligned}&\forall n \in V_{\text{VHDL}}, \text{ with } cat(n) = ifstatement:\ MFP_{Int}(n) \vdash eval(expr(n)) \notin \{\bot, \top\}\\ &\implies timing\_dead(n) = true\end{aligned}$$

Using the assumption evaluation analysis as presented in this section in combination with the results from a preceding reset analysis thus supports the process of timing model derivation, namely the environmental assumption refinement (cf. Section 5.2.1 on page 88). In the following, some problems, optimizations, and enhancements supporting the assumption evaluation analysis will be discussed.

6.3.1 Widening and Path-Sensitivity

Interval analysis as presented here has one limitation: depending on the program to be analyzed, termination of interval analysis is not guaranteed due to infinite ascending chains in the determination of the fixed-point.

Figure 6.6 on the next page shows the control flow graph of the 3-bit counter example shown in Listing 4.2 on page 67. Assumption evaluation using the assumption "reset is never active", and using the result of a preceding reset analysis ($cnt = 0$ in this example), yields the data-flow computation shown in the figure. Thereby, we focus only on value computation for the internal counter signal cnt. The several iterations of the fixed-point computation are shown as the evolution of data-flow information for the counter signal, i.e. a data-flow value trace $cnt \rightarrow [0,0]\ [0,1]\ [0,2]$ denotes a data-flow value of $[0,0]$ for the signal cnt in the first iteration, and an evolution to $[0,1]$ in the second iteration. The fixed-point is reached if for all edges in the control-flow graph the computed value range does not differ from the one of the previous iteration (cf. Definition 2.2.3 on page 13).

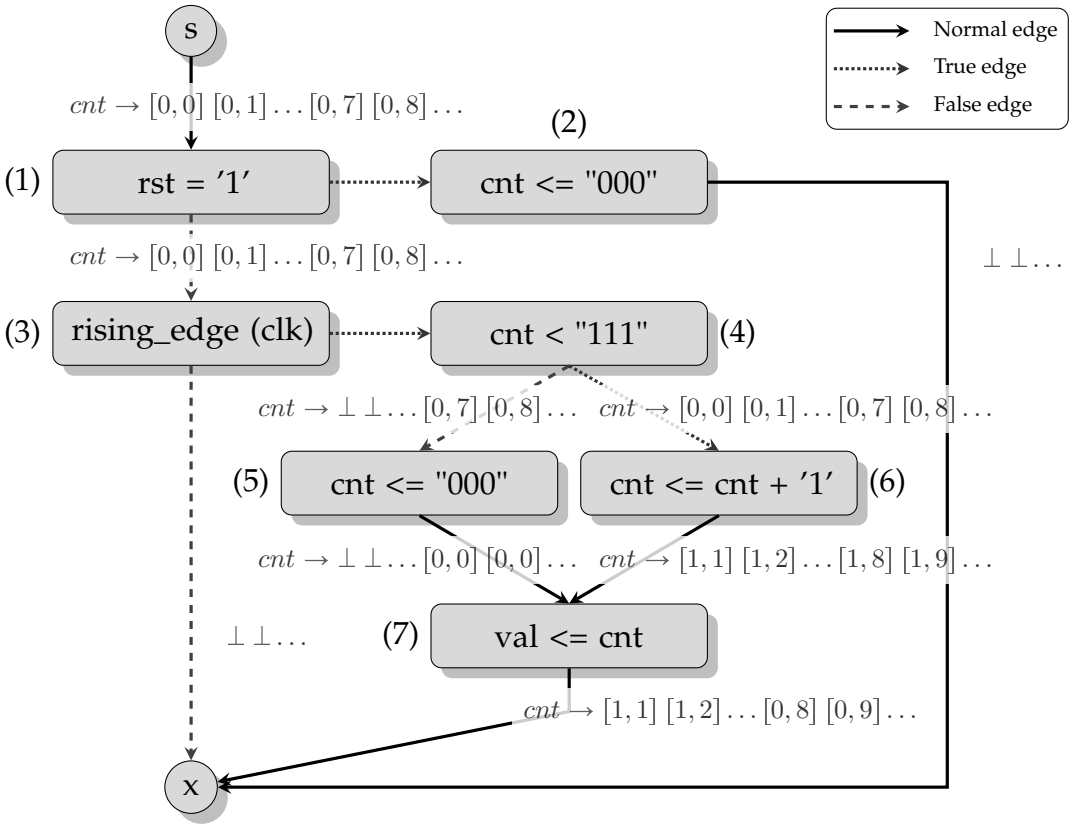


Figure 6.6 – Data-flow computation for the assumption "Reset is never active" on the 3-bit counter from Listing 4.2.

Starting with the result of the reset analysis, $cnt \to [0,0]$ in the above example, the then-branch of the conditional statement at node (1) is known to never be taken (due to the assumption "The reset is never active"). Thus, the data-flow value is propagated over the false-edge, and $\bot$ is propagated over the true-edge. The circuit in the example is synced on the rising edge of an external clock signal (cf. node (3)), thus, w. l. o. g., in the following, we focus on the rising edge of the clock process of the analysis framework.

At a falling clock edge, information is only propagated over the false-edge (edge from node (2) to the exit node) of the second conditional statement, thus no updates of the computed data-flow values occur.

In the first iteration, the conditional statement cnt < "111" at node (4) can be evaluated to $true$, thus, data-flow information is only propagated over the true-edge, whereas the least element $\bot$ is used on the else-branch. The counter increment cnt <= $cnt + 1$ schedules a transaction on cnt, and the increment leads to an updated value of $[1,1]$. Please note that in this example, the current value of cnt and its scheduled value $\overline{cnt}$ are treated the same due to readability

reasons. Furthermore, data-flow value update at the synchronize statement is handled by the update function of the simulation routine $transfer_{simul}$, and also the frequent change of the clock signal handled by $transfer_{clock}$ is only implicitly shown here. Finally, at the exit node of the control-flow graph, information is joined leading to $[1,1]$ as the value of cnt, which than has to be merged into the collecting contexts.

The next iteration uses $[0,1]$ as the possible value range of the counter signal. Now, the same rules as in the first iteration apply, leading to $[1,2]$ in the end, which has to merged in the collecting context, and the third round uses a value range of $[0,2]$.

In the 8th iteration using a value range of $[0,7]$, the conditional statement at node (4) cannot be evaluated deterministically anymore, so also the else-branch has to be taken into account, and the effects from branch parts need to be merged in front of node (7). From now on, all further iterations behave the same, leading to an infinite ascending chain, where the next iteration $i+1$ always differs from the previous one by

$$transfer_{Int}^{i+1}(\iota)(cnt) = transfer_{Int}^{i}(\iota)(cnt) + [0,1]$$

Consequently, the least fixed-point cannot be computed due to an infinite ascending chain, and the assumption evaluation analysis won't terminate. In order to ensure termination of the analysis, widening as described in Section 2.4.1 on page 23 can be used. Intuitively, the idea underlying widening is that it is always safe to take a fixed point that is greater than the least fixed point. Therefore, a widening operator needs to be defined that takes the old and the new values of an iteration sequence such that the result is always greater or equal than the new value. This approach reduces the length of the iteration sequence at the cost of precision.

For the assumption evaluation analysis using intervals, we can therefore define a widening operator $\bigtriangledown_{V_{Int}} : V_{Int} \times V_{Int} \to V_{Int}$ as follows:

$$\bigtriangledown_{V_{Int}}(A,B) = \begin{cases} \top & \text{if } A = \top \vee B = \top, \\ A & \text{if } B = \bot, \\ B & \text{if } A = \bot, \\ \top & \text{if } \underline{A} = \underline{B} \wedge \overline{A} < \overline{B} \wedge \underline{A} \neq \overline{A}, \\ \top & \text{if } \overline{A} = \overline{B} \wedge \underline{A} > \underline{B} \wedge \underline{A} \neq \overline{A}, \\ \top & \text{if } \underline{A} > \underline{B} \wedge \overline{A} < \overline{B} \wedge \underline{A} \neq \overline{A}, \\ A \sqcup_{V_{Int}} B & \text{otherwise.} \end{cases}$$

Whenever a bound is increased in the current iteration, and the interval in the last iteration was not precise, we set the resulting interval to $\top$. Obviously, the

widening operator computes an upper bound for all pairs $l, m \in V_{Int}$ and for every ascending chain $l_1 \sqsubseteq_L l_2 \sqsubseteq_L \cdots$, the ascending chain $m_1 \sqsubseteq_L m_2 \sqsubseteq_L \cdots$, constructed by

$$\begin{aligned} m_1 &= l_1 \\ m_{i+1} &= m_i \bigtriangledown l_{i+1} \end{aligned}$$

stabilizes. Thus, the following lemma holds.

Lemma 6.3.1
$\bigtriangledown_{V_{Int}} : V_{Int} \times V_{Int} \to V_{Int}$ is a widening operator as defined in Definition 2.4.5 on page 23.

Therewith, it is possible to also define a widening operator $\bigtriangledown_{D_{Int}} : D_{Int} \times D_{Int} \to D_{Int}$ for the domain D_{Int} underlying the assumption evaluation analysis as follows:

$$\bigtriangledown_{D_{Int}}(l, m) = \begin{cases} \top & \text{if } l = \top \vee m = \top, \\ l & \text{if } m = \bot, \\ m & \text{if } l = \bot, \\ \lambda i \in identifier.\ l(i) \bigtriangledown_{V_{Int}} m(i) & \text{otherwise.} \end{cases}$$

Theorem 6.3.1 (Widening $\bigtriangledown_{D_{Int}}$)
$\bigtriangledown_{D_{Int}} : D_{Int} \times D_{Int} \to D_{Int}$ is a widening operator enforcing the termination of the assumption evaluation analysis.

The proof of this theorem is omitted here since $\bigtriangledown_{D_{Int}}$ obviously is an upper bound operator for the domain D_{Int}. As the set of identifiers that occur within a VHDL model is finite stabilization of ascending chains is already guaranteed by Lemma 6.3.1.

Figure 6.7 on the next page again shows the control-flow graph of the 3-bit counter example. Data-flow computation depicted in this figure uses widening to enforce termination of the fixed-point iteration. Before starting a new iteration, the widening operator as defined above is used to derive the new data-flow value. By this, the first two iteration rounds behave as before (i.e. as without using widening), but at the beginning of the third round, the old data-flow value of $[0, 1]$ is widened with the new data-flow value of $[0, 2]$, which leads to a new (greater) data-flow value of $\top$. The fixed point is reached after the third iteration.

However, results of the assumption evaluation analysis might become unsatisfactory due to the use of widening. Thus, it is possible to define an enhanced transfer function that augments the assumption evaluation analysis to be path sensitive. Figure 6.8 on page 138 depicts the data-flow value computation

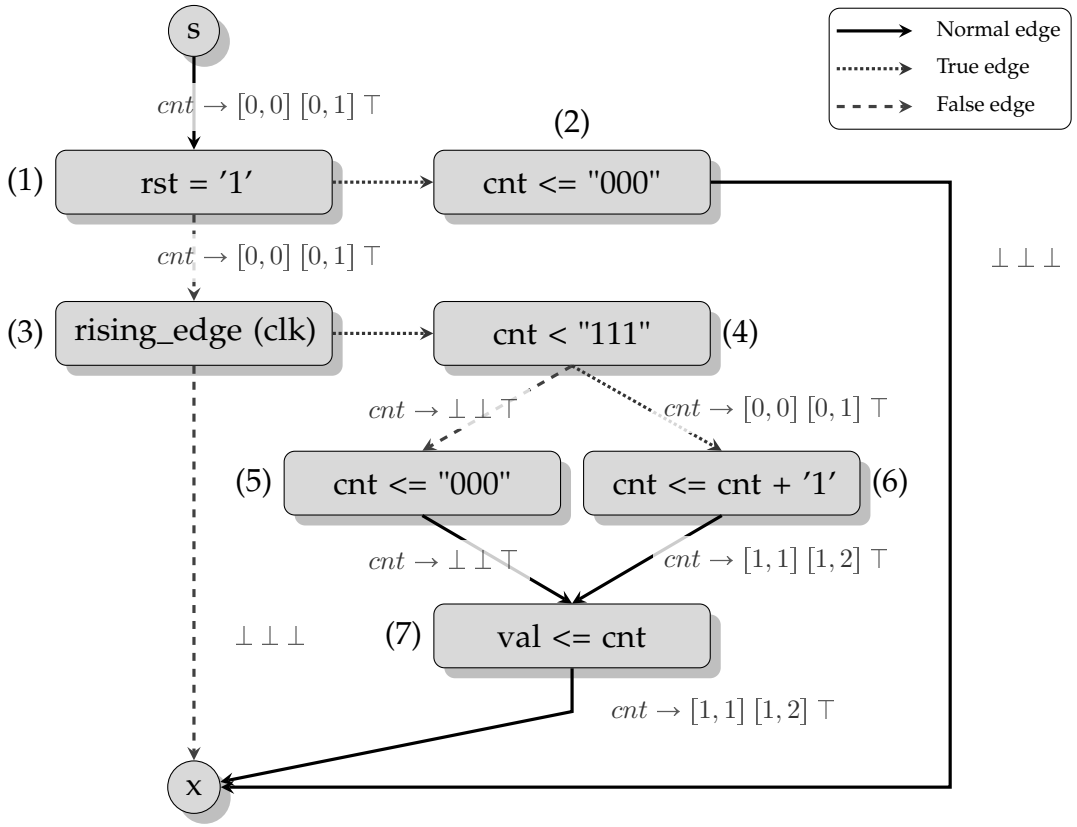


Figure 6.7 – Data-flow computation using widening.

using an enhanced transfer function that uses additional statically available information to compute more precise values.

The idea behind the enhanced transfer function is that whenever a conditional statement is encountered, the value range for the identifier used for comparison is restricted with the solution space of the condition, the edge type, and the identifier's type range[3].

So, within the example, the benefit of adding path sensitivity to the assumption evaluation analysis is visible within the third iteration. Widening still ensures termination by "cutting off" the infinite ascending chain for the value range of cnt, but the conditional statement $cnt <$ "111" allows to refine the value range of cnt depending on the edge type. From the specification (cf. Listing 4.2), cnt is known to be of an unsigned type of 3-bit width, and the condition ensures that within the then-branch, the value must be less than "111", thus it is safe to propagate a data-flow value of $[0, 6]$ for cnt along the taken edge, whereas on the fall-through edge, cnt is known to be of value $[7, 7]$. So, adding path sensitivity

[3]Within VHDL, every type has a defined range that is known statically. The range of a type T is defined as $[T'low, T'high]$.

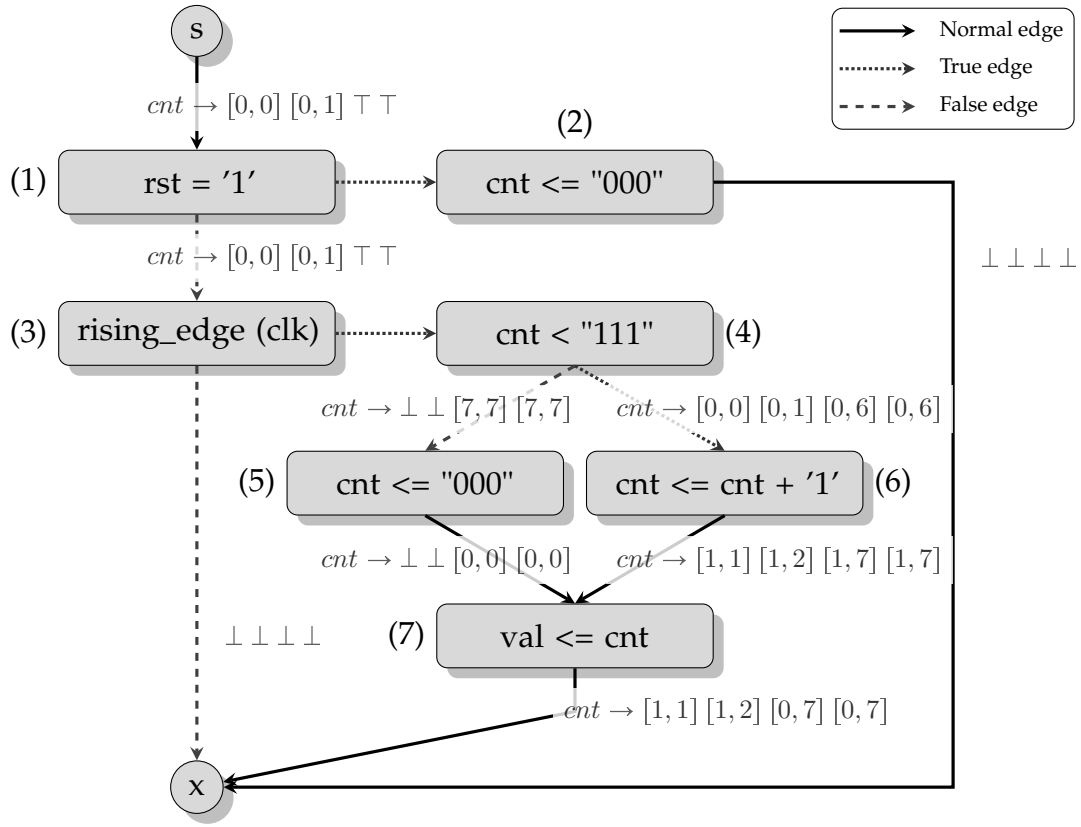


Figure 6.8 – Data-flow computation using widening and the enhanced transfer function.

to the analysis allows to refine information according to the information known at conditional statements.

Please note that the use of this enhancement is currently limited to "simple" conditional statements, i.e. compare statements of the kind

$$< \texttt{simple\_conditional} > \equiv < \texttt{identifier} > < \texttt{relation} > < \texttt{expression} >,$$

whereas in general, additional computational effort for rearranging the conditional expressions is required.

Thus, we can define an enhanced transfer function $transfer_{if_{simple}} : D_{Int} \times E_{\text{VHDL}} \rightarrow D_{Int}$ for a simple conditional statement $i\ rel\ expr$ on a lattice element l and an edge e. Let $A = \{x \mid x\ rel\ expr\}$ be the solution space of the (in-)equation $x\ rel\ expr$, and $B = \{x \mid x\ !rel\ expr\}$ be the solution space of the (in-)equation $x\ !rel\ expr$, and let $A^{[\,]}$ and $B^{[\,]}$ be the interval hulls of the sets A and B, respectively, i.e. $A^{[\,]} = [v, w]$, such that $\forall a \in A\colon v \leqslant a \leqslant w$ and $v, w \in A$. Furthermore,

```
   ⋮
DP_TakeExternalInterrupt <= InterruptRequest
                            and (DP_InterruptEnableFlag
                                 or DP_ExecuteRfe);
   ⋮
```

Listing 6.11 – Sample VHDL snippet of the superscalar DLX machine.

let T_i be the type of identifier i.

$$transfer_{if_{\,simple}}(l, e) =$$

$$\lambda j \in identifier. \begin{cases} l(i) \sqcap_{V_{Int}} [T_i'low, T_i'high] \sqcap_{V_{Int}} A^{[\,]} & \text{if } j = i \wedge cat(e) = true, \\ l(i) \sqcap_{V_{Int}} [T_i'low, T_i'high] \sqcap_{V_{Int}} B^{[\,]} & \text{if } j = i \wedge cat(e) = false, \\ l(j) & \text{otherwise.} \end{cases}$$

This function can be used within the update function $transfer_{if}$ in order to refine the results of the assumption evaluation analysis.

6.3.2 Live-Variables Analysis

Using the results of the assumption evaluation analysis, a subsequent *live-variables analysis* can be used to identify further timing-dead statements. The live-variables analysis is described here in more detail.

Listing 6.11 shows a small excerpt of the VHDL specification of the superscalar DLX machine available from the TU Darmstadt [Hor97]. As it is often the case in timing analysis, the effects of external interrupts occurring in a non-deterministic sporadic manner cannot be covered, thus the external signal $InterruptRequest$ is assumed to be of a constant value, namely $'0'$. Thus, the statement depicted in the listing can be evaluated statically, and also the signal $DP\_TakeExternalInterrupt$ becomes static. By this, the whole statement can be marked as timing dead by using the results of the assumption evaluation analysis. If there are no further read references of the signals $DP\_InterruptEnableFlag$ and $DP\_ExecuteRfe$ that are also used in the assignment statement, also the nodes, where their values are assigned, can be marked as timing dead. The transitive closure of this problem can be automatically determined by the *live-variables analysis*.

An identifier is *live* behind a statement if there exists a path from the statement to a use of the identifier that does not redefine the identifier. Thus, the live-variables analysis will determine for each program point of a control-flow graph, which identifiers may be live at the exit of the point.

The live-variables analysis is a well-known analysis in the area of compiler construction used as the basis for dead code elimination. The analysis belongs to the class of *bit vector problems* in which the data-flow value is a set, e.g., the set of live variables. These sets can be represented efficiently as bit vectors, in which each bit represents set membership of one particular element. Using this representation, the join and transfer functions can be implemented as bitwise logical operations. For the live-variables analysis, the join operation is the union, implemented by bitwise logical or. The transfer function for each type of statement can be decomposed in so-called *gen* and *kill sets*.

The kill set for live-variables analysis is the set of identifiers that are written in a particular statement, whereas the gen set is the set of identifiers that are read without being written first. A formal definition of the data-flow problem to determine live variables is given in [NNH99] and is therefore omitted here. Please note that nodes (i.e. statements) that have been already identified as timing dead are skipped while determining the live variables (as these statements are known to not contribute any longer). This analysis is a backward analysis, where information is build up from the information at the control-flow successors. At joins, the union of the incoming information is formed.

The maximal fixed-point solution MFP_{lv} for this data-flow problem can now be used for determining further timing-dead identifiers as described above. Thus, given a control-flow graph $G = (V, E, s, x)$, all assignment statement nodes assigning a value to an identifier that is not live at that node can be marked as timing dead.

$$\begin{aligned} &\forall n \in V : cat(n) = signalassignment \wedge (def(n) \cap MFP_{lv}(n)) = \varnothing \\ &\implies timing\_dead(n) = true; \end{aligned}$$

By this, a further reduction in terms of size and complexity of a given VHDL specification can be achieved.

6.4 Backward Slicing

A *program slice* of a program P consists of all parts of P that potentially affect the values computed at some point of interest, referred to as the *slicing criterion* C. All parts of P that have a direct or indirect influence on the criterion C are called *program slice with respect to* C.

The original concept of *program slicing* – the task to compute program slices – was introduced by Weiser [Wei79, Wei81, Wei82, Wei84]. Weiser argues that a slice corresponds to abstractions that programmers make in their mind while debugging a program, and therefore suggests to integrate slicing into debugging environments. Building on the definitions of Weiser, various slightly different

```
1 read(n);
  int fac = 1;
3 int sum = 0;
  while (n >= 0) {
5   if (n == 0)
      fac = fac * 1;
7   else
      fac = fac * n;
9   sum = sum + n;
    n--;
11 }
  write(fac);
13 write(sum);
```

(a)

```
read(n);
int fac = 1;

while ( n >= 0 ) {
  if ( n == 0 )
    fac = fac * 1;
  else
    fac = fac * n;

  n--;
}
```

(b)

```
    fac = fac * 1;

    fac = fac * n;


write(fac);
```

(c)

Listing 6.12 – Example (a), backward (b) and forward slice (c) for criterion $(6, \{fac\})$.

notions of program slices and methods for program slicing have been proposed. This is mainly due to the fact that different applications require slightly different characteristics of slices. According to Weiser, a program slice is a reduced executable program derived from the original program P by removing statements, such that the slice replicates parts of the behavior of P. Building on the definition of Weiser, Ottenstein et al. restate the problem of slicing in terms of a reachability problem in a *program dependency graph* [KKP+81, OO84, FOW87], a directed graph where nodes represent statements and control predicates, and edges correspond to data and control dependencies. The definition of a slice is changed to a subset of the statements and control predicates of the program which directly or indirectly affect the values computed at the slicing criterion, but the subset must not necessarily constitute an executable program.

Besides the slightly different definitions, an important distinction is that between *static* and *dynamic* slices. Static slices are computed without making assumptions regarding the program's input and environment, whereas dynamic slices rely on some specific input vector. As this thesis is concerned with the static analysis of hardware description languages, dynamic slicing will not be further investigated here, and is just mentioned for completeness reasons. A detailed discussion of dynamic as well as static slicing, different approaches for their computation and a survey on different slicing definitions can be found in [Tip95].

Slices mentioned so far are computed by collecting statements and control predicates by means of a *backward* traversal of the program, starting at the given

slicing criterion. Listing 6.12 (b) shows a *backward slice* of the program given in Listing 6.12 (a) with respect to slicing criterion $(6, \{fac\})$. Usually, the slicing criterion consists of a tuple (program point, set of variables). In the example mentioned here, the program point is referred to as the line-number of the original program. Besides this classical backward slicing, Bergeretti and Carré had formally introduced the term *forward slicing* [BC85]. Intuitively, a forward slice for a slicing criterion C consists of all statements and control predicates of a program that depend on C. A statement "depends" on a slicing criterion C, if the values computed at that statement depend on a value computed at C, or the execution of the statement under consideration depends on a value computed at C. An example of a forward slice with respect to the slicing criterion $(6, \{fac\})$ is depicted in Listing 6.12 (c).

For timing model derivation, understanding a VHDL specification is of utmost importance. Finding abstractions suitable for the given model in order to make timing analysis feasible requires a detailed view in the several components forming a process while not losing the overall view of the dependencies contained in the specification.

Slicing as introduced by Weiser offers the possibility to focus on more strategic decisions. Furthermore, model preprocessing as described in Section 5.2.1 on page 88 introduced the concept of timing dead code elimination. As stated there, backward slicing with respect to program points where instructions leave the processor's pipeline results in a slice, whose inversion (i.e. statements not in the slice become part of it, and vice versa) represents all statements that are not relevant with respect to the timing behavior of a processor. Please note that depending on the processor specification to be analyzed, there might exist several points where instructions will leave the pipeline, thus the *union* of slices for each of these points needs to be formed before building the inversion.

In the following, program slicing using program dependencies will be introduced. Furthermore, an extension due to the two-level semantics of hardware description languages will be described. Building on that, data-flow problems for reconstructing these dependencies using the analysis framework introduced at the beginning of this chapter will be defined. Also the resulting slicing algorithm and some further optimizations will be described.

[CFR+99, CFR+02] describe a similar approach for slicing of hardware description languages. They map language constructs from VHDL to language constructs of traditional procedural languages like C or ADA. The reactive nature being special to hardware description languages is modeled via a special master process.

Another method to compute slices on synchronous circuits is described in [RKK04] focusing on the event-oriented communication structure of VHDL. Slicing as presented here is not limited to synchronous circuit specifications.

6.4.1 Slicing Using Dependencies

In [OO84], Ottenstein and Ottenstein presented a slicing approach based on a program dependency graph. Horwitz, Reps, and Binkley [HRB88] extended this approach to compute interprocedural, context-sensitive slices. This section briefly introduces the approaches presented there and extends them to cope with the two-level semantics being special for hardware description languages.

Definition 6.4.1 (Slicing criterion)
Let $G = (V, E, s, x)$ be the control-flow graph of a program P. A tuple $C = (n, U)$ with $n \in V$ and $U \subseteq \mathit{def}(n) \cup \mathit{use}(n)$ is called a *slicing criterion*.

Definition 6.4.2 (Slice)
Let $G = (V, E, s, x)$ be the control-flow graph of a program P. A *slice* with respect to a slicing criterion $C = (n, U)$ is a subset $S \subseteq V$ such that the following holds: If P halts on input I, then the value of $u \in U$ at the statement represented by n each time n is executed in P is the same in P and P_S. If P fails to terminate normally, n may execute more times in P_S than in P, but P and P_S compute the same values each time n is executed by P.

Obviously, for any program P represented by $G_P = (V, E, s, x)$ and for any slicing criterion (n, U), there exists at least one slice S, namely the identity $S = V$. In most cases, this conservative solution is not very satisfactory. Thus, we are looking for a subset $S \subseteq V$, where S is approximately minimal.

Definition 6.4.3 (Minimal slice)
A slice S with respect to a slicing criterion $C = (n, U)$ is called *minimal*, if there exists no other slice S' with respect to C, where $|S'| < |S|$ holds.

According to Weiser, minimal slices are not necessarily unambiguous and the problem of determining minimal slices is not decidable in general. In the following, this thesis presents the theory of an iterative algorithm to compute approximations to minimal slices.

Using the CRL2-attributes def and use, several types of *data dependence* can be defined, such as flow dependence, output dependence and anti dependence [FOW87]. For slicing, only flow dependence is taken into account.

Definition 6.4.4 (Flow dependence $\Downarrow$)
Let $G = (V, E, s, x)$ be a control-flow graph. A node $n \in V$ is *flow dependent* on a node $m \in V$, if there exists an identifier i such that:

$$
\begin{aligned}
& i \in \mathit{def}(m), \\
\wedge \quad & i \in \mathit{use}(n), \\
\wedge \quad & \exists \pi \in P[m, n] \colon \forall u \in \pi \backslash \{m, n\} \colon i \notin \mathit{def}(u)
\end{aligned}
$$

We use the relation $\Downarrow_i$ to express this dependency: $m \Downarrow_i n$. The set of nodes defining an identifier i for a node $n \in V$ can be defined as

$$data(n, i) = \{m \mid m \in V \wedge m \Downarrow_i n\}$$

If the execution of a program statement depends on the execution of another program statement, this is called *control dependency*. A control dependence is usually defined in terms of *postdominance*.

Definition 6.4.5 (Postdomination $\uparrow$)
Let $G = (V, E, s, x)$ be a control-flow graph. A node $n \in V$ is *postdominated* by a node $m \in V$, if all paths from n to the exit node of the control-flow graph contain the node m.

$$\forall \pi \in P[n, x] \colon m \in \pi$$

We will use $\uparrow$ to denote this relation: $m \uparrow n$. The postdominator set for a node $n \in V$ thus is defined as:

$$pdom(n) = \{m \mid m \in V \wedge m \uparrow n\}$$

Using this definition, control dependence can be defined as follows:

Definition 6.4.6 (Control dependence $\downarrow$)
Let $G = (V, E, s, x)$ be a control-flow graph. A node $n \in V$ is *control dependent* on a node $m \in V$, iff

$$\begin{aligned} & \exists \pi \in P[m, n] \colon \forall u \in \pi \backslash \{m, n\} \colon n \in pdom(u) \\ \wedge \quad & n \notin pdom(m) \end{aligned}$$

We will use $\downarrow$ to express this relation: $m \downarrow n$. The set of control-dependent nodes for a node $n \in V$ can be defined as

$$infl(n) = \{u \mid u \in V \wedge n \downarrow u\}$$

$infl(n)$ is also called the *range of influence* of the node n.

Using the definitions 6.4.4 and 6.4.6, an approximation to minimal slices can be found by computing successive sets of *relevant identifiers* for each node of a control-flow graph.

Definition 6.4.7 (Directly relevant identifiers)
Let $G = (V, E, s, x)$ be a control-flow graph and $C = (n, U)$ be a slicing criterion. The set of *directly relevant identifiers* R^0_C for a node $o \in V$ is defined as:

$$R^0_C(o) \equiv \begin{cases} U & \text{if } o = n, \\ \{i \mid \forall (o, p) \in E \colon (i \in R^0_C(p) \wedge i \notin def(o)) & \\ \quad \vee (i \in use(o) \wedge def(o) \cap R^0_C(p) \neq \varnothing)\}, & \text{otherwise.} \end{cases}$$

Building on this definition, the set of directly relevant statements is defined as:

Definition 6.4.8 (Directly relevant statements)
Let $G = (V, E, s, x)$ be a control-flow graph and $C = (n, U)$ be a slicing criterion. The set of *directly relevant statements* S_C^0 is defined as the set of all nodes $o \in V$ which define an identifier i that is relevant for a successor of o in the control-flow graph.

$$S_C^0 \equiv \{o \mid \exists (o, p) \in E \colon def(o) \cap R_C^0(p) \neq \varnothing\}$$

Building on the definitions of R_C^0 and S_C^0, the sets R_C^k, S_C^k, B_C^k and A_C^k for $k \geqslant 0$ will be defined by induction.

Identifiers that are referenced in the control predicates of branch statements are indirectly relevant, if at least one of the statements being control dependent on the branch statement is relevant. The execution of a slicing criterion also depends on all nodes on which the criterion is control dependent. In general, control dependencies have to be considered for all relevant statements. Thus, the set of relevant control statements can be defined as follows:

Definition 6.4.9 (Relevant control statements)
The set of *relevant control statements* B_C^k which are relevant due to the influence they have on nodes n in S_C^k are:

$$B_C^k \equiv \{b \mid \mathit{infl}(b) \cap S_C^k \neq \varnothing\}$$

Within VHDL, execution of a process is "controlled" via its sensitivity list. Thus also the signals used in the sensitivity lists induce a dependency that has to be taken into account. Within the analysis framework presented in this thesis, the sensitivity list of a process can be accessed via the $sensitivity$ attribute of its starting node s_i. Section 6.1 on page 98 has already shown that process (re-)activation can be modeled using the if-guard statement Ξ, however, for slicing, adding nodes induced by the framework to the set of relevant statements with respect to a slicing criterion C is not desirable. Hence, *activation dependencies* due to sensitivity lists are handled by defining the set of relevant activation statements A_C^k.

Definition 6.4.10 (Relevant activation statements)
Let $G = (V, E, s, x)$ be a supergraph of a VHDL model derived from the analysis framework, i.e. G is constructed from processes $P_0, P_1, \ldots, P_n$ and the additional framework code. The set of *relevant activation statements* A_C^k which are relevant for the activation of a process P_i represented via $G_i = (V_i, E_i, s_i, x_i)$ is defined as:

$$A_C^k \equiv \{s_i \mid \exists n \in S_C^k, n \in V_i\}$$

Using the definitions from above, we can define the set of indirectly relevant identifiers.

Definition 6.4.11 (Indirectly relevant identifiers)
The set of indirectly relevant identifiers R_C^{k+1} is defined by considering the identifiers in the predicates of the control statements B_C^k and the signals used in the sensitivity lists of the affected processes to be relevant.

$$R_C^{k+1}(o) = R_C^k(o) \cup \bigcup_{b \in B_C^k} R^0_{(b,use(b))}(o) \cup \bigcup_{a \in A_C^k} R^0_{(a,sensitivity(a))}(o)$$

Definition 6.4.12 (Indirectly relevant statements)
The set of *indirectly relevant statements* S_C^{k+1} consists of all nodes in B_C^k together with the nodes o defining an identifier that is relevant to a control-flow successor p:

$$S_C^{k+1} = B_C^k \cup \{o \mid \exists (o,p) \in E, def(o) \cap R_C^{k+1}(p) \neq \varnothing\}$$

Please note that nodes in A_C^k do not belong to the set of indirectly relevant statements, since the activation dependency induced by the sensitivity list of a process is only reflected implicitly. The semantics of VHDL does not require an identifier contained in the sensitivity list of a process to be used within the process body.

Both sequences $\left(R_C^k\right)_{k \in \mathbb{N}}$ and $\left(S_C^k\right)_{k \in \mathbb{N}}$ represent monotonically increasing subsets of identifiers and statements of the VHDL model, respectively; the fixed point of the sequence $\left(S_C^k\right)_{k \in \mathbb{N}}$ constitutes the desired program slice with respect to the slicing criterion C.

Theorem 6.4.1 (Correctness)
Let $G = (V, E, s, x)$ be the control-flow graph of a VHDL model derived from the analysis framework, and $C = (n, U)$ with $n \in V$ and $U \subseteq def(n) \cup use(n)$ be a slicing criterion. The fixed point of the sequence $\left(S_C^k\right)_{k \in \mathbb{N}}$ is a safe slice with respect to the slicing criterion C.

The definition of S_C^k provides an iterative approach for its computation resulting in an approximation of a minimal slice. Termination in computing the fixed point is guaranteed because the set of nodes is finite. The following subsections now describe static analyses that aid in computing the dependencies required for instantiating the slicing algorithm.

6.4.2 Flow-Dependency Analysis

Reconstruction of data dependencies, or, to be more precise, of flow dependencies can be easily formulated as a data-flow problem by computing the reaching definitions for each node in the control-flow graph.

A definition *reaches* a program point if there is a path from the definition to this point without any redefinition. Let $G = (V, E, s, x)$ be a control-flow graph. An identifier i defined at node $m \in V$ reaches a node $n \in V$, iff

$$\begin{aligned} & i \in def(m) \\ \wedge \quad & \exists \pi \in P[m,n] \colon \forall o \in \pi \backslash \{m,n\} \colon i \notin def(o) \end{aligned}$$

Obviously, the result of a data-flow analysis building on this definition yields a superset of the flow dependency defined in Definition 6.4.4 on page 143.

Computing reaching definitions thus requires a mapping f defined on the set $identifier$ of identifiers used in the VHDL specification such that $f(i)$ is the set of nodes in the control-flow graph defining identifier i.

$$map \equiv \{f \mid f \colon identifier \to \mathcal{P}(V)\}$$

with the pointwise ordering relation $\sqsubseteq_{map}$ for $f, g \in map$ defined as

$$f \sqsubseteq_{map} g \iff \forall i \in identifier \colon f(i) \subseteq g(i)$$

The combine and meet functors $\sqcup_{map}$ and $\sqcap_{map}$ for $f, g \in map$ are defined as:

$$\begin{aligned} f \sqcup_{map} g &\equiv \lambda i \in identifier.\ f(i) \cup g(i) \\ f \sqcap_{map} g &\equiv \lambda i \in identifier.\ f(i) \cap g(i) \end{aligned}$$

The domain used for reaching-definitions analysis consists of this function domain map extended by additional least and greatest elements $\bot$ and $\top$ easing further optimizations

$$D_{rd} \equiv map \cup \{\bot, \top\}$$

ordered such that

$$\begin{aligned} & \forall f \in D_{rd} \colon && \bot \sqsubseteq_{D_{rd}} f \sqsubseteq_{D_{rd}} \top \\ \wedge \quad & f, g \in map \colon && f \sqsubseteq_{D_{rd}} g \iff f \sqsubseteq_{map} g \end{aligned}$$

The join and meet functors $\sqcup_{D_{rd}}$ and $\sqcap_{D_{rd}}$ for $x, y \in D_{rd}$ are defined as follows:

$$x \sqcup_{D_{rd}} y \equiv \begin{cases} \top & \text{if } x = \top \vee y = \top, \\ x & \text{if } y = \bot, \\ y & \text{if } x = \bot, \\ x \sqcup_{map} y, & \text{otherwise.} \end{cases}$$

$$x \sqcap_{D_{rd}} y \equiv \begin{cases} \bot, & \text{if } x = \bot \vee y = \bot, \\ x, & \text{if } y = \top, \\ y, & \text{if } x = \top, \\ x \sqcap_{map} y, & \text{otherwise.} \end{cases}$$

For the reaching definitions, information being valid on one path through the control-flow graph is interesting, thus, the union of incoming information is formed at control-flow joins and information is propagated in forward direction. Thus, the lattice used for the data-flow problem is

$$(D_{rd}, \sqsubseteq_{D_{rd}}, \sqcup_{D_{rd}}, \sqcap_{D_{rd}}, \bot, \top)$$

The domain chosen for analysis maps identifiers used in the program to program points, where this identifier potentially is to be modified. Whereas signal and variable assignments of *scalar* types target the whole identifier, modifications of composite types (i.e. bitvectors in VHDL) may only update a "part" of the identifier, e.g., a field in an array. VHDL offers two classes of composite types, arrays and records. Whereas record types are non-interesting here[4], VHDL additionally offers the possibility to modify a consecutive part of an array type at once, which is called an array slice. For slicing, assignment to composite types does not necessarily write the content that is to be read later on. Thus, we define two functions $must\colon D_{rd} \times identifier \times V \to D_{rd}$ and $may\colon D_{rd} \times identifier \times V \to D_{rd}$ as follows:

Definition 6.4.13 ($must$ update)
Let $G = (V, E, s, x)$ be a control-flow graph. The *must update* $must\colon D_{rd} \times identifier \times V \to D_{rd}$ for an identifier $i \in identifier$ and a node $n \in V$ on a lattice element $l \in D_{rd}$ is defined as:

$$must(l, i, n) = l[i \leftarrow \{n\}]$$

Definition 6.4.14 (may update)
Let $G = (V, E, s, x)$ be a control-flow graph. The *may update* $may\colon D_{rd} \times identifier \times V \to D_{rd}$ for an identifier $i \in identifier$ and a node $n \in V$ on a lattice element $l \in D_{rd}$ is defined as:

$$may(l, i, n) = l[i \leftarrow l(i) \cup \{n\}]$$

Using the definitions of $must$ and may, it is possible to define the update function $transfer_{\text{VHDL}}\colon D_{rd} \times E_{\text{VHDL}} \to D_{rd}$ for an edge $e = (m, n) \in E_{\text{VHDL}}$ on a lattice element $l \in D_{rd}$ as follows:

$$transfer_{\text{VHDL}}(l, e) = \lambda i \in identifier. \begin{cases} may(l, i, m) & \text{if } i \in def(m) \wedge composite(type(i)) = true, \\ must(l, i, m) & \text{if } i \in def(m) \wedge composite(type(i)) \neq true, \\ l(i) & \text{otherwise.} \end{cases}$$

[4]The elaboration process already "flattens" these structures in order to enhance synthesizability of the given model.

This update function only deals with statements being directly derived from the original VHDL model. In order to define a data-flow problem, also updates for the routines derived from the analysis framework need to be given. Since these statements do not define any signal or variable (except the external clock signal that cannot be modeled using the synthesizable VHDL subset), these statements do not contribute to the reaching definition analysis.

Thus, the update function $transfer_{other}\colon D_{rd} \times E \backslash E_{\text{VHDL}} \to D_{rd}$ for a given control-flow graph $G = (V, E, s, x)$ on a lattice element $l \in D_{rd}$ and an edge $e \in E \backslash E_{\text{VHDL}}$ can be easily defined as:

$$transfer_{other}(l, e) = l$$

Given a control-flow graph $G = (V, E, s, x)$, we can now define the transfer function $transfer_{rd,e}\colon D_{rd} \to D_{rd}$ for an edge $e = (m, n) \in E$ on a lattice element $l \in D_{rd}$ as:

$$transfer_{rd,e}(l) = \begin{cases} transfer_{\text{VHDL}}(l, e) & \text{if } e \in E_{\text{VHDL}}, \\ transfer_{other}(l, e) & \text{otherwise.} \end{cases}$$

The return function $return\colon D_{rd} \times D_{rd} \to D_{rd}$ is defined as:

$$return_{rd} = \sqcup_{D_{rd}}$$

Now, it is possible to define a function $\mathfrak{f}_{rd}\colon E \to (D_{rd} \to D_{rd})$ for an edge $e \in E$ as

$$\mathfrak{f}_{rd}(e) = transfer_{rd,e}$$

and use it as the transfer function of the data-flow problem $dfp_{rd} = (G, D_{rd}, \mathfrak{f}_{rd})$.

The maximal fixed-point solution MFP_{rd} of dfp_{rd} starting with the initial element $\iota \equiv \lambda i \in identifier.\ \{\ \}$ can now be used to access a superset of the definitions reaching a program point.

Theorem 6.4.2 (Soundness of dfp_{rd})
The data-flow problem $dfp_{rd} = (G, D_{rd}, \mathfrak{f}_{rd})$ computes a superset of the flow dependencies of the VHDL model described by G. MFP_{rd} is a sound approximation of MOP_{rd} and the following holds:

$$\forall n \in V\colon \forall i \in identifier\colon m \in data(n, i) \implies m \in MFP_{rd}(n)(i)$$

Proof 6.4.1
In order to prove this theorem, the following two propositions need to be shown:

1. dfp_{rd} is a monotone data-flow problem $\implies$ MFP_{rd} is a sound approximation of MOP_{rd}.
2. $\forall n \in V\colon \forall i \in identifier\colon m \in data(n, i) \implies m \in MFP_{rd}(n)(i)$

Let $G = (V, E, s, x)$ be a control-flow graph and $dfp_{rd} = (G, D_{rd}, \mathfrak{f}_{rd})$ be the data-flow problem as described above.

to 1. The data-flow problem dfp_{rd} is monotone, iff $\forall e \in E\colon \mathfrak{f}_{rd}(e)$ is monotone (cf. Definition 2.4.2 on page 20).
By definition of $\mathfrak{f}_{rd}$, this property only depends on the monotonicity of the function $transfer_{rd,e}\colon D_{rd} \to D_{rd}$. By definition, $transfer_{rd,e}$ is monotone.
$\Longrightarrow$ dfp_{rd} is monotone.
$\Longrightarrow$ MFP_{rd} is a sound approximation of MOP_{rd} (cf. Theorem 2.4.1 on page 21).

to 2. Let $n \in V$, and $i \in identifier$.

$$
\begin{array}{lll}
 & & m \in data(n, i) \\
\Longrightarrow & & m \Downarrow_i n \\
\Longrightarrow & & i \in def(m) \\
 & \wedge & i \in use(n) \\
 & \wedge & \exists \pi \in P[m, n] : \forall u \in \pi \backslash \{m, n\}\colon i \notin def(u) \\
\Longrightarrow & & m \in MFP_{rd}(n)(i)
\end{array}
$$

□

6.4.3 Control-Dependency Analysis

The reconstruction of control dependencies is analogous to the two-level definition given in Definition 6.4.6. It can be computed by two successive standard data-flow analyses, namely a postdominator and a dominator analysis.

The domain underlying both analyses is defined as

$$D_{ctrl} \equiv \mathcal{P}(V) \cup \{\bot, \top\}$$

representing the set of nodes of the control-flow graph $G = (V, E, s, x)$ dominating or postdominating the actual node. The additional greatest and least elements $\top$ and $\bot$ are used for further optimizations described later on.

The order of the set D_{ctrl} is defined as:

$$
\begin{array}{lll}
 & \forall x \in D_{ctrl} : & \bot \sqsubseteq_{D_{ctrl}} x \sqsubseteq_{D_{ctrl}} \top \\
\wedge & x_1, x_2 \in \mathcal{P}(V) : & x_1 \sqsubseteq_{D_{ctrl}} x_2 \Longleftrightarrow x_1 \subseteq x_2
\end{array}
$$

The join and meet operators $\sqcup_{D_{ctrl}}$ and $\sqcap_{D_{ctrl}}$ are defined analogously to the operators $\sqcup_{D_{rd}}$ and $\sqcap_{D_{rd}}$ described in Section 6.4.2 on page 146.

Postdominator Analysis

The postdominator analysis is used to directly compute the postdominator set $pdom$ as defined in Definition 6.4.5 on page 144. For each node $n \in V_{\text{VHDL}}$ of a control-flow graph $G = (V, E, s, x)$, we are interested in the set of successor nodes that is contained in every path from n to the exit node x. For the postdominator analysis, we are interested in information on the successors of the actual node, thus, the postdominator analysis is a backward data-flow problem. The result must be valid on all paths through a control-flow graph, so at control-flow joins, the intersection of incoming information is computed. The resulting lattice for the postdominator analysis hence is defined as

$$(D_{ctrl}, \sqsupseteq_{D_{ctrl}}, \sqcap_{D_{ctrl}}, \sqcup_{D_{ctrl}}, \top, \bot)$$

The update function $transfer_{pdom,e}\colon D_{ctrl} \to D_{ctrl}$ for an edge $e = (m, n) \in E$ of a control-flow graph $G = (V, E, s, x)$ representing a VHDL model derived from the analysis framework on a lattice element $l \in D_{ctrl}$ is defined as

$$transfer_{pdom,e}(l) = \begin{cases} l \sqcup_{D_{ctrl}} \{m\} & \text{if } m \in V_{\text{VHDL}}, \\ l & \text{otherwise.} \end{cases}$$

The return function $return_{pdom}\colon D_{ctrl} \times D_{ctrl} \to D_{ctrl}$ used to combine results at function calls in the interprocedural control-flow graph is defined as

$$return_{pdom} = \sqcup_{D_{ctrl}}$$

Now, we can define the transfer function $\mathfrak{f}_{pdom} : E \to (D_{ctrl} \to D_{ctrl})$ for any edge $e \in E$ as

$$\mathfrak{f}_{pdom}(e) = transfer_{pdom,e}$$

and obtain the data-flow problem $dfp_{pdom} = (G^{-1}, D_{ctrl}, \mathfrak{f}_{pdom})$, whose maximal fixed-point solution $MFP_{pdom}(n)$ for a node $n \in V$ yields a superset of the nodes of the postdominator set $pdom(n)$.

Dominator Analysis

The second analysis, namely the dominator analysis, is used to restrict the number of nodes in a control-flow graph that *may be* control nodes. In order to define this analysis, it is necessary to introduce the concept of domination.

Definition 6.4.15 (Domination)
Let $G = (V, E, s, x)$ be a control-flow graph. A node $m \in V$ *dominates* a node $n \in V$, iff

$$\begin{aligned} & \exists \pi \in P[m, x] \colon n \in \pi \\ \wedge \quad & |\{o | (m, o) \in E\}| \geqslant 2 \end{aligned}$$

Unlike the postdominator analysis, the dominator analysis is a classical forward problem using the same domain as the postdominator analysis. Since we are interested in an information being valid on all paths through a program, also the dominator analysis is a data-flow problem built under intersection. The resulting lattice is thus defined as

$$(D_{ctrl}, \sqsupseteq_{D_{ctrl}}, \sqcap_{D_{ctrl}}, \sqcup_{D_{ctrl}}, \top, \bot)$$

As we are only interested in dominators that are part of a given VHDL specification, and not in any dependencies introduced by the analysis framework, the update function $transfer_{dom,e} \colon D_{ctrl} \to D_{ctrl}$ for an edge $e = (m, n) \in E$ of a control-flow graph $G = (V, E, s, x)$ on a lattice element $l \in D_{ctrl}$ is defined as

$$transfer_{dom,e}(l) = \begin{cases} l \sqcup_{D_{ctrl}} \{m\} & \text{if } |\{o \mid (m, o) \in E_{\text{VHDL}}\}| \geqslant 2, \\ l & \text{otherwise.} \end{cases}$$

The return function $return_{dom} \colon D_{ctrl} \times D_{ctrl} \to D_{ctrl}$ used to combine results at the return of a function call in the interprocedural control-flow graph is defined as

$$return_{dom} = \sqcup_{D_{ctrl}}$$

Now, we can define the transfer function $\mathfrak{f}_{dom} \colon E \to (D_{ctrl} \to D_{ctrl})$ for any edge $e \in E$ as

$$\mathfrak{f}_{dom}(e) = transfer_{dom,e}$$

and obtain the data-flow problem $dfp_{dom} = (G, D_{ctrl}, \mathfrak{f}_{dom})$, whose maximal fixed-point solution $MFP_{dom}(n)$ for a node $n \in V$ yields a superset of possible control nodes. Combining the results of this analysis with the results of the postdominator analysis allows for computing the control dependencies of a given control-flow graph.

Combining the Results

As stated before, a node n is control dependent on a node m, if there exists at least one path in the control-flow graph starting at m to the exit x that does not contain n. Additionally, there must also exist at least one path from m to x that

contains n. The results from the postdominator and dominator analyses can now be combined in order to determine the set of nodes on which a node n is control dependent.

The nodes on which a specific node n of the control-flow graph is control dependent are computed by combining the data-flow values of these two analyses:

$$ctrl(n) = \begin{cases} \varnothing & \text{if } MFP_{dom}(n) = \top \\ & \quad \vee\ MFP_{dom}(n) = \bot, \\ \{m \in MFP_{dom}(n) \mid n \notin MFP_{pdom}(m)\} & \text{otherwise.} \end{cases}$$

Theorem 6.4.3 (Soundness of $ctrl$)
Let $G = (V, E, s, x)$ be a control-flow graph derived from the analysis framework. Let dfp_{pdom} and dfp_{dom} be the data-flow problems as described above. For all nodes $m \in V_{\text{VHDL}}$, the set $ctrl(m)$ is a sound approximation of the control dependencies and the following holds:

$$\forall m \in V_{\text{VHDL}} \colon \forall n \in \mathit{infl}(m) \Rightarrow m \in ctrl(n)$$

Proof 6.4.2
In order to prove this theorem, the following two propositions need to be shown:

1. dfp_{dom} and dfp_{pdom} are monotone data-flow problems $\implies$ MFP_{dom} and MFP_{pdom} are sound approximation of MOP_{dom} and MOP_{pdom}, respectively.

2. $\forall m \in V_{\text{VHDL}} \colon \forall n \in \mathit{infl}(m) \colon m \in ctrl(n)$

Let $G = (V, E, s, x)$ be a control-flow graph, and $dfp_{dom} = (G, D_{ctrl}, \mathfrak{f}_{dom})$ and $dfp_{pdom} = (G^{-1}, D_{ctrl}, \mathfrak{f}_{pdom})$ be the data-flow problems as described above.

to 1. Monotonicity of both data-flow problems, dfp_{dom} and dfp_{pdom}, follows directly from the construction of the update functions $transfer_{dom,e}$ and $transfer_{pdom,e}$. According to Theorem 2.4.1 on page 21, MFP_{dom} and MFP_{pdom} are sound approximations of MOP_{dom} and MOP_{pdom}, respectively.

to 2. Let $m \in V$.

$$\begin{aligned} & n \in \mathit{infl}(m) \\ \iff\ & m \downarrow n \\ \iff\ & \exists \pi \in P[m, n] \colon \forall u \in \pi \backslash \{m, n\} \colon n \in pdom(u) \end{aligned}$$

$$
\begin{aligned}
& \wedge \quad n \notin pdom(m) \\
\iff & \quad \exists \pi \in P[m,n] \colon \forall u \in \pi \backslash \{m,n\} : \forall \pi' \in P[u,x] \colon n \in \pi' \\
& \wedge \quad \exists \pi'' \in P[m,x] \colon n \notin \pi'' \\
\overset{*}{\iff} & \quad |\{z | (m,z) \in E\}| \geqslant 2 \\
& \wedge \quad \exists \pi'' \in P[m,x] \colon n \notin \pi'' \\
\implies & \quad m \in MFP_{dom}(n) \\
& \wedge \quad n \notin MFP_{pdom}(m) \\
\iff & \quad m \in ctrl(n)
\end{aligned}
$$

with $(*)$ basing on the conclusion that every path to the exit x starting at a node in between any path from m to n must contain n, but an alternative path $\pi'' \in P[m,x]$ with $n \notin \pi''$ exists. Thus, the number of outgoing edges of node m must be greater or equal to two.

□

6.4.4 Computing Slices

Interprocedural slices on VHDL for arbitrary criteria can be computed by means of reachability along control- and flow-dependence edges as described in [OO84]. Furthermore, the dependence introduced by sensitivity lists has to be gathered.

The previous two sections have described data-flow problems that yield sound approximations of a VHDL model's control- and flow-dependencies. Activation dependencies of processes introduced by the sensitivity list of the process needs not be recomputed since the analysis framework presented here already explicitly states them as the if-guard statement Ξ_p of the simulation process *controlling* execution of a process p. However, since these statements are explicitly bypassed while reconstructing the control-dependencies of a VHDL model (as framework induced parts shall not be part of a slice), the induced dependency needs to be covered for the computation of correct slices. Thus, we define the function $act \colon V_{\text{VHDL}} \to V_{simul}$ returning for a node $n \in V_{\text{VHDL}}$ the if-guard node that "controls" the execution of n.

The general slicing algorithm is shown in Listing 6.13 on the facing page. The input for the algorithm is the control-flow graph derived from the analysis framework. After the computation of the data-flow problems introduced in the Sections 6.4.2 and 6.4.3, slices can be computed arbitrarily often until abort is requested.

For computation, the algorithm holds two sets. One set is the working set ($wset$) containing tuples of nodes and identifiers that still have to be considered. The

```
Input: control−flow graph (V, E, s, x)
compute MFP_rd, MFP_pdom, and MFP_dom for (V, E, s, x)

while (no abort)
    wait for new slicing criterion C = (n, U)
    wset = {(n, u) | u ∈ U}
    vset = ∅

    while (wset ≠ ∅)
        let (m, w) ∈ wset
        vset = vset ∪ {(m, w)} ∪ {(c, _) | c ∈ ctrl(m)}
        tset = {m} ∪ ctrl(m) ∪ {act(m)}
        wset = wset\{(m, w)} ∪
                ( ⋃_{o∈tset, u∈use(o)} {(x, u) | x ∈ MFP_rd(o)(u)}\vset )

    slice = {m | (m, w) ∈ vset}
```

Listing 6.13 – The slicing algorithm.

visited set ($vset$) contains all tuples that have already been treated guaranteeing termination of the algorithm. A third set ($tset$) is employed as temporary storage. While the working set is not empty, the slice is not yet completely computed. In this case, an element (m, w) is selected from the working set and added to the visited set, together with all "control nodes" c on which the current node m is control-dependent.

All data and activation dependencies of the current node m and also the data dependencies of the control nodes are intersected with the complement of the visited set and then added to the working set. This guarantees the termination of the algorithm in $\mathcal{O}(|V| \times |identifier|)$ where $|V|$ and $|identifier|$ are the number of nodes and the number of identifiers used in the VHDL specification, respectively.

The slice for the specified criterion C can be calculated as the projection of the visited set to the nodes of the given control-flow graph.

6.4.5 Analysis of Functions and Procedures

The quality of the slicing algorithm presented in the last section strongly depends on the quality of the results of the flow- and control-dependency reconstruction, and therewith on the three presented data-flow problems. The term *quality of*

a slice aims at the number of statements, or to be more precise the number of nodes contained in it.

A slice S'_C with respect to a criterion C is said to be of a *better quality* than a slice S''_C for the same criterion, iff $|S'_C| < |S''_C|$.

Computing slices on a hierarchical composed system – even after elaborations – still requires analysis of function and procedure calls. As stated earlier, VHDL distinguishes between functions and procedures: the first passes parameters by value, whereas the latter uses parameter passing by reference. Furthermore, functions return a value, whereas procedures do not.

In the following, we will describe an extension to the flow-dependency analysis that yields more precise results when analyzing functions and procedure calls. The optimization is build on the call/return node approach presented earlier. Every function (or procedure) call statement within a given processor specification is represented in the CRL2 description using a pair of call and return nodes connected to the routine representing the called function (or procedure). Access to the formal parameters, a function or procedure defines, is given over the function $formal\colon V_{\text{VHDL}} \to \mathcal{P}(identifier)$ returning for a given call node $c \in V_{\text{VHDL}} \wedge cat(c) = callnode$ the set of formal parameters of the called function.

Using this function, we can define an update function for call edges $transfer_{call}$: $D_{rd} \times E_{\text{VHDL}} \to D_{rd}$ as follows:

$$transfer_{call}(l,(c,s_i)) = \lambda j \in identifier.\begin{cases} must(l,j,n) & \text{if } j \in formal(c) \wedge (n,c) \in E_{\text{VHDL}}, \\ l(j) & \text{otherwise.} \end{cases}$$

Intuitively, each formal parameter of a function or procedure call is defined at the statement representing the VHDL call. Please note that elaboration ensures that all identifiers used in a VHDL specification have unique names, thus no additional effort to detect identifier overloading is required.

Additionally, functions as well as procedures may define *local variables* (cf. Section 4.5.2 on page 66) that are only accessible from within the function or procedure. Thus, information being computed for these local variables is no longer needed after the return from the call. Let $local\colon V_{\text{VHDL}} \to \mathcal{P}(identifier)$ be a function that returns for a return node $n \in V_{\text{VHDL}}$ the set of local identifiers defined by a function (or procedure). If n does not belong to a function or procedure, the empty set is returned. The return function $return\colon D_{rd} \times D_{rd} \to D_{rd}$ combining the data-flow values coming from the local edge and the return edge at a return

node $r \in V_{\text{VHDL}}$ is extended to:

$$return(l_{call_i}, l_{x_i}) =$$
$$\lambda j \in identifier. \begin{cases} \bot & \text{if } j \in local(x_i), \\ l_{x_i}(j) & \text{if } is\_procedure(x_i) = true, \\ may(l_{call_i}, j, x_i) & \text{if } \exists (n,c) \in E_{\text{VHDL}} : \exists e = (c,r) \in E_{\text{VHDL}} : \\ & \quad cat(e) = localedge \wedge j \in def(n) \\ & \quad \wedge\ composite(type(j)) = true, \\ must(l_{call_i}, j, x_i) & \text{if } \exists (n,c) \in E_{\text{VHDL}} : \exists e = (c,r) \in E_{\text{VHDL}} : \\ & \quad cat(e) = localedge \wedge j \in def(n), \\ l_{call_i}(j) & \text{otherwise.} \end{cases}$$

Using this return function, information computed for local variables will be destroyed at the return node. Parameter passing by reference is honored, as for procedure calls only the newly computed information on the return edge will be propagated. If the return value is used for assigning it to an identifier, also the dependency to the return statement will be established.

Use of these enhanced functions within the data-flow problem dfp_{rd} enables analysis of functions and procedures calls and results in a more precise reconstruction of flow-dependencies.

6.4.6 Timing-Dead Paths and Statements

This section introduces an extension of the flow-dependency analysis as well as the dominator and postdominator analyses in order to improve the quality of slices by incorporating knowledge from the assumption-based model refinement as described in Section 6.3 on page 126. Statements and consecutive parts of a VHDL specification that have already been marked as timing dead due to some assumptions provided by the user can also be safely discarded for backward slicing. Doing so, the resulting slices will be more precise.

In order to incorporate results from the assumption-based model refinement, the update functions presented in the last sections need to be extended to also consider timing-dead marks during analysis. In following, the enhancement to the transfer functions of the flow-dependency analysis will be described first. Afterwards, also the enhancements to the dominator as well as the postdominator analysis will be given.

The data-flow problem dfp_{rd} describes a problem, where information is computed in forward direction of a given control-flow graph. Since the result of the analysis must be valid only on one path through the graph, the union of

incoming information is formed at control-flow joins. Thus, the neutral data-flow value not destroying information at joins is the least element $\bot$ of the function lattice D_{rd}. The transfer function $transfer_{\text{VHDL}} \colon D_{rd} \times E_{\text{VHDL}} \to D_{rd}$ for the flow-dependency analysis therefore is defined for edges $e = (m, n) \in E_{\text{VHDL}}$ as:

$$transfer_{\text{VHDL}}(l, (m, n)) =$$

$$\lambda i \in identifier. \begin{cases} \bot & \text{if } timing\_dead((m,n)) = true, \\ l(i) & \text{if } timing\_dead(m) = true, \\ may(l, i, m) & \text{if } composite(type(i)) = true \\ & \quad \wedge\ i \in def(m), \\ must(l, i, m) & \text{if } i \in def(m), \\ l(i) & \text{otherwise.} \end{cases}$$

This function can now be used within the global transfer function in order to enhance the results of the flow-dependency analysis.

The idea of propagating neutral elements on paths can also be applied to the dominator and postdominator analyses. In contrast to the flow-dependency analysis, information computed by these two analyses shall be valid on all paths, thus, the intersection of incoming information is formed at control-flow joins. Thus the transfer functions $transfer_{dom} \colon D_{ctrl} \times E \to D_{ctrl}$ and $transfer_{pdom} \colon D_{ctrl} \times E \to D_{ctrl}$ for the dominator and postdominator analyses, respectively, are defined for an edge $e = (m, n) \in E$ as:

$$transfer_{dom,e}(l, (m, n)) = \begin{cases} \top & \text{if } timing\_dead((m,n)) = true, \\ l & \text{if } timing\_dead(m) = true, \\ l \sqcup_{D_{ctrl}} \{m\} & \text{if } |\{o \mid (m, o) \in E_{\text{VHDL}}\}| \geqslant 2, \\ l & \text{otherwise.} \end{cases}$$

$$transfer_{pdom,e}(l, (m, n)) = \begin{cases} \top & \text{if } timing\_dead((m,n)) = true, \\ l & \text{if } timing\_dead(m) = true, \\ l \sqcup_{D_{ctrl}} \{m\} & \text{if } m \in V_{\text{VHDL}}, \\ l & \text{otherwise.} \end{cases}$$

Building the intersection at control-flow joins ensures in both analyses that no information gets lost.

Using the enhanced transfer functions results in an improved prediction (or reconstruction) of a program's flow and control dependencies. As the determination of slices depends on these reconstructed dependencies, also the quality of the resulting slices is improved.

7

Implementation and Evaluation

> Thinking is easy, acting is difficult, and to put one's thoughts into action is the most difficult thing in the world.
>
> *(Johann Wolfgang von Goethe)*

The static analysis framework described in Section 6.1 is intended for but not limited to use with *PAG*. However, a specification needs to be translated into the intermediate language CRL2.

In order to show the applicability and the effectiveness of the framework and the analyses described so far, a compiler transforming a VHDL specification into a CRL2 description including elaboration, etc. has been developed. Furthermore, reset analysis, assumption-based model refinement, and also backward slicing using the abstract semantics as defined in Sections 6.2, 6.3, and 6.4, respectively, have been implemented. Their implementations and usability, the soundness of the analyses, but also some general applicability rules are subject of this chapter. Additionally, some general code of work patterns supporting the process of timing model generation are given.

This chapter is organized as follows: Section 7.1 describes some processor specifications that have inspired parts of this thesis, but also have been used for evaluating the work presented in this thesis. Section 7.2 describes the implementation of the compiler transforming a VHDL specification into a CRL2 description. Also the implementations of the analyses are detailed. Section 7.3 presents the experimental evaluation, soundness and applicability considerations.

7.1 Hardware Models

This section provides an overview on the VHDL models used for evaluation. Moreover, the insights of these models have directly influenced the genesis of the timing model derivation methodology and the static analysis framework.

In general, hardware models are hard to obtain. Besides some industrial open-source implementations of cores like the LEON family, there exist a wide range of research implementations of the DLX machine.

Developing a processor or even a part of it (e.g., a memory controller) is a quite challenging task requiring large development teams with many years of experience. Thus, it is not amazing that these specifications are part of the intellectual property of the manufacturers, and access to them is strongly regulated, and in most cases strictly prohibited. However, access to some commercial specification is granted under confidentially within some of the projects mentioned at the beginning of this thesis.

In the following, the characteristics of the models used for evaluation are given. To respect the intellectual property of others, abstract names for the confidential models are used.

7.1.1 Gaisler Research LEON2

LEON2 [Gai05] is a 32-bit CPU microprocessor core based on the SPARC-V8 RISC architecture and instruction set [SPA91]. The core arose from the LEON CPU family that was originally designed by the European Space Research and Technology Centre (ESTEC), the largest site of the European Space Agency (ESA). Today, development is performed at site of Gaisler Research [Gai]. The core is used in system-on-a-chip designs both in research and commercial settings.

LEON2 has a five-stage pipeline [Gai05] while later versions (LEON3 and LEON4) have a seven-stage pipeline [Gai08]. It is designed for embedded applications and provides separate instruction and data caches and hardware multiplier and divider support. Additionally to the core itself, LEON2 also provides the following features:

- Interrupt controller,
- Debug support unit with trace buffer,
- Two 24-bit timers,
- 16-bit I/O port, and
- Memory controller.

The LEON2 core is described in fully synthesizable VHDL and can be implemented on both, FPGAs and ASICs. The specification of the core offers the possibility to configure several components through VHDL generics before hardware synthesis. Configuration management thereby includes the possibility of changing the cache size of the instruction and data caches (i.e. 1 – 4 sets, 1 – 64 kilobyte/set, 16 – 32 bytes per line). Also the replacement policy employed after synthesis can be configured through these generics. LEON2 offers all basic functions of a pipelined in-order processor. The VHDL description of the LEON2 core consists of 69144 lines of code, 210 entities and 61 packages. The specification can be viewed as large in both, size and complexity.

7.1.2 Superscalar DLX Processor

In Section 4.1.2, the simple DLX machine [HPG03, MP00] has been introduced. Various implementations of this machine – pipelined and non-pipelined – exist. The implementation provided by TU Darmstadt [Hor97] is one of the most challenging ones. The design of the *superscalar DLX* is based on the design of Freescale's PowerPC MPC603e processor family [Fre02].

Figure 7.1 on the next page shows the block diagram of the superscalar DLX machine being able to issue and retire a maximum of two instructions per clock cycle. The four independent execution units (branch-resolve unit, arithmetic-logic unit, multiply-divide unit, and load-store unit) have a reservation station each in order to maintain a coherent system state [Tom67].

Execution of instructions can be out-of-order, and a reorder buffer is used to commit instructions in program order. The latter also enables precise exception processing. Additionally, the core features a branch-target buffer supporting the instruction prefetch of up to two instructions. In order to increase the performance, the superscalar DLX provides two separate instruction and data caches, each with a width of 64 bytes, and 4-entry instruction and data address-translation buffers. Access to the main memory is controlled via a centralized bus interface unit.

Despite of all these features, implementation is given in only one entity and one support package within 5022 lines of code in total. The implementation provided by TU Darmstadt [TUD] is not described in fully synthesizable VHDL, but compliance to the synthesizable substandard [IEE99] can be achieved easily. The VHDL specification of the superscalar DLX is small in terms of size and complexity.

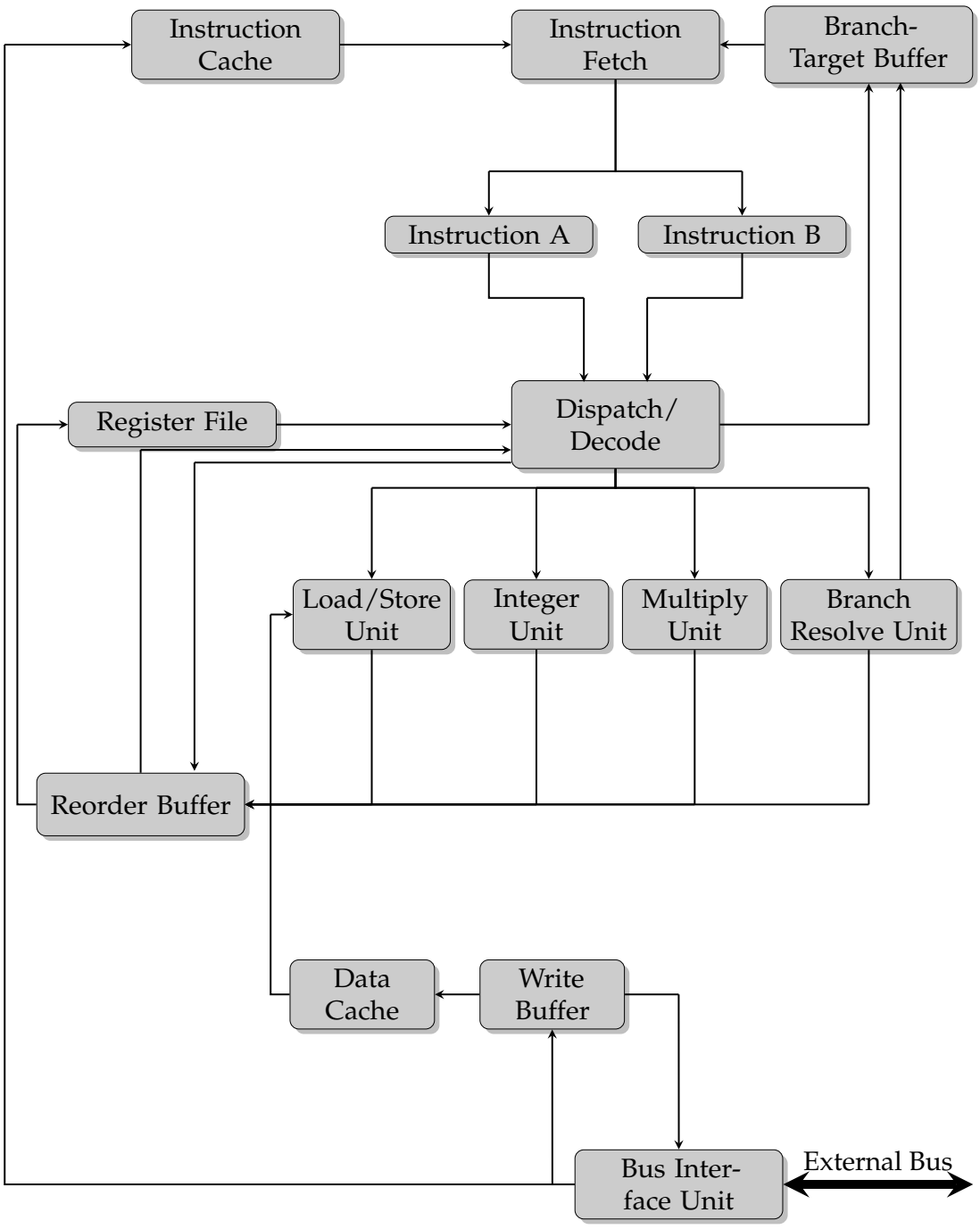


Figure 7.1 – Block diagram of the superscalar DLX machine.

7.1.3 Confidential Specifications

Beside the open-source processor descriptions described above, the collaboration within national and European research projects has offered limited access to commercial processor specifications from the automotive as well as the avionics field of application. These models have also been used for evaluation the methods presented within this thesis and are thus briefly described here. Due to confidentiality, pseudonyms are used to distinguish the different cores.

Automotive CPU 1 is a 32-bit peripheral co-processor used in the automotive area. It is fully user-programmable and offers efficient support for DMA and bus transactions. Automotive CPU 1 is designed as a Harvard architecture, i.e. it offers separate code and data memory spaces. The specification is given in form of fully synthesizable VHDL in around 31000 lines of code providing 49 entities and 10 packages.

The second processor whose specification is closed source and which was only accessible within a project is called *Automotive CPU 2*. This processor is widely in use within the automotive area, but is also suitable for use within the process measuring and control technology. Automotive CPU 2 is a superscalar multi-stage pipelined CPU with a 32-bit load/store Harvard architecture with several parallel execution path for different instruction classes. It further comprises a deep memory hierarchy with several communication buses and supports both, 16-bit and 32-bit instruction formats. The VHDL specification is fully synthesizable and consists of more than 160000 lines of code, 284 different design entities and 13 packages. Unfortunately, access to the whole specification of Automotive CPU 2 was provided only for a limited period. Thus, only the memory subsystem could have been used for evaluation. These parts consist of around 8400 lines of code, divided into 9 entities and 6 packages.

Besides the two automotive cores, also a memory controller being used in the avionics industry has been used for evaluation. Since this memory controller is confidential, it will be called *Avionics MCU* for the remainder of this thesis. Avionics MCU supports static as well as dynamic RAM and additionally allows for connecting external periphery via the peripheral component interconnection bus. The specification comprises of around 20000 lines of synthesizable VHDL code, divided in 32 entities and 4 packages.

7.1.4 Model Review

The previous sections have shortly introduced several processor and component specifications being subject to the evaluation of this thesis. An overview on the size of the used models in terms of lines of code, empty lines, and comment lines

VHDL model	Blank lines	Comment lines	VHDL lines	Total
LEON2	5793	7253	56098	69144
Superscalar DLX	718	1108	3196	5022
Automotive CPU 1	4058	5637	22286	31981
Automotive CPU 2	20719	27326	116431	164476
Avionics MCU	3783	3500	11703	18986

Table 7.1 – VHDL model characteristics – Lines of code.

VHDL model	Entities	Packages	Total Units
LEON2	210	61	271
Superscalar DLX	1	1	2
Automotive CPU 1	49	10	59
Automotive CPU 2	284	13	297
Avionics MCU	32	4	36

Table 7.2 – VHDL model characteristics – Design unit overview.

is given in Table 7.1. By only considering the lines of code, the superscalar implementation of the DLX processor is the smallest implementation that has been used, whereas Automotive CPU 2 is the largest one. As access to Automotive CPU 2 was restricted to a limited time period, Automotive CPU 1 is the largest processor specification. Even though Avionics MCU is only the implementation of a dynamic memory controller, i.e. the specification describes an open design, Avionics MCU is half of the size of Automotive CPU 1 in terms of lines of code.

The complexity of a specification given in VHDL is not revealed by only considering the amount of lines of code. Due to the hierarchical design methodology behind VHDL (and other hardware description languages) to support a distributed, but also structured development, the complexity of a specification also depends on the number of component instantiations. In general, the more different design units a specification consists of, the more complex the resulting model is. Obviously, the number of lines of code and the number of design units are in general not independent, but there exists no strict connection. Common practice shows that implementations become unreadable, incomprehensible, and unmaintainable, if a design unit description becomes too large.

Table 7.2 gives an overview over the design units of the hardware models

used for evaluation. Unsurprisingly, the implementation of the superscalar DLX machine is the most simple one even in terms of number of design units, whereas the specification of LEON2 that is of average size in terms of lines of code tends to be complex in terms of number of design units.

It is a surprising finding that publicaly available models often use more complicated VHDL constructs, whereas commercial models of processors that are widely in use employ a much more restricted subset. Furthermore, commercial models seem to be closer to the real hardware, i.e. their specifications are closer to a model using explicit registers and transactions between them, whereas publicaly available models use more abstract data types.

This is mainly based on the fact that open-source models often employ configurability by using generics allowing modification of replacement policies of caches, queue size, etc. to address a wide range of application and allow for assembly on FPGAs and ASICs, whereas within commercial processor specifications, use of VHDL constructs is limited to a set of standard idioms that have been intensely tested and are known to be synthesizable without further adjustments.

7.2 Implementation

This section describes the implementation of key components of the static analysis framework introduced in Section 6.1. Furthermore, implementational details showing the flexibility and reliability of the framework as well as the simplicity of analyses specifications are given.

7.2.1 VHDL Analysis-Support Library

Most analyses presented in this thesis but also the analysis framework itself are defined using the function *eval* for evaluation of VHDL expressions. In order to ease the development of the analyses, but also to ease the process of elaboration, a VHDL analysis-support library has been developed with the goal of providing an easy-to-use interface for evaluating VHDL expressions. To overcome the need of parsing a VHDL expression with its complex operator precedence rules, its potentially deep nesting, and its ambiguity in the used syntax[1], expressions to be evaluated has to be given in a kind of *prefix notation*. The prefix notation of mathematical expression was invented by Łukasiewicz [LT30] and is also known as Polish notation.

[1] In VHDL, it is impossible to distinguish an indexed access to an array element from a function call with one parameter by syntactical means. For this, semantic information is of urgent need.

Whereas the prefix notation by Łukasiewicz for the expression

$$(3 - 5) * 9$$

would be

$$* (- 3\ 5)\ 9,$$

the variety of basic VHDL functors requires unary, binary, and ternary operators[2], thus brackets and comma for expressing the affiliation of arguments are used explicitly. This enhances readability, but also eases parsing. The above example will be expressed as

$$*(-(3, 5), 9).$$

The library offers support for most basic types covered in the synthesizable VHDL standard [IEE99]. Additionally, some basic types of widely used VHDL libraries (e.g., IEEE standard logic and standard arithmetic) are supported. Thus, integer, real, bit, bitstring, std_ulogic, std_logic, and strings are natively supported. The current implementation does not offer support for array data types, and along with that, array slices and indexed accesses cannot be evaluated.

Beside this, support for interval arithmetic is also included following the rules presented in [MKC66]. Intervals are supported for all basic types mentioned above except strings due to the lack of a proper interval arithmetic for string intervals. In order to express indecisiveness due to a lack of concrete values or the ambiguity of a result (e.g., $[0, 1] \geqslant 0$), the library additionally supports a "don't know" (dunno) element, i.e. $\top$.

Most expression to be evaluated are not of the kind as in the example above – simple expressions as above can be directly evaluated and their results can be inlined in order to simplify the VHDL specification. Normally, expressions contain at least one signal or variable, whose value depends on the current execution history. In order to evaluate these expressions, the current environment, i.e. the mapping from identifiers to current values as described in Section 6.1.4 on page 106 is used for evaluation.

Obviously, the VHDL analysis-support library does not offer support for function calls being part of the expression to be evaluated. As evaluating of function calls would require an evaluation of arbitrary VHDL code, this drawback can easily be softened by requiring the *administrative normal form* (ANF) [FSDF93] for all expressions. Calls to functions within expressions can be factored out, their results can be assigned to temporary variables that have to be used within the original expression instead of their calls. By this, function calls can be handled by the analyzer that uses the support library and expressions can be evaluated by use of the environment provided for evaluation.

[2] In many programming languages, `?` is a ternary operator: `e1 ? e2 : e3`.

The library as described has been implemented in C++ and offers an easy-to-use interface that can be accessed from any analyzer. Basically, it offers two functions for any expression $expr \in Expression$ given in prefix notation

$$eval\colon Expression \to V_{Int}, \text{ and}$$
$$eval\colon Expression \times D_{Int} \to V_{Int},$$

with V_{Int} and D_{Int} as defined in Section 6.3 on page 126, where the first function uses the predefined environment $\lambda i \in identifier.\top$ for evaluation.

7.2.2 VHDL Compiler

In general, the VHDL specification of a processor (or a component) consists of several modules, the so-called design units. Usually, these units are separated into several files. Before a VHDL model can be elaborated, each design unit has to be "compiled" and added to the *design library*. Compilation into the design library ensures syntactical and semantical correctness and also guarantees that dependencies to other modules can be resolved. In order to resolve dependencies to other design units, all *signatures* of units already contained in the design library have to be preloaded before parsing a new module. The signature of a design unit describes the interface of the unit that is visible from the outside. It does not care about implementational details, and compared to other programming languages, it is similar to header files in C++. Each design unit has its own namespace, but use-clauses can be used to remove the namespace name as an explicit qualifier when parsing a new module.

In order to get a CRL2 description of a processor from its VHDL specification, a VHDL compiler called VHDL2CRL2 has been developed. It is used for building up the design library of the processor specification and performs all the required renaming for unifying names, instantiates referenced components, and wires structural descriptions as described in Section 4.5.3. Figure 7.2 on the following page gives a schematic overview of the structure of the compiler. VHDL2CRL2 is organized into three main phases:

1. Model parsing,
2. Model elaboration, and
3. CRL2 generation.

In order to derive an elaborated, i.e. flattened, model of a processor, every design unit has to be compiled into the design library. Parsing of VHDL offers several characteristic problems:

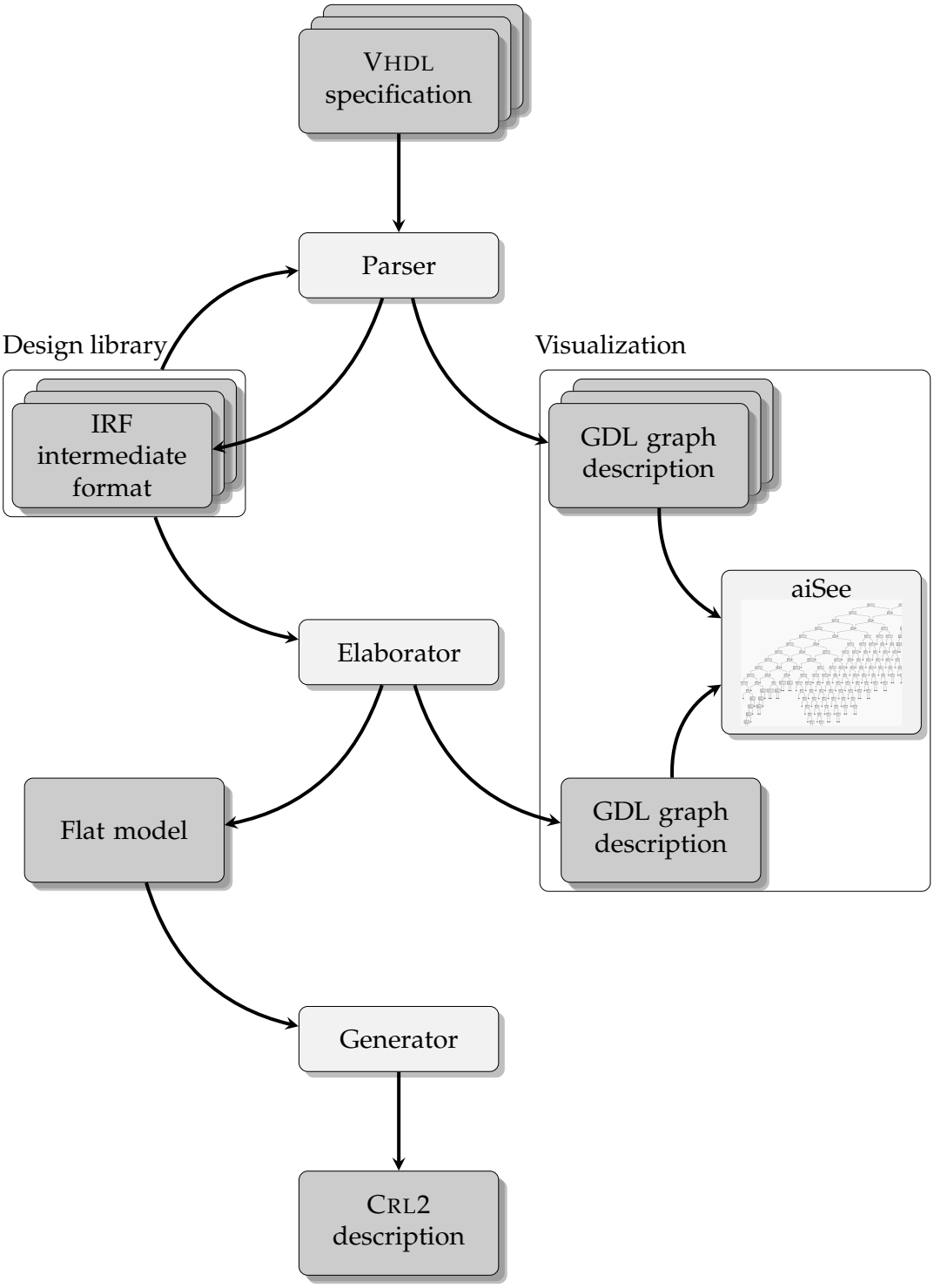


Figure 7.2 – VHDL compiler – Overview.

- The language design of VHDL makes it impossible to distinguish indexed accesses to arrays from calls to functions with only one parameter by syntactical means. In the syntax of VHDL, both are written as `a(i)` requiring semantical knowledge to distinguish them.
- Name scoping rules of VHDL allows for accessing arbitrary identifiers from disjoint blocks by using appropriate prefixed names. Thus, parsing requires a powerful symbol table, as usual scoping rules known from wide-spread programming languages are not sufficient for VHDL.

During parsing, several semantic preserving transformations are automatically applied to ease subsequent phases and later static analyses. As described in Section 4.5.4 on page 69, the implementation of a module is given in form of one or more processes that all run in parallel. Instead of specifying every process with its sensitivity list, VHDL offers two shortcut versions: concurrent signal assignments and concurrent procedure call statements. Their semantic is equivalent to a process with all used signal identifiers as part of the sensitivity list, and only one statement in the process' body, namely the signal assignment or the procedure call, respectively. VHDL2CRL2 syntactically changes these kind of statements in order to ease the generation of CRL2. Furthermore, switch statements contained in the specification are transformed to if-then-else cascades. Doing this eases static analyses as the condition for each decision is explicitly visible. Both transformations that are automatically applied to a given design unit specification are only of a syntactical kind and do not affect the semantics of the design unit.

Every parsed design unit is added to the design library – a centralized storage for all items belonging to a specification. Internally, the design library is organized into several libraries. All items of a dedicated library must be loaded before parsing a new design unit. In order to ease this, we have developed an internal format, called *IRF*, containing all information necessary to rebuild the annotated syntax tree and to update the internal symbol table. A small excerpt of an IRF file is shown in Listing 7.1 on the following page.

After having parsed all design units belonging to a specification, elaboration has to be performed in order to flatten the design hierarchy and to wire all instantiated components. Elaborating a VHDL model is a complex process comprised of several steps. In order to ease later static analyses, the elaboration process described in the VHDL language standard [IEE87] is supplemented by some steps that do not change the semantics nor the timing of a VHDL specification. These additional steps are marked as not required by the standard. In the following, the steps for elaborating a model are detailed:

1. Elaboration starts with a *transformation of subtypes* mapping all identifiers of a subtype back to the VHDL standard type, the subtype is derived from. Subtypes enhance the readability of a specification, but after parsing, the

```
⋮
<NODE>
# Common node properties inherited from ParseNode
ID       ="21262";
TYPE     ="NODE_SUBTYPEINDICATION";
LINENO   ="17";
FILENAME ="sources/Dlx.vhd";
MNEMONIC ="";

# Common node properties inherited from DeclarationNode
NAME="ieee.numeric_bit.unsigned";
TYPENAME="ieee.numeric_bit.unsigned";

# Specific properties
RESOLUTION_NAME="";

# Definition of the left child
<LCHILD>
<NODE>
# Common node properties inherited from ParseNode
ID       ="21261";
TYPE     ="NODE_INDEXCONSTRAINT";
LINENO   ="17";
FILENAME ="sources/Dlx.vhd";
MNEMONIC ="";

# Specific properties

# Definition of the left child
<LCHILD>
<NODE>
# Common node properties inherited from ParseNode
ID       ="21260";
TYPE     ="NODE_DISCRETERANGE";
LINENO   ="17";
FILENAME ="sources/Dlx.vhd";
MNEMONIC ="DIRECTION_DOWNTO";

# Common node properties inherited from ConstraintNode
DIRECTION_TYPE="DIRECTION_DOWNTO";

# Specific Properties
⋮
```

Listing 7.1 – Excerpt of an IRF file.

```
for i in x'RANGE loop
    < loop body >
end loop;

                ⇓

if x'ASCENDING then
    for i in x'LEFT to x'RIGHT loop
        < loop body >
    end loop;
else
    for i in x'LEFT downto x'RIGHT loop
        < loop body >
    end loop;
end if;
```

Listing 7.2 – Transformation of *range* attributes.

restricted co-domains are no longer needed. To support static analyses, information on restricted co-domains is stored at the annotated syntax tree, but the mapping to standard types enables use of the analysis-support library.

2. The process of elaboration proceeds with *resolving of overloaded functions*. VHDL supports function overloading, but in order to flatten a design hierarchy, calls to overloaded functions need to be cleared.

3. Functions are allowed to return composite data-types, but the elaboration process needs to break record structures to standard types. Thus, functions returning record data-types need to be transformed into procedures of equivalent semantics by introducing a new temporary variable of the same type as the return value of the function. *Transformation of record-returning functions* into procedures also enables applicability of the next step.

4. *Collapsing of record data-structures* has to be performed to unfold (possibly nested) composite record structures and to map them back to basic VHDL types. Therefore, assignments to records need to be duplicated to assign each record element. This step is rather complex and increases the size of the model due to inserting new identifiers and duplicating assignments.

5. *Removal of range attributes* forms the next step in elaboration. Whereas the previous steps have all been required by the standard, this step is only required for CRL2 generation. The *range* attribute in VHDL is defined for arrays and is independent from the concrete array definition. The attribute is usually used within for-loops and allows for writing code independently

from the iteration direction. Listing 7.2 on the previous page depicts the transformation of these loop statements into semantically equivalent loops. Depending on the direction of the array type that can be checked via the $ascending$ attribute, different loops are executed easing a later loop transformation to enhance analyzability. If the direction of the loop is statically known, the then or else branch can be omitted, but sometimes (e.g., within functions), both outcomes are possible.

6. *Unification of type names* is performed to enhance the analyzability of the resulting flattened VHDL model. For example, the IEEE support library defines the type $unsigned$ within several packages, namely in the numeric_bit package and the std_logic_arith package. If both packages are used within different design units (within one design unit, parsing will fail with a type-checking error), elaboration will see an overloaded type. In order to solve this, the unique prefix of a type is prepended at each declaration.

7. Expressions in VHDL can be arbitrarily nested including function calls. In order to enhance the analyzability, each statement is *converted to the administrative normal form* [FSDF93]. This transformation eases CRL2 generation and also the specification of data-flow problems. Furthermore, this step enables use of the analysis-support library.

8. Many components of a VHDL specification can be implemented as iterative compositions of smaller components. Memories being composed of a rectangular array of storage cells are a classical example of this. In order to ease their implementation, VHDL offers the possibility of expressing the repetition of subsystems by only describing the subsystem once and generating the remaining ones. But for elaboration, these *generate statements* need to be *unrolled* and the subsystems need to be instantiated.

9. In order to obtain a flat model (i.e. a model without hierarchy), *component instantiations* need to be *embedded* into the topmost design hierarchy. This can be easily implemented by replacing every component instantiation statement by its corresponding entity implementation.

The last two steps (8 and 9) have to be applied iteratively until all component instantiations have been embedded into the topmost design hierarchy element. Elaboration is performed using the topmost design entity, which is usually the description of a whole processor. Every subsystem used by this unit is embedded (i.e. the implementation is inlined instead of using the port mapping), which may result in new component instantiations within the newly inlined implementations. Thus, elaboration flattens the design hierarchy by transitively inlining all instantiated components into the topmost design hierarchy element.

10. Besides variable and signal declaration, VHDL also allows to declare and define constant values. To enhance the analyzability of the model, *con-*

stants referenced in the specification are *inlined* and their declarations are removed.

11. Blocks in VHDL are used to introduce a new naming scope and can be compared to braces in C++. Embedding of component instantiations as described above is implemented using these blocks, but for elaboration, signal and variable declarations need to be moved to their correct position at the topmost design hierarchy element.

12. Elaboration ends with the *removal of* remaining *entity declarations and implementations* as they were orphaned after having been embedded into the topmost hierarchy element.

The result of the elaboration process is a flat version of the VHDL specification that can be either used for simulation or hardware synthesis.

After elaboration, the flattened model is passed to the CRL2 generator applying the mapping rules described in Section 6.1.3 on page 104 including the transformation of for-loops to while-loops. In a second step, while-loops are transformed into recursive routines enhancing their analyzability (cf. [MAWF98]). Each statement is annotated with attributes that ease analysis specification. The attributes being added classify each statement according to the statement's type, encode information on defined and used identifiers, and also encode the prefix notation of expressions. Adding the prefix notation of expressions enables use of the VHDL analysis-support library (cf. Section 7.2.1) by static analyzers. Furthermore, attributes encoding the actual and formal parameters of functions and procedures depending on the calling context are added.

The CRL2 generator adds the analysis framework simulation routines used to model the second semantical level of VHDL to the model. During simulation routine generation, a fixed update ordering of VHDL processes is chosen. Since *PAG* currently requires a non-recursive routine to be the entry for generated analyzers, the generator additionally adds an analysis start routine simply calling the generated recursive clock process. Each of these supporting routines is additionally annotated with attributes marking them as support routines and enabling a later classification of control-flow graph nodes into different (and disjoint) sets of nodes as described in Section 6.2 on page 113.

Passing a flattened (and elaborated) VHDL model to the CRL2 generator yields a CRL2 description fulfilling the requirements of the analysis framework (cf. Section 6.1) enabling use of *PAG* to generate efficient static analyzers from high-level specifications.

In order to ease the development and to support the debugging of the VHDL compiler, each stage in the compilation process is fully visualizable. For this, the *graph description language* (GDL) [Abs05] is used and each stage in the VHDL

```
SET
    strl = list (str)

DOMAIN
    func = str -> Value
    map = lift (func)
    env = map * map * map
```

Listing 7.3 – Lattice specification of the reset analysis.

compiler is extended by GDL-export facilities. The resulting graphs can be interactively explored using AbsInt Angewandte Informatik's graph visualization software *aiSee*.

Although the projects, in which this work was supported, require a closed source development also of the VHDL compiler, there exists an open-source implementation of a VHDL compiler and simulator, called GHDL [Gin07]. In contrast to VHDL2CRL2, GHDL requires the GCC toolchain. The model is compiled resulting in an executable simulator that can be further inspected using gtkwave [Byb12]. In order to obtain a CRL2 description of a VHDL specification, at least the code generation backend of GHDL had to be adjusted.

7.2.3 Reset Analyzer

Chapter 6 has introduced an abstract semantics allowing static analyses of hardware descriptions given in VHDL. As stated before, *PAG* is used to generate program analyzers from concise high-level specifications. The reset analysis presented in Section 6.2 on page 113 has been implemented using *PAG*.

An analysis specification to be used with *PAG* consists of two parts: a domain specification part and a problem and transfer specification part. In the following, the specifications to implement the reset analysis are detailed.

Listing 7.3 depicts the DATLA definitions of data types and lattices that are required for analysis specification. The specification of data types is given after the `SET` keyword, whereas lattices that can be used as basis of data-flow problems have to be defined after the `DOMAIN` keyword. Building on the predefined data type `str` representing strings, a list of strings called `strl` is defined. The lattice D_{cp} used within the data-flow problem specification of the reset analysis is constructed in the `DOMAIN` section. First, a function domain `func` mapping identifiers (represented via strings) to values is constructed. The type `Value` used within the specification refers to the data type provided by the VHDL analysis-support library representing the interval domain V_{Int} (cf. Section 6.3). This

```
PROBLEM ResetAnalysis
    direction  : forward
    carrier    : env
    init       : (bot, bot, bot)
    init_start : (top, top, top)
    combine    : comb
    widening   : wid
    equal      : eq
```

Listing 7.4 – Problem specification of the reset analysis.

function domain is extended by additional bottom and top elements resulting in the domain D_{cp}. The final domain `env` used for analysis is then constructed as the Cartesian product (d_1, d_2, d_3) of this domain modeling past, present and future values of signals and variables, respectively. Thus, `env` is an implementation of the environment Θ (cf. Definition 6.1.1) easing the implementation of the synchronization statement Ω.

The specification of the lattice can then be used within the specification of the data-flow problem. A data-flow specification within *PAG* is split up into three sections: the problem specification, the transfer-function specification, and the specification of support functions.

Listing 7.4 gives the problem specification of the reset analysis. This part of a *PAG* specification is introduced after the keyword `PROBLEM` and covers the direction of the data-flow problem (forward or backward), the lattice used for analysis, and also the initial value to be assigned to the nodes of the control-flow graph to be analyzed. The initial value to be used at analysis start is given after the `init_start` keyword. Obviously, the value provided matches the definition of the initial element ι as described in Section 6.2. Furthermore, functors used to combine data-flow values at control-flow joins, a widening operator and also an equality functor used for verifying the achievement of the fixed point are required. The implementation of the functors used in the problem specification has to be provided within the support section.

The specification of the transfer functions of a data-flow problem has to be provided in the transfer section of a *PAG* specification. *PAG* offers support to provide transfer functions for edges as well as for nodes. Transfer functions that shall be applied to edges have to be introduced following the keyword `TRANSFER_EDGE`. The applicability of a transfer function to an edge can be restricted by requiring a dedicated source node, a dedicated edge type and a dedicated destination node. Listing 7.5 on the following page gives an excerpt of the transfer-function specification of the reset analyzer. For readability reasons, only the transfer function to be applied to true-edges is given; the transfer

```
TRANSFER_EDGE
normal_node (operation::[!]), edge_true, _:
  let
    cond = eval (vhdl_rhand_side (operation), @!1!3, @!2!3);
  in
    if (cond = top || cond != Value(0)) then
      //condition could not be evaluated or evaluates to true
      @
    else
      bot
    endif;
⋮
TRANSFER_STATEMENT
normal_node(operation::[!]):
  if (vhdl_synchronize (operation)) then
    //sync point: make all scheduled signals visible at once
    (@!2!3, @!3!3, @!3!3)
  else if (vhdl_environment (operation)) then
    //ensure reset signal is active
    (update (rst_signal (), @!1!3, rst_value ()),
     update (rst_signal (), @!2!3, rst_value ()),
     update (rst_signal (), @!3!3, rst_value ()))
  else let
      def = vhdl_definition (operation);
    in
      if (def = "") then
        //no definition here, conditions are evaluated at edge
        @
      else let
          result = eval (vhdl_rhand_side  (operation), @!2!3);
        in
          if (vhdl_variable_assignment (operation)) then
            (@!1!3,
             update (def, @!2!3, result),
             update (def, @!3!3, result))
          else
            (@!1!3,
             @!2!3,
             update (def, @!3!3, result))
          endif
      endif
  endif;
```

Listing 7.5 – Excerpt of the transfer-function specification of the reset analyzer.

function for false-edges can be implemented analogously. Depending on the result of the conditional expression of the source node when evaluated under the current environment, the actual data-flow value[3] or bottom is propagated to the next statement. Thus, also the update function $transfer_{if}$ as defined in Section 6.2 on page 113 can be easily implemented. Process reactivation due to signal changes at the if-guard statements Ξ can also be directly implemented by comparing the signals' past values (i.e. the first element of the Cartesian product domain) with the current ones (i.e. the second element in the Cartesian product domain). The corresponding code in the specification is omitted here for readability.

Transfer functions used for updating nodes of the control-flow graph have to be defined following the keyword `TRANSFER_STATEMENT`. A small excerpt of the specification of the reset analysis is also shown in Listing 7.5. The snippet depicts the specification of the synchronize statement Ω and also the handling of signal and variable assignments. If the current node in the control-flow graph is the synchronize statement introduced in the analysis framework, the current variable and signal mapping is moved to the first element of the Cartesian product domain in order to be able to check signals for events (i.e. the past value of a signal differs from the current one). Consequently, scheduled transaction are made visible by copying the third element of the Cartesian product domain to the second element denoting the current environment.

The update function for variable and signal assignments is also given in Listing 7.5 on the facing page from line 23 on. First, the set of identifiers being defined at the current node is obtained. If this set is empty, the current node does not define any identifier, so no further update is required, and the current data-flow value is propagated to the control-flow successor node. Otherwise, the right-hand side of the VHDL statement represented by the node is evaluated under the current data-flow value (i.e. the second element in the Cartesian product domain) using the VHDL analysis-support library. The result is then used to update the current and future value mapping in case of a variable assignment, whereas in case of a signal assignment, only the future value mapping is updated.

The lines 18 – 22 in Listing 7.5 implement the update function $transfer_{env}$ as defined in Section 6.2 on page 113. To access the user provided name of the reset signal and its activation value, the external functions rst_name and rst_value are used.

Functions that are used within the problem specification and the transfer function specification need to be given in the support section. This section is introduced by the keyword `SUPPORT` and allows for providing the definition of functions within the functional language FULA. Besides this, it is also possible

[3]Within FULA, the data-flow value entering a node is accessible via the special symbol `@`.

```
SUPPORT
update :: str, map, Value -> map;
update (def, env, value) = env\[def -> value];

//access to attributes
vhdl_synchronize :: CrlItem -> bool;
vhdl_environment :: CrlItem -> bool;
vhdl_variable_assignment :: CrlItem -> bool;
vhdl_definition :: CrlItem -> str;
vhdl_rhand_side :: CrlItem -> str;

//external interface to support library
eval :: str, map -> Value;
```

Listing 7.6 – Excerpt of the support function specification of the reset analyzer

to declare function prototypes that need to be given externally in C/C++. Listing 7.6 depicts an excerpt of the support function section of the reset analysis specification. Definitions of combine, widening and equality checking are omitted here; their implementation is straight forward to their definitions given in Section 6.2 on page 113. In addition to access functions returning the existence of several attributes as mentioned before, also the external interface to the VHDL analysis-support library is given here. The function $update$ is used for updating the current value mapping of the identifier to a new value. Its definition is given in the functional specification language FULA.

Using this specification, *PAG* generates the source code of a static data-flow analysis implementing the reset analysis. The source files can be compiled into a library that can be easily used from within own projects. The analysis is accessible via a dedicated interface function that computes the maximal fixed-point solution of the data-flow problem dfp_{cp} for a given CRL2 description of a VHDL model. Afterwards, the resulting data-flow values can be accessed from within a project. By this means, also the set of stable signals S_{stable}, but also the set of possible clock domains S_{pclk} as defined in Section 6.2 on page 113 can be easily computed. Both sets are added to the CRL2 description for later usage by other analyses.

Computation of the sets of stable identifiers and possible clock domains have been implemented, and, paired with the analysis library, form the tool *reset analyzer*. The input for the analyzer is the CRL2 description of a VHDL model, the name of the reset signal as well as its activation value[4], and the name of the

[4]Some designers prefer active-low signal values, other prefer active-high values. An *active-high* signal represents the asserted state by the higher of two voltages, whereas an *active-low* signal represents the asserted state by the lower of two voltages.

external clock signal. Using these informations, the analyzer computes the sets of stable signals under reset and possible clock domains, and adds the resulting sets to the CRL2 description for later usage.

7.2.4 Assumption-based Model Refiner

As the reset analysis, the assumption evaluation analysis (cf. Section 6.3 on page 126) is implemented using *PAG*. The specification of the analysis is rather similar to the specification of the reset analyzer given in the previous section, and thus is omitted here. Interval arithmetics as required for the assumption evaluation analysis is already embedded in the data type provided by the VHDL analysis-support library, and thus, the domain specification for this analysis equals the domain specification of the reset analyzer. The analysis is integrated into a tool called *assumption-based model refiner* allowing the user to specify arbitrary assumptions of the kind "the value of a signal is guaranteed to be in the interval range a till b". Assumptions of this kind are supported for all data types with a properly defined interval arithmetic, such as bits, bitstrings, integers, reals, and enumerations. Due to the lack of an interval arithmetic for strings, the assumption-based model refiner currently does not support assumptions over signals of type string. In order to enhance the results of the analysis, results of the reset analyzer are incorporated together with the user-given assumptions into the initial data-flow element.

After the computation of the maximal fixed-point solution, timing-dead statement and edges can be computed. Thus, every node and every edge of the control-flow graph must be revisited once. The CRL2 frontend of *PAG* already provides an interface

```
kfg_forall_edges (cfg, <callback function>, NULL);
kfg_forall_nodes (cfg, <callback function>, NULL);
```

allowing to iterate over the set of edges E and the set of nodes V, respectively. For each edge (or node), a user-defined callback function is executed allowing to access properties of the edge (or node). By this, marking of timing-dead edges and nodes can be easily implemented. Listing 7.7 on the following page depicts the implementation of the callback function to compute timing-dead information for edges. As defined in Section 6.3 on page 126, timing-dead marks are only computed at the outcome of conditional statements, i.e. for true and false edges. So, the implementation first checks the edge type, and then tries to evaluate the expression of the last statement of the source basic block using the incoming data-flow value (accessed via `dfi_statement_get_pre()` returning the combination of all data-flow values entering the last statement). If the evaluation of the expression returns a static value that differs from the condition

```
1  void compute_edge_infeasible (KFG cfg, KFG_EDGE edge, void*)
   {
3    if (kfg_edge_get_type (cfg, edge) == CRL_EDGE_TRUE
         || kfg_edge_get_type (cfg, edge) == CRL_EDGE_FALSE)
5    {
       //get dfi_value of source of the edge and evaluate underlying constraint
7      CrlBlock* source = kfg_edge_get_source (cfg, edge);
       CrlInstruction *last_stmt = kfg_node_get_last_statement (cfg, source);
9      o_map value_map = o_env_select_2 (dfi_statement_get_pre (cfg, source, last_stmt));

11     Value *result = eval (vhdl_rhand_side (cfg, source, last_stmt), value_map);

13     if (result->valid ()) //not top, nor bot
       {
15       if (result->lowerIntVal () == result->upperIntVal ())
           if ((!result->lowerIntVal () && kfg_edge_get_type (cfg, edge) == CRL_EDGE_TRUE
                )
17              || (result->lowerIntVal () && kfg_edge_get_type (cfg, edge) ==
                    CRL_EDGE_FALSE))
           {
19           //mark edge as timing dead
             edge->set_sym ("vhdl_timing_dead", true);
21
             //also mark the if statement as timing dead
23           last_stmt->set_sym ("vhdl_timing_dead", true);
             last_stmt->single_operation ()->set_sym ("vhdl_timing_dead", true);
25         }
       }
27   }
   }
```

Listing 7.7 – Computation of timing-dead edges.

induced by the edge type, the edge is marked as infeasible. Please note that the function does not compute the transitive closure on timing-dead edges, and it is the task of the analyzer to propagate a neutral data-flow element over these edges, or to *bypass* data-flow value computation on these paths. For the later, *PAG* offers a dedicated skip directive, called BYPASS.

The goal of the assumption-based model refiner is to identify statements that can be removed from a VHDL specification because their outcome is known to be static. Thus, if one outcome of a conditional statement can be marked as timing dead, the condition is known to be constant, and thus, also the conditional statement can be marked as timing dead (cf. line 23 f. in the above listing).

Similar to the computation of timing-dead edges, also statements can be marked as timing dead. Therefore, the set of stable identifiers $S_{stable_{Int}}$ as defined in Section 6.3 is computed first. The set is then to be used when examining every node of the control-flow graph adding timing-dead marks to those statements assigning a value to an element of $S_{stable_{Int}}$.

Also the live-variables analysis as a first backward analysis using the VHDL analysis framework has been implemented to enhance the results. The domain for the analysis is the power-set domain of identifiers used in a VHDL description.

```
SET
// basic block, instruction number, context
node = snum * snum * snum

DOMAIN
node_set = set (node)
// domain for postdominator and dominator analyses
dom = lift (node_set)
// mapping from identifiers to set of definition points
map = str -> node_set
// domain for reaching definition analysis
rd = lift (map)
```

Listing 7.8 – Lattice specification of the backward slicer.

Due to the definition/use classification already present in the CRL2 description, implementation is straight forward. The maximal fixed-point solution of the data-flow problem is then used to further classify statements as timing-dead (cf. Section 6.3.2 on page 139).

Finally, the results of both analyses are added to the CRL2 description. As timing-dead statements and statements that are no longer reachable when skipping timing-dead edges during analysis, are to be purged from the specification, their constant values need to preserved for further analyses. Thus, a mapping $assumptions\colon identifers \rightarrow V_{Int}$ mapping identifiers to their constant value is added to the CRL2 description. This mapping can be used for accessing the value of signals that could statically be proven to be constant. Furthermore, assumptions specified by the user are added to this map. Obviously, an assumption must persist, otherwise results cannot be guaranteed to be correct. Information from the assumption map are also automatically added to the initial data-flow elements of subsequent runs of the assumption-based model refiner allowing to compute the transitive closure of stable signals.

7.2.5 Backward Slicer

Within the timing model derivation cycle, slicing is of utmost importance. Besides its usage in the timing dead code elimination phase, it further supports model understanding. Thus, slicing as described in Section 6.4 on page 140 has been implemented using *PAG* to automatically generate the required data-flow analyzers. The slicing algorithm has been implemented using C++. The specifications of the three collaborating analyses, namely reaching-definition analysis, post- and dominator analyses, are straight forward to their definitions and are

mostly omitted here for readability reasons. Only the domain specification will be detailed.

As stated before, a CRL2 description consists of routines, basic blocks, instructions, and operations. The later is always enclosed into the former. For interprocedural analyses, *PAG* uses the static call-string approach [SP81] together with the basic block optimization to speed up computation time. In order to ease implementation, but also to ease specification of slicing criteria, basic blocks are assigned a unique id, called *basic block id*, and the list of instructions contained in an basic block are numbered according to their sequential ordering within the basic block. Thus, each node in the control-flow graph $G = (V, E, s, x)$ can be addressed via the tuple $(b, i, c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, where b refers to the unique basic block id, i refers to the linear ordering number of the node within the basic block, and c refers to the context of the node within the mapping computed by *PAG* (cf. Section 6.1.1 on page 99).

The combined lattice specification for the reaching definition analysis, the postdominator analysis, and the dominator analysis is given in Listing 7.8 on the previous page. A node of the control-flow graph can be uniquely specified using the above tuple and is specified here using the predefined *PAG* data type `snum` representing the signed numbers. Building on that definition, the power-set domain `node_set` of nodes is specified. Extended with additional least and greatest elements, this domain forms the lattice for the postdominator and dominator analyses. Also the function domain mapping identifiers to sets of nodes is extended with additional least and greatest elements to form the lattice for reaching definition analysis.

Using *PAG* on this specification and the problem and transfer-function specifications results in three data-flow analyzers that can be combined to a *backward slicer* for VHDL models. Given a CRL2 description of the VHDL model to be analyzed and a slicing criterion using the above notation, the backward slicer returns an approximation to the minimal slice for the specified criterion. As the results from the three different data-flow problems are independent from any slicing criterion, it is also possible to specify a set of slicing criteria resulting in the unification of the individual slices to be returned. Thus, the backward slicer can be directly used for timing dead code elimination as described in Section 5.2.1 on page 88.

In order to support model understanding, also an interactive version of the backward slicer has been developed. Figure 7.3 on the next page depicts the tool interaction of the interactive slicer: Given a CRL2 description, the tool *crl22gdl* converts the control-flow representation into the graph description language, which can be interpreted by *aiSee*. Both tools have been provided by AbsInt Angewandte Informatik GmbH. The CRL2 description is also passed to the backward slicer computing the maximal fixed-point solutions of the three analyses. Afterwards, a socket for the communication with aiSee is created

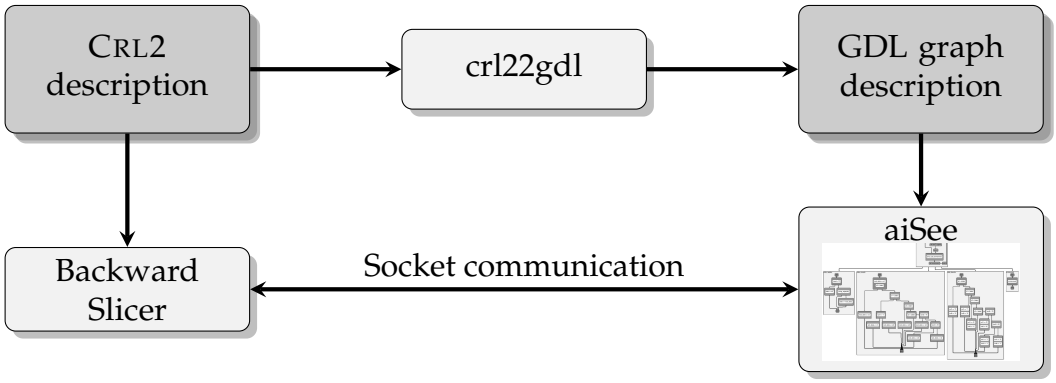


Figure 7.3 – Interactive backward slicer – Tool interaction.

allowing the user to interactively select a slicing criterion. All nodes belonging to the computed slice are then highlighted within aiSee's visualization of the input model.

In order to allow the user to focus on the parts of a design that are currently in focus of interest, the interactive version of the slicer allows to prevent certain dependencies (i.e. flow dependency, control dependency, and activation dependency) from being considered by the slicing algorithm. The resulting slice is no longer a slice matching the definition, but gives the user the freedom to focus on what he believes to be important.

7.3 Evaluation

This section gives an overview on the usability and applicability of the tools presented in the previous section. For evaluation, the VHDL models introduced in Section 7.1 have been used. First, performance and memory consumption of the VHDL compiler and the static analyzers are considered. The suitability of the tools on different VHDL models is subject to investigation in Section 7.3.3. Also the soundness of the analyses results will be further investigated. Finally, a use case building on the superscalar DLX is given applying the typical code of work rules to derive a timing model.

Measurements have been performed on an Intel® Core™ 2 Duo CPU 6700 running at 2.66 GHz each. The test system comprises 8 GB of DDR3 main memory and runs a Linux-based Ubuntu 11.04 "Natty Narwhal" operating system. All runtime measurements and the memory consumptions of the several tools have been measured using `proc-time`, version 1.3.55, from LilyPond.

VHDL model	Parsing	Elaboration	Total
Superscalar DLX	11.02	21.26	32.28
Automotive CPU 2	69.47	82.68	152.15
Avionics MCU	82.97	78.85	161.82
Automotive CPU 1	198.61	191.16	389.77
LEON2	517.55	497.82	1015.37

Table 7.3 – VHDL compiler performance (in seconds) per phase.

7.3.1 VHDL Compiler Performance and Memory Consumption

The evaluation of the VHDL compiler is divided into two parts. The first part focuses on the performance of the compiler, whereas the second part considers its memory consumption. Compiling a VHDL model into a semantically equivalent CRL2 description is split into three phases: model parsing, model elaboration, and CRL2 generation.

Generating CRL2 from an elaborated specification is a quite simple and linear process requiring each statement of the specification to be considered once. Thus, this phase in the VHDL compiler is rather similar to other compiler backends, and its performance is mainly limited by the bandwidth of the file-system of the underlying test systems. Hence, performance and memory considerations for this phase were not made.

Table 7.3 lists the runtime of the VHDL compiler on different VHDL models in order to obtain an elaborated model. The overall runtime is partitioned into the time for model parsing and the time for model elaboration. Figure 7.4 on the next page gives a visualization of the data from the table.

Compiling a VHDL model into a design library is an iterative process. First, the modules already present in the design library have to be loaded. Second, the current module has to be parsed, and finally it has to be added to the design library. As this process is to be performed for every design unit a VHDL specification comprises, the overall runtime of the parsing phase is dependent on the number of design units, i.e. the n^{th} unit requires at least $n - 1$ modules to be loaded from the design library. The runtime for parsing a single module, i.e. the isolated runtime of parsing the new module, is linear to the lines of code of the module.

Elaboration of a VHDL model is performed to flatten the design hierarchy by

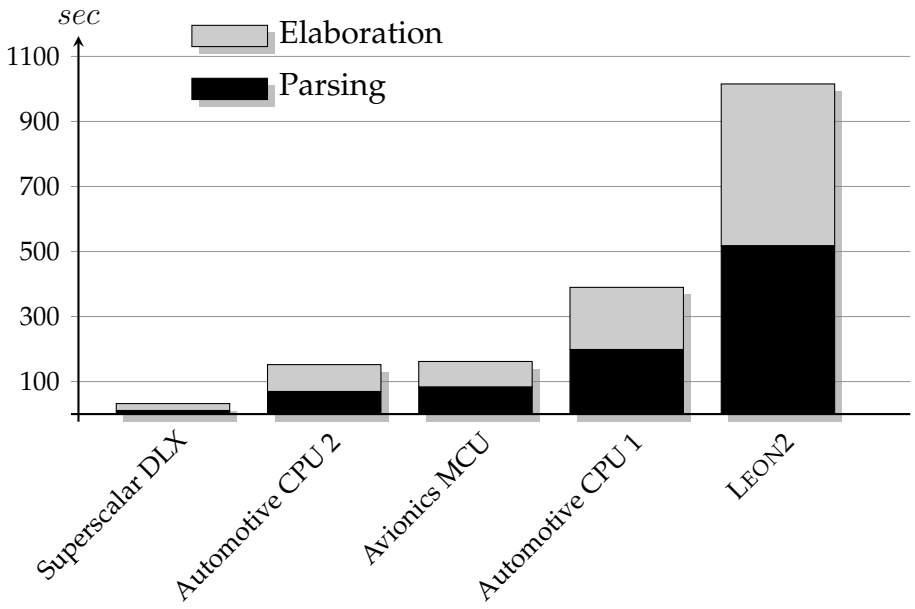


Figure 7.4 – Performance of the VHDL compiler.

instantiation all referenced components and inlining their implementation. Due to the use of generate statements within most VHDL specifications, instantiation of components is an iterative process. Thus, the runtime of the elaboration phase depends on the number of component instantiations contained in the model, but also on the size of the entities to be instantiated. In general, it can be stated that the more entities a VHDL specification comprises, the more components have to be instantiated, and thus, the more complex the elaboration phase is.

Figure 7.5 on the following page visualizes the relations between different terms of model complexity (lines of code, number of entities, and total number of design units) and the runtimes for model parsing and model elaboration. Model complexity in terms of lines of code, and also design complexity in terms of number of design units directly influence the performance of the parsing phase. Design complexity in terms of number of entities influences the performance of the elaboration phase. Please note that the number of entities is not always an expressive number: e.g., memories are normally composed of a rectangular array of storage cells. Each cell is an unique instantiation of one design entity. Thus, a memory can be specified by only two design entities using thousands of component instantiations. In general, counting of component instantiations is not simple due to the use of generate statements at different hierarchical levels, thus, the number of entities is used for determining the complexity of a design.

Table 7.4 shows the memory consumption of the VHDL compiler in megabytes. Since parsing and elaboration phases run sequentially and require at least two

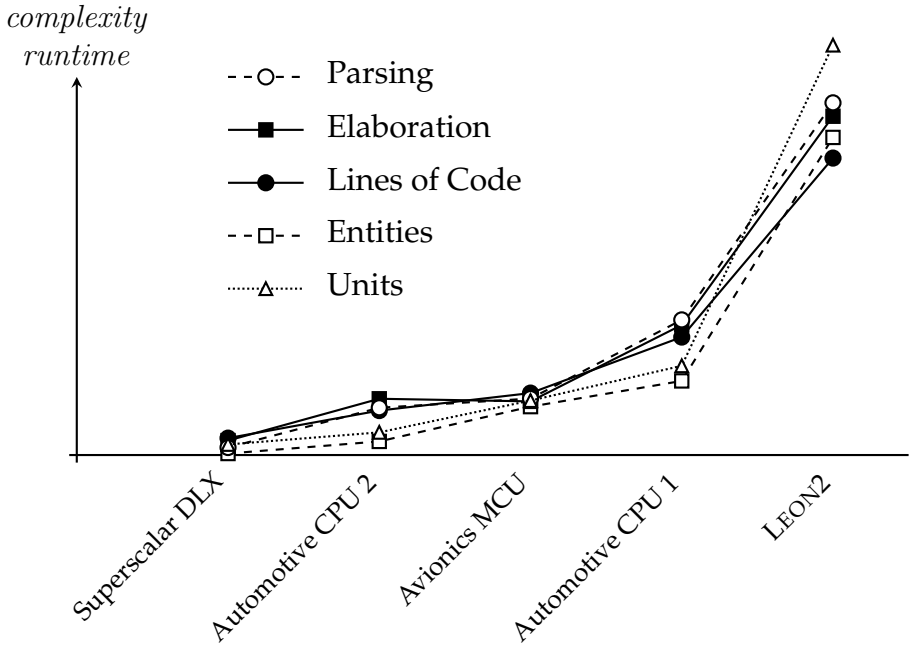


Figure 7.5 – Relationship of VHDL model complexity and VHDL compiler performance.

runs of the compiler, the memory consumption is given per phase. Figure 7.6 visualizes the data given in the table.

The memory consumption of the parsing phase increases linearly with the number of lines of code. The maximal value is achieved while parsing the last design unit of a VHDL specification, as all other modules need also to be loaded. Thus, the memory consumption while parsing the smallest VHDL specification, the superscalar DLX, is the lowest, whereas the most complex specification of the LEON2 processor requires the most memory for model parsing. Obviously, this depends on the fact that the whole specification of a processor resides in the main memory when parsing the last design unit.

In contrast to that, the memory consumption of the elaboration phase is dominated by the number of component instantiations. Similar to the parsing phase, the whole specification of the model is loaded from the design library into main memory. Elaboration as defined in the VHDL standard [IEE87] requires every component instantiation in the specification to be replaced by the component's architectural body. Thus, elaboration increases the size of the model while flattening the design hierarchy. In general, the more design entities – design packages do only provide shared functionality that can be analyzed using a call-string approach – a VHDL specification comprises, the more memory is required for elaboration. This trend is confirmed by the results of the evaluation.

VHDL model	Parsing	Elaboration
Superscalar DLX	111	576
Automotive CPU 2	191	989
Avionics MCU	432	1680
Automotive CPU 1	756	3043
LEON2	1929	6960

Table 7.4 – VHDL compiler memory consumption (in MB) per phase.

7.3.2 Analyzer Performance and Memory Consumption

This section describes the efficiency of the analyzers that have been implemented using *PAG*. As in the previous section, evaluation is split into two parts for each analyzer examining the analyzer's performance and its memory consumption.

Runtime considerations have been partitioned into the time required to build up the control-flow graph from a CRL2 description and calculating a mapping, analysis time, and time for evaluating the results of the analysis. The control-flow graph and the mapping need to be constructed/computed only once, independently from the number of analyses the tool comprises. In contrast to the performance consideration, the memory consumption is considered per analysis. Every analysis requires the data structure containing the control-flow graph and the mapping, whereas the evaluation of analysis results does not increase the memory consumption.

In following, the runtime performance and the memory consumption of the three different analyzers described in Section 7.2 on page 165 will be given. Results are given for the VHDL models described above, namely the models of the superscalar DLX, Automotive CPU 1, Avionics MCU, Automotive CPU 2, and LEON2.

Reset Analyzer

Table 7.5 on the following page lists the runtime of the reset analyzer on the different models used for evaluation. A visualization of the data is given in Figure 7.7. The overall runtime of the analyzer is dominated by the analysis itself, whereas the runtime of constructing the control-flow graph and the evaluation phase are negligible.

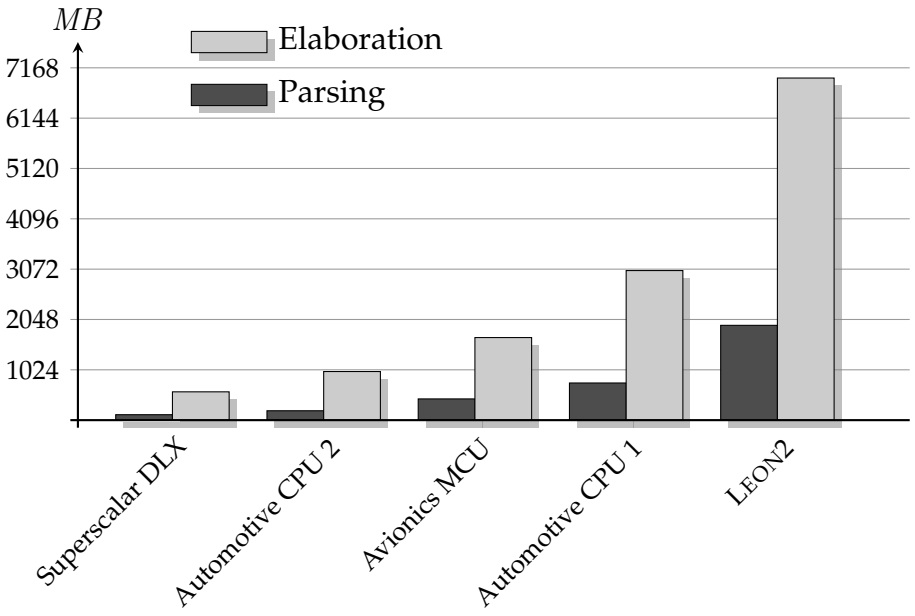


Figure 7.6 – Memory consumption of the VHDL compiler per phase.

The measured runtimes of the reset analyzer are impressively low even for large processor specifications. In general, the runtime increases with increasing model complexity showing an surprising cut for the specification of the Avionics MCU. This fact is caused by the implementation of reset handling within the design: in contrast to the other specifications, the reset handling in the Avionics MCU is strictly spatially uncoupled from the remaining parts. This fact allows the analyzer to skip large parts of the control-flow graph describing this model and yields to a tremendous speed-up in analysis time.

Meanwhile, the runtime of parsing a CRL2 description and building up the internal data structures required for analysis increases linearly with the number

VHDL model	Control-flow	Analysis	Evaluation	Total
Superscalar DLX	0.30	4.00	0.02	4.32
Automotive CPU 2	0.36	4.73	0.03	5.12
Avionics MCU	0.55	0.90	0.04	1.49
Automotive CPU 1	0.59	5.20	0.45	6.34
LEON2	0.71	7.97	0.54	9.22

Table 7.5 – Reset analyzer performance (in seconds) per phase.

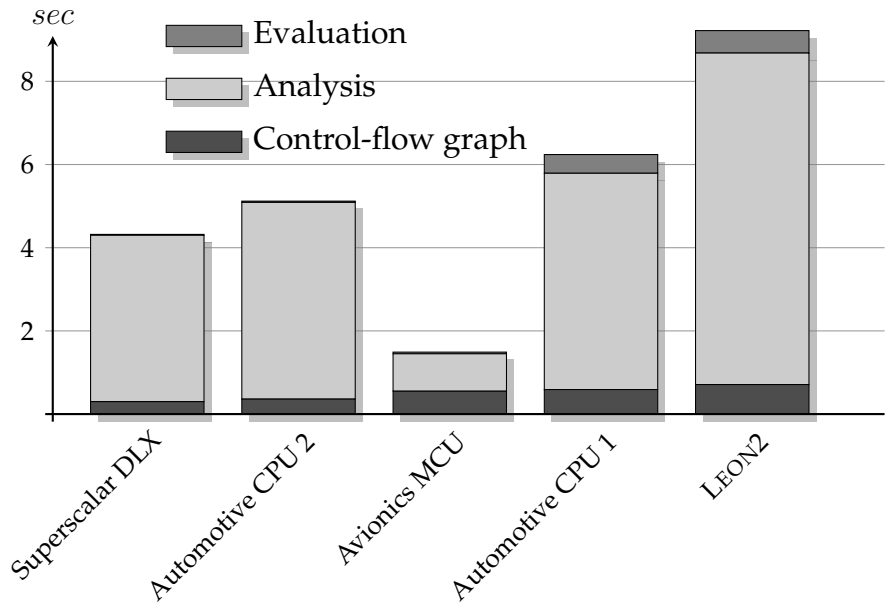


Figure 7.7 – Performance of the reset analyzer.

of statements contained in the VHDL specification, and thus with the number of lines of code of the model.

The runtime of the evaluation phase of the reset analyzer is dependent on the number of signals contained in the data-flow values: the more identifiers are contained in the data-flow values at the return edges of the clock simulation routine (cf. computation of the set $S_{stable_{cp}}$, Section 6.2 on page 113), the longer the evaluation phase takes.

In general, evaluation of large specifications takes longer compared to smaller specifications, as the complexity of a specification in terms of lines of code often is a metric for the number of declared identifiers. But this is only a weak proposition.

Table 7.6 on the following page shows the memory consumption of the reset analysis measured for the five specifications. A graphical visualization is given in Figure 7.8. The consumption is correlated to the runtime of the analyzer: the longer the analyzer runs, the more memory is consumed. The data-flow value has to be stored at each edge (or, if basic block optimization is in use, at least for each basic block), so the larger the specification is, the more memory is consumed. As for the runtime, the memory consumption of Avionics MCU shows a cut compared to its complexity, which is again caused by the strict spatial isolation of the reset handling within this specification.

VHDL model	Analysis
Superscalar DLX	533
Automotive CPU 2	547
Avionics MCU	126
Automotive CPU 1	654
LEON2	1136

Table 7.6 – Reset analyzer memory consumption (in MB).

VHDL model	Control-flow	Interval analysis	Live-variables analysis	Evaluation	Total
Superscalar DLX	0.31	10.72	11.51	7.63	30.17
Automotive CPU 2	0.37	11.30	12.76	8.45	32.88
Avionics MCU	0.56	16.06	15.33	18.64	50.59
Automotive CPU 1	0.60	16.67	16.03	19.20	52.50
LEON2	0.73	18.75	17.86	36.02	73.36

Table 7.7 – Assumption-based model refiner performance (in seconds) per phase.

Assumption-based Model Refiner

The assumption-based model refiner comprises four different phases: parsing of CRL2 and building up of internal data structures, an interval analysis to identify stable signals, a live-variables analysis, and an evaluation phase. Thereby, the latter phase always relies on the results of the former phase.

The runtime of the analyzer depends on the assumptions given by the user. If an assumption provided by the user renders most of the parts of a specification infeasible, the runtime of the analyzer decreases. To enable a comparison of runtimes of the analyzer, a similar starting condition has to be provided. As the assumptions that are used for analysis are dependent on both, the hardware and the environment, providing an identical assumption is in general impossible. Fortunately, each model used for evaluation of this thesis provides an external interface including the reset signal. For timing model derivation, it is usually a common assumption that the reset does not occur during normal operation, and thus, assuming the absence of the reset is an appropriate assumption. This assumption can be provided for every of the used models, and thus allows for

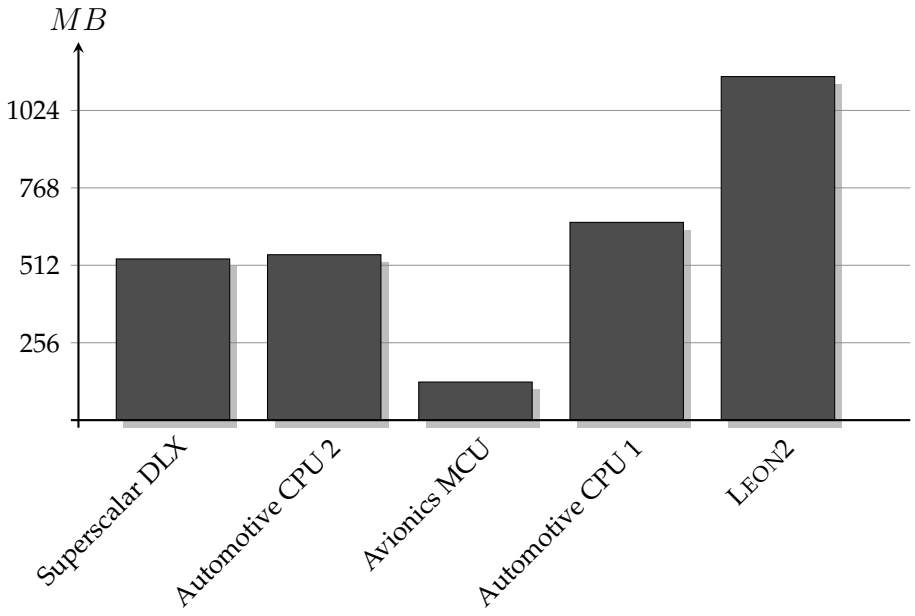


Figure 7.8 – Memory consumption of the reset analyzer.

comparing the runtimes of the assumption-based model refiner.

Table 7.7 shows the runtime of each phase of the assumption-based model refiner for the assumption "the reset is not active". A graphical representation is given in Figure 7.9. Compared to the runtime of the reset analyzer, assumption evaluation is more complex which results in a higher runtime.

The slight increase in runtime of the control-flow graph construction phase is caused by uncertainties during measurement and the additional effort for parsing the results of a preceding reset analysis. The runtime of the parsing phase is negligible compared to the runtimes of the other phases.

The runtime of the interval analysis and the live-variables analysis are comparable. Both analyses have to consider nearly the whole specification of each model. Only the parts that implement reset handling can be skipped.

Evaluation of analyses results is a complex process requiring every node and every edge of the control-flow graph to be considered once. Adding timing-dead marks to edges requires the data-flow values of the several analysis contexts at each edge to be joined. Joining data-flow values entering an edge requires the pointwise join of each function in the data-flow value. Thus, the more elements are contained in the data-flow values, the longer the join takes. As a rule of thumb, we can notice that the larger a specification is in terms of lines of code, the more different identifiers are used, which results in larger data-flow values. This rule is confirmed by the runtimes measured for the evaluation phase.

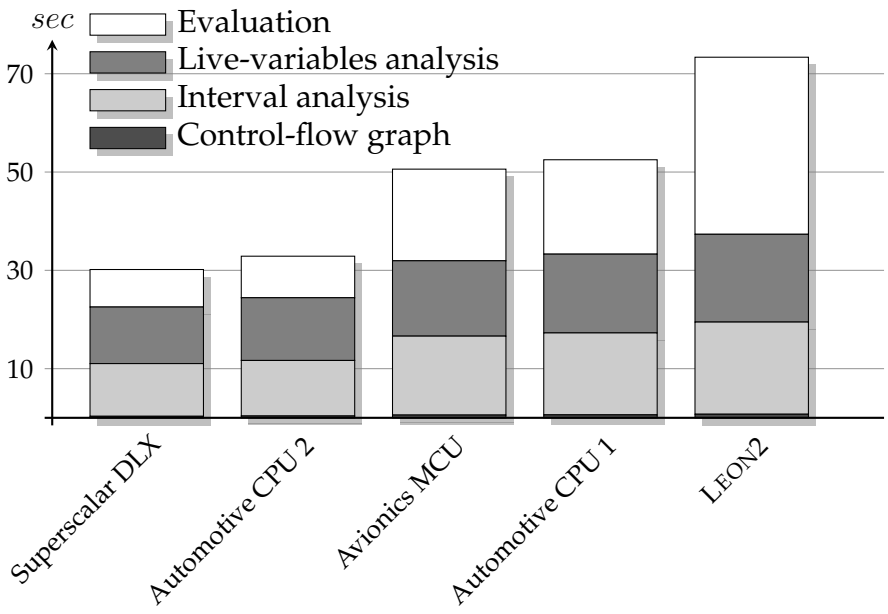


Figure 7.9 – Performance of the assumption-based model refiner.

The memory consumption of the assumption-based model refiner is given in Table 7.8. As the later live-variables analysis relies on the results of the preceding interval analysis, memory consumed by the analyses cannot be separated. Thus, the memory consumed by the live-variables analysis is determined by the memory consumed for interval analysis plus the memory consumed for storing the results of this analysis. In general, we can observe that the memory consumption of the analyzer increases in parallel with the model complexity in terms of lines of code and the number of design units. The larger the control-flow graph of a specification, the more memory is consumed.

It might be a surprise that the memory consumption of the live-variables analysis is much higher compared to the interval analysis, even though if the memory consumption of the interval analysis is counted out. Figure 7.10 on page 194 illustrates this finding. The increase of memory consumed for the live-variables analysis can be easily explained with additional background knowledge on the implementation of function lattices within *PAG*. In order to safe space, *PAG* uses a wildcard element for all identifiers mapping to the default value of a function. Applied to interval analysis, all identifiers whose values are unknown can be efficiently handled. Only those identifiers whose value range could be restricted by the analyzer must be stored separately. Using this implementation allows for saving a lot of space.

In contrast to that, the live-variables analysis computes a superset of all identifiers being live at a certain node of the control-flow graph. Thus, a space-saving

VHDL model	Interval analysis	Live-variables analysis
Superscalar DLX	1233	3372
Automotive CPU 2	1345	3467
Avionics MCU	1323	4176
Automotive CPU 1	1487	4332
LEON2	1488	5321

Table 7.8 – Assumption-based model refiner memory consumption (in MB).

VHDL model	Control-flow	Reaching definitions	Post-dominators	Dominators	Total
Superscalar DLX	0.30	432.57	1.18	1.27	435.32
Automotive CPU 2	0.35	382.43	1.32	1.41	385.51
Avionics MCU	0.53	393.85	1.81	1.95	398.14
Automotive CPU 1	0.61	456.76	1.83	2.01	461.21
LEON2	0.71	63.50	1.62	1.89	67.72

Table 7.9 – Backward slicer precomputation times (in seconds).

implementation as for function lattices is not available for this kind of lattice, which results in the more in memory consumption compared to the interval analysis.

Backward Slicer

The backward slicer comprises three different analyses that closely work together in order to compute slices for arbitrary criteria. A reaching definition analysis is used to reconstruct flow dependencies of a given VHDL specification, the results of a dominator analysis and a postdominator analysis together are used to reconstruct control dependencies. All analyses rely on a control-flow graph data structure that is to be build from a CRL2 description of the model to be analyzed. After the analyses have finished, slices for arbitrary criteria can be computed.

Table 7.9 lists the runtimes of the several parts comprising the backward slicer, a

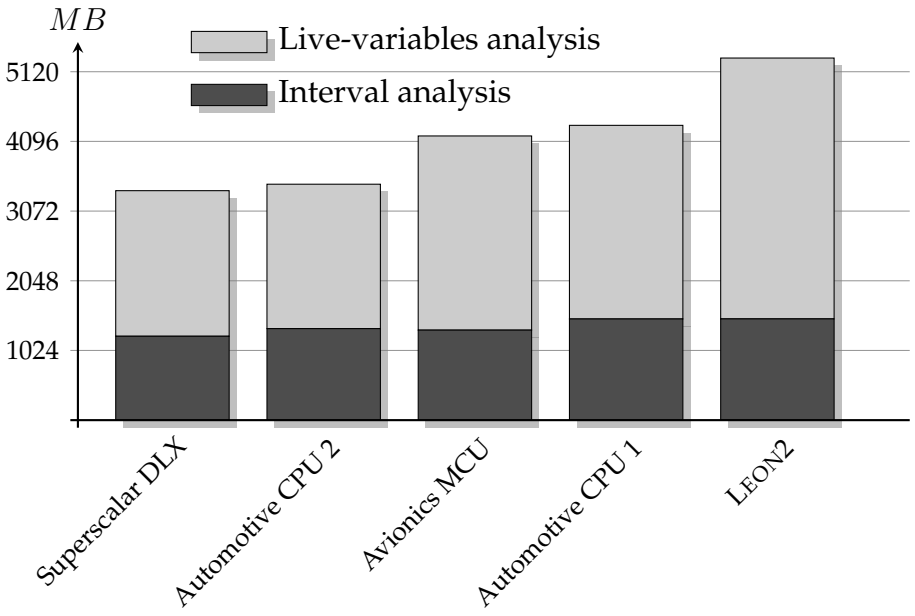


Figure 7.10 – Memory consumption of the assumption-based model refiner.

graphical evaluation is given in Figure 7.11 on the next page. The precomputation time of the slicer is dominated by the runtime of the reaching definition analysis, whereas the times for building up the internal data structures, dominator and postdominator analyses are negligible.

Whereas the runtimes of the dominator analysis and the postdominator analysis linearly increase with the model complexity in terms of lines of code, the runtime of the reaching definition analysis does not. This is caused by the 2-level semantics that is special to hardware description languages: a definition that occurs later in the imperative part of a VHDL process may reach preceding statements. Thus, computing the maximal fixed-point solution or reaching the fixed point in the analysis is more complex. Within the structure of the VHDL analysis framework, the second semantical level is expressed in the synchronize and delta-delay statements by introducing a back-edge to the entry of the simulation routine. Furthermore, a further rise in complexity of computing the maximal fixed-point solution is caused by the recursive structure of the clock simulation routine. The capability of distinguishing several clock cycles during analysis is a feature for all path-sensitive analyses, but induces a drawback for path-insensitive analyses. The high precomputation time is a result of the VHDL semantics coupled with the structure of the analysis framework. Nevertheless, precomputation times at a scale of 400 seconds even for large specifications are reasonable.

Interestingly, the runtime of the reaching definition analysis on the specification

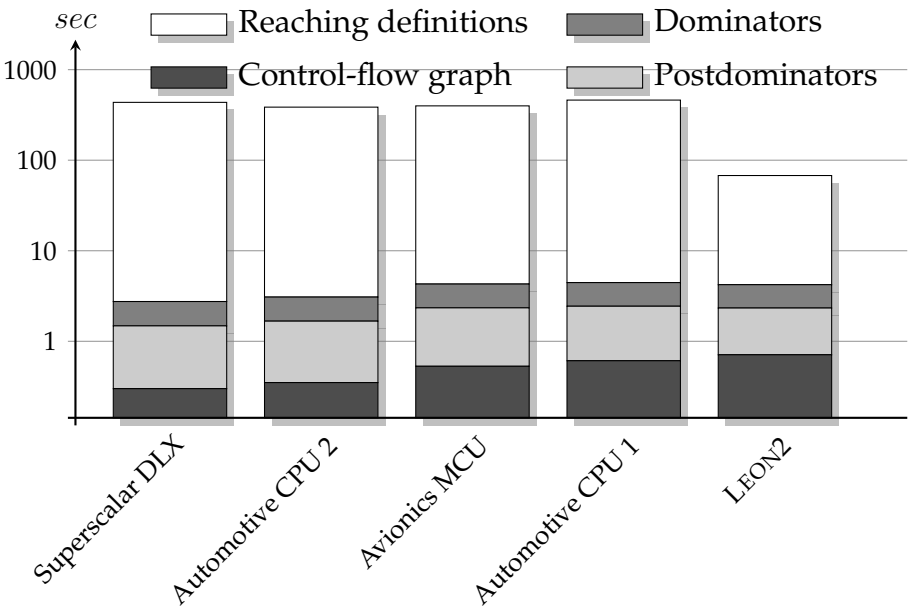


Figure 7.11 – Precomputation times of the backward slicer.

of LEON2 is significantly lower compared to all other measured execution times. This is due to the way how the design of this CPU is structured: LEON2 is composed of many small processes with process-local variables easing the reaching definition analysis. Only key parts are modeled using large data structures. The drawbacks of such a design methodology are described in Section 7.3.3.

The memory consumption of the backward slicer is depicted in Table 7.10, a graphical evaluation is given in Figure 7.12 on the next page. The consumption is given per analysis, but for slicing, the sum of the individual values is required.

VHDL model	Reaching definitions	Post-dominators	Dominators
Superscalar DLX	3898	58	59
Automotive CPU 2	3904	57	58
Avionics MCU	3912	68	68
Automotive CPU 1	3987	71	71
LEON2	3898	60	61

Table 7.10 – Backward slicer memory consumption (in MB).

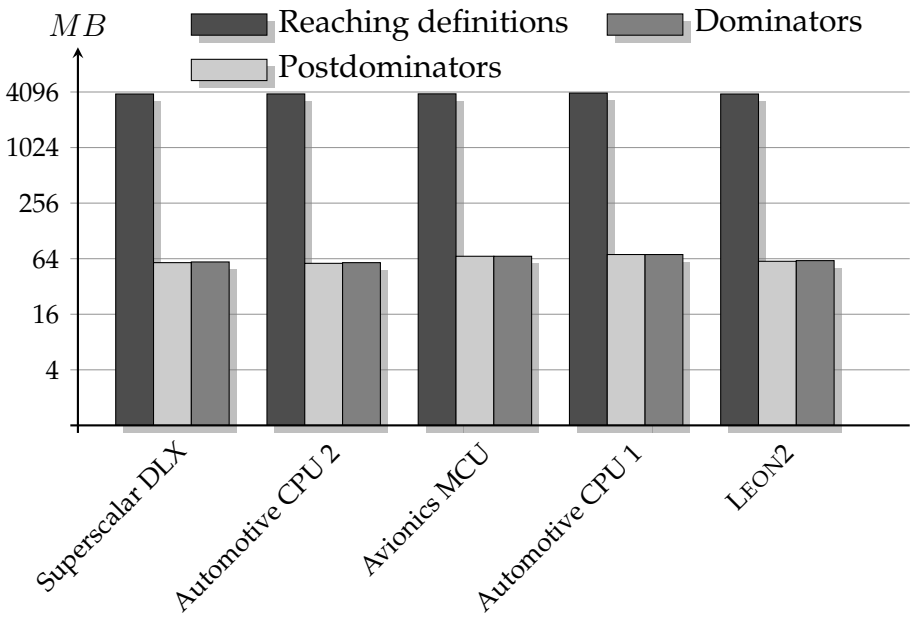


Figure 7.12 – Memory consumption of the backward slicer.

Dominator and postdominator analyses require only a small amount of main memory (around 70 MB for each analysis), whereas the consumption of the reaching definition analysis is significantly higher (nearly 4 GB). This is partly caused by the structure of data-flow information that has to be propagated over the control-flow graph of a specification: where dominator and postdominator analyses use a power-set domain of nodes of the control-flow graph, the reaching definition analysis relies on a more advanced function lattice. The main reason for a nearly identical memory consumption independent from the specification that is to be analyzed is an optimization in the implementation of the analysis. In order to speedup the computation time of the reaching definition analysis, the analyzer preallocates 50% of the main memory that is available at analysis start. As the system used for evaluation comprises 8 GB of main memory, the consumption of around 4 GB can be explained. Therefore, the results are only of limited significance, but it is noticeable that the preallocated memory seems to be sufficient for each specification.

7.3.3 Applicability and Assessment

The analyses presented so far are applicable to arbitrary specifications given in VHDL. The VHDL standard gives designers the freedom to specify a circuit using different idioms. Depending on the idioms used within a VHDL specification of a circuit, the precision of the analyzers, and along with that, the quality of

results might diverge. In following, we will first briefly explain the different idioms to specify a circuit. Thereafter, the impact of them on the results of the different analyses is examined in more detail.

In general, we can distinguish between sequential and combinatorial circuit specifications, but due to hierarchical design composition, also mixed specifications are widespread.

Sequential circuits are those that maintain an internal state. The outputs such a circuit produces in response to given inputs depend on the history of inputs received previously. Hence, sequential designs are usually used for modeling finite state machines. Most sequential designs are synchronous designs using a rising or falling edge of a clock, or a kind of control logic, to control the transition from one state to another. Nevertheless, it is possible to specify asynchronous sequential designs in VHDL, but the synthesizable substandard forbids them.

Combinatorial circuits are those in which the outputs are determined solely by the current values of inputs. The circuit does not maintain an internal state. Anyhow, the specification of a combinatorial circuit may require some internal storage after synthesis that is not obvious in the specification (more details on the synthesis of combinatorial circuits can be found in [Ash01]).

Most current design methodologies used for designing advanced processors and peripheral components prefer synchronous, i.e. edge driven, sequential designs as timing constraints and the storage requirements in hardware are more explicit.

The analyzers presented within this thesis do not rely on a certain kind of idiom used to specify a processor, or parts of it. Nevertheless, some kind of idioms may result in more imprecise results compared to other kinds. In general, it is impossible to provide a metric allowing to classify the quality of analysis results. However, some typical scenarios leading to imprecision will be given. Please note that these scenarios diverge depending on the analysis.

Results of the reset analysis implemented in the reset analyzer do not differ for sequential or combinatorial designs. On the sample specifications used to evaluate this thesis, the initial values of all signals of scalar data types could be automatically determined. Due to the missing support for composite data structures, initial values of identifiers of such data types could not be determined. It has been an interesting finding that the specifications of processors that are widely in use (e.g., Automotive CPU 1 and Automotive CPU 2) or that have been designed for use within a specialized field of application (e.g., Avionics MCU) makes only limited use of composite data types. Moreover, the usage of data types maintaining an internal structure is limited to array types; record structures are usually not used. In contrast to that, the open-source specification of LEON2 comprises nested composite data types of both. Thus, results for this model are not as good as for the four other models used within this thesis.

Besides the identification of initial values, a side product of the reset analysis is the set of possible clock domains. The computed set is overestimated for two of the five models. One model, namely Automotive CPU 2, does not contain an additional clock domain in its specifications. Even though the model describes the memory subsystem, clock signal generation is made externally, i.e. core and bus clock are inputs of the given circuit specification. For the superscalar DLX and the Avionics MCU, the automatically derived sets of possible clock domains exactly contains the clock signals contained in the specification.

A general assessment on the quality of results of the assumption-based model refiner is not possible. Results of the analyzer strongly depends on the assumptions provided by the user making a global evaluation impossible. Furthermore, the assumptions strongly depend on the specific field of application, a processor will be used in. As described earlier, we have only investigated the results of the analyzer with one assumption being guaranteed, namely that the reset signal was inactive. On all models used for evaluation, this assumption has led to the result that reset handling within each model has been marked as timing dead.

In general, results could become less good depending on the structure of the specification to be analyzed. As for reset analysis, indexed and slice accesses of identifiers of composite data structures result in less precise co-domains being automatically computed by the analyzer. This is caused by the fact that these accesses are currently not supported by the analyzer itself. This drawback can be solved by adding support for array data types. Regarding the two types of circuit specifications, it has to be noticed that sequential designs that maintain an internal state lead to worse results compared to combinatorial circuit specifications. This is mainly based on write-enable signals that are used to control the update of internal registers. If the value of these signals cannot be determined, uncertainties on the contents of internal registers arise, which lead to a loss in precision.

Assessing the quality of slices computed by the backward slicer is difficult. The term *quality of slice* has already been introduced in Section 6.4.5 on page 155. A complete assessment would require the computation of minimal slices, which is in general not possible nor deterministic [Wei79]. Furthermore, a slice is always computed for a certain criterion making general assessments impossible. [Sch05] describes a similar approach, i.e. an approach using reconstructed dependencies, to compute slices on binary programs. Also the quality of slices computed for some selected criteria was examined there and evaluated as well.

Slicing using program dependencies is independent from the kind of idiom used within a specification. Thus, sequential and combinatorial specifications lead to the same quality of slices. Uncertainties in the reconstruction of flow- and control dependencies leading to less precise slices arise from two things:

- Array data types and indexed accesses lead to some overestimation in the reconstruction of flow dependencies since identifiers of array types are treated as a whole in the reaching definition analysis.

- Mutual exclusiveness in the imperative part of processes may lead to some overestimation, if the conditions cannot be statically evaluated.

Some of the processor specifications used for evaluation have been developed in compliance with some high-level design paradigms that are mainly in use for design and implementation of large software projects, but only applicable in a limited way to hardware description languages. One example is the LEON2 specification encapsulating components into huge data types in order to define proper interfaces. Unfortunately, also the sensitivity lists of processes are studded with references of signals of these data types, leading to huge activation dependencies after elaboration. An example within the LEON2 is the implementation of the data cache. In order to check for a current bus activity of the instruction cache, one certain bit of this cache implementation is required. But due to the design paradigms, the sensitivity list contains a reference to the whole data structure describing the instruction cache. Since the standard does not require a signal contained in the sensitivity list of a process to be used within its implementation, restricting the sensitivity lists to the set of used signals is not possible in general. For the LEON2 data cache implementation, this results in every signal of the data cache to be dependent on every signal of the instruction cache. As slicing computes a transitive closure on the dependencies, the resulting slices become very unsatisfactory and imprecise.

7.3.4 Soundness

Results of the different analyzers can be proven to be correct. Whereas the soundness of backward slicing already has been proven, the soundness of the reset analyzer and the assumption-based model refiner are subject to this section.

In order to show their correctness, formal verification has been used. Formal verification analyzes a mathematical model of the design and proves that the design exhibits the desired behavior for all possible input patterns. Nowadays, formal verification plays an increasingly important role in system-on-chip design, because it has a big advantage with respect to simulation: in principle, formal verification can detect all bugs in a design model.

Complete formal verification [Bor09] is a methodology combining operation-based circuit representation, interval property checking, and a completeness checker. In following, the principles of complete formal verification will be

briefly introduced. Subsequently, its application to proof the soundness of analysis results will be detailed.

An operation-based circuit specification consists of a set of operation properties. A single *operation property* expresses a single design transaction, which is a transition between abstract, high-level design states. A transaction describes the behavior of the design in an abstracted way, being free of implementation details. Properties have an implicative structure consisting of an *assume* and a *prove* part. The assume part of a property is used to restrict and describe the (abstract) starting state and the condition under which an operation is to be executed. The prove part describes the expected output resulting from the operation and the abstract state where the transition ends. Operation properties can be derived from a transaction-based representation of a circuit. Combining several operation properties results in an abstract description of a circuit.

Proving the correctness of operation properties is done using *interval property checking* (IPC). The implementation of the circuit to be proven is translated into an equivalent Mealy machine. IPC can be used to relate input-, output-, and state variables of a circuit represented as a Mealy machine within a bounded time interval $[t, t+n]$. Thus, IPC is able to prove operation properties derived from a high-level transaction-based representation. IPC checks these properties for all time points $t \geqslant 0$ and all possible starting states. To do so, the transition and output functions of the Mealy machine are unrolled $n+1$ times. This creates instances of the functions for each time point $t, t+1, \cdots, t+n$ allowing to substitute the variables of the property by the function's output. This results in a Boolean function whose arguments are the instances of the input variables for each of the Mealy machines at time point t till $t+n$ and the instance of the state variables at time point t. The circuit meets the original property if the boolean function returns true for all arguments. If the boolean function returns false, a counter example can be constructed by considering the state at time point t and applying the sequence of inputs for the time points t till $t+n$. Although, the reachability of the circuit state at time point t is not guaranteed by IPC, i.e. it is not guaranteed that the initial state at time point t can occur during normal operation of the circuit. If not, the counter example is unrealistic, and the proof needs to be redone with additional properties restricting the set of initial states. Checking the boolean function for zero is made using an arbitrary SAT-solver. In general, IPC is related to bounded model checking (BMC) with the difference that BMC relies on a time interval starting at a fixed time point 0. More details on IPC, complete formal verification, and operation properties can be found in [Bor09, USB+10, LWS+10].

Complete formal verification using IPC has been successfully applied to the Infineon TriCore2 processor [BBM+07] and other components like DMA controllers, bus arbiters, SDRAM-controllers, and ECC protection. The methodology has been integrated into the tool OneSpin 360 MV [One06].

Interval property checking of operation properties can also be used to prove the soundness of results of the reset analyzer and the assumption-based model refiner. In order to do so, we have closely cooperated with the group of Prof. Kunz at TU Kaiserslautern.

The result of the reset analyzer is the set of stable identifiers $S_{stable_{cp}}$ together with their initial values (that can be obtained via $MFP_{cp}(x)(s)$ for all $s \in S_{stable_{cp}}$). Using this information, an operation property for the reset behavior of a system can be derived that can be proven using IPC. The assume part of the property is specified as "the reset signal is active", the prove part is specified as $\forall s \in S_{stable_{cp}} \colon s = MFP_{cp}(x)(s)$ at $t + n$, where n is at least the number of cycles of the longest activation chain. If interval property checking on the original (unmodified) VHDL sources of the circuit specification returns true for this property, results from the reset analyzer are proven to be correct.

Equivalence checking verifies the functional equivalence of two models of the same design. Traditionally, equivalence checking is used to verify transformations and optimizations that are done automatically by synthesis tools. Two designs are said to be *functionally equivalent* if they produce the same output for all possible input patterns. Thus, equivalence checking can also be used to proof the results of the assumption-based model refiner. The unmodified VHDL specification serves as the original model, which is assumed to be correct. This model is also called the *golden model*. The annotated model, i.e. the CRL2 description including timing-dead marks, derived after applying several user assumptions is used as the second model. To make equivalence checking feasible, the annotated model must be available in VHDL. Fortunately, reconstructing of VHDL from CRL2 is feasible [Pis12]. All statements and transitions in the CRL2 description that have been marked as timing dead are skipped during reconstruction resulting in a modified VHDL model. The equivalence of both models is to be proven using the set of user assumptions. A successful proof guarantees the soundness of the assumption-based model refiner with respect to the set of assumptions.

In cooperation with TU Kaiserslautern, results of both analyzers could be proven to be correct.

7.3.5 Code of Work Rules

Previous sections have dealt with runtime considerations of the tools described within this thesis, their applicability, and also their soundness. This section now describes the typical working patterns used to derive a timing model from a high-level specification given in VHDL. The specification, which is used within this section, is that of the superscalar DLX machine [TUD]. As the topic of this thesis is limited to static analysis of hardware description languages, this section focuses on the model preprocessing phase of the timing model derivation cycle

```
process
begin
    wait on Clock until Clock = '1';
    < process body >
end process;

        ⇓

process(Clock)
begin
    if rising_edge(Clock) then
        < process body >
    end if;
end process;
```

Listing 7.9 – Superscalar DLX code change.

(cf. Section 5.2 on page 87), and therein on environmental assumption refinement and timing dead code removal.

The model of the superscalar DLX comprises around 4000 instructions distributed over 849 processes after elaboration. Indeed, a specification of a size like the superscalar DLX is already suitable (with some slight extension to cope with non-determinism) for use within a static timing analyzer framework, but it also allows for illustrating the general code of work rules.

The implementation provided by TU Darmstadt has a slight drawback: the specification does not fulfill the requirements defined in the synthesizable substandard of VHDL. As this standard defines language constructs and idiom maps that shall be used for hardware synthesis, slight modifications at the source code of the specification had to be made. Furthermore, the analysis framework currently limits the VHDL specifications to rely on sensitivity lists instead of explicit wait statements. Therefore, the specification of the superscalar DLX has been modified. Listing 7.9 depicts the transformation of the explicit wait statement into a semantically equivalent sensitivity list. The standard of VHDL describes the semantics of a sensitivity list of a process as an implicit wait statement at its end. The condition clause "`until Clock = '1'`" can be transformed to a conditional checking for a rising edge of the clock signal.

In order to derive a timing model, an examination of the physical environment of the field of application, the processor shall be used in, is required. The specific type of use of a processor in its surrounding environment offers the most potential for assumption-based model refinement.

The methodology to derive a timing model from a VHDL specification of a

```
entity Dlx is
    port (IncomingClock : in bit;
          BusClock : out bit;
          AddressBus : out TypeWord;
          DataBus : inout TypeBidirectionalDataBus;
          ByteEnable : out unsigned( 7 downto 0 );
          TransferStart : out bit;
          WriteEnable : out bit;
          TransferError : in bit;
          TransferAcknowledge : in bit;
          InterruptRequest : in bit;
          CacheInhibit : in bit;
          Reset : in bit;
          Halt : out bit);
end Dlx;
```

Listing 7.10 – External interface of the superscalar DLX processor.

processor as introduced in this thesis starts with the model preprocessing phase. The process of environmental assumption refinement is based on the knowledge of initial values of signals and variables used in the specification. Thus, timing model derivation usually starts with determining the initial values using the reset analyzer.

In order to do so, the name and the activation value of the reset signal and also the name of the external clock signal have to be determined by manual examination of the VHDL model sources. As the synthesizable substandard of VHDL does not allow a modeling of the frequent change of a clock signal, and basing on the fact that a system reset is often triggered by the environment surrounding a processor, both signals usually can be found in the interface specification of the processor, namely the topmost entity description of a design, which is known due to elaboration. Listing 7.10 shows the external interface of the superscalar DLX machine declaring the signals `Reset` and `IncomingClock`. A further lookup in the topmost entity's architectural body yields that the reset signal within this processor is modeled as an active-high signal. Providing both information to the reset analyzer, the initial values of the signals depicted in Listing 7.11 on the following page can be automatically derived.

Additionally, reset analysis computes the set of possible clock domains. For the superscalar DLX, two signals, `Clock` and `BusClock`, are identified as possible clock domains. The correctness of the set of possible clock domains could be proven by manual examination of the VHDL sources of the processor. For example, the signal `Clock` is defined within the sources as

```
Clock <= IncomingClock and not DP_HaltFlag;
```

```
ALU_ValidFlag = 0
BIU_ActiveFetchFlag = 0
BIU_ActiveLoadFlag = 0
BIU_ActiveStoreFlag = 0
BIU_FirstBusClockOfActiveCycleFlag = 0
BRU_ValidFlag = 0
BTB_ValidFlag =0
CU_NextCommitPointerReg = 16
DC_ValidFlag = 0
DP_HaltFlag = 0
DP_InterruptEnableFlag = 0
DP_ProcessIdentifierReg = 0
DTB_ValidFlag = 0
IC_ValidFlag = 0
IF_ValidFlagA = 0
IF_ValidFlagB = 0
ITB_ValidFlag = 0
LSU_EA_ValidFlag = 0
LSU_SPR_ValidFlag = 0
LSU_ValidFlag = 0
MDU_ValidFlag = 0
RB_ValidFlag = 0
WB_EntranceValidFlag = 0
WB_ValidFlag = 0
```

Listing 7.11 – Initial signal values of the superscalar DLX identified by the reset analyzer.

and is further used for all internal processor updates.

Using the results from the reset analysis, the assumption-based model refiner can be used to identify timing-dead parts in the specification to reduce their size. Our experience shows that the inputs of the external interface of a processor design provides a good start point for assumption-based model refinement. Depending on the specific usage of a processor within its surrounding environment, assumptions on the behavior of the environment may differ. For the example derivation shown is this section, we assume the following reactions of the environment:

- the reset is not triggered during normal system operation (which is normally the case),
- no external interrupt occurs (as external interrupts are simply not predictable), and

```
Reset = 0
InterruptRequest = 0
TransferError = 0
DP_TakeExternalInterrupt = 0
BIU_TransferErrorFetch = 0
BIU_TransferErrorLoad = 0
BIU_TransferErrorStore = 0
CU_Inhibit = 0
IF_TransferErrorFlagA = 0
IF_TransferErrorFlagB = 0
IF_TransferErrorFlagB_Input = 0
IF_TransferErrorFlagA_Input = 0
```

Listing 7.12 – Stable signals of the superscalar DLX based on the assumptions no reset, no external interrupts, and no transfer errors.

- there is no faulty transmission on the external bus (as transfer errors can handled by statistical means).

The three propositions can be easily mapped to assumptions on the value range of the corresponding signals. The "reset is not active" is equivalent to the statement "the value of the identifier `Reset` is always in the interval $[0, 0]$". Similarly, the absence of an external interrupt request can be expressed as `InterruptRequest` $\in [0, 0]$, and the absence of transfer errors on the system bus can be expressed as `TransferError` $\in [0, 0]$.

Providing these assumptions together with the name of the external clock signal (`IncomingClock`) to the assumption-based model refiner, 43 VHDL statements can be marked as timing dead. Furthermore, the outcome of five conditionals can be statically predicted, such that large parts concerned with interrupt handling and transaction repetition become unreachable. Also 53 VHDL processes could be statically proven to be timing dead based on these assumptions. Listing 7.12 lists the signals that become stable under the set of assumptions. Please note that only stable identifiers are reported by the assumption-based model refiner and assignments to them are marked as timing-dead. Unstable identifiers with new restricted co-domains are not reported and are handled internally by the analyzer.

In this example, all assumptions have been passed to the analyzer at once. The results of the assumption-based model refiner will be the same, if assumptions are provided sequentially.

Up to now, only input signals of the VHDL specification have been examined. The next step focuses on the output signals defined in the interface of a processor. For example, the superscalar DLX machine drives six signals comprising the

outgoing interface of the processor. Whenever the semantics of an identifier is not clear to an engineer working on an abstract timing model, backward slicing is helpful in getting the missing intuition.

A backward slice for the output signal `Halt` on the superscalar DLX yields that this signal is a carbon copy of the internal signal `DP_HaltFlag`. This signal is set to `'1'`, if the halt instruction of the DLX instruction set is decoded in the dispatch stage, and is only reassigned during system reset. As this signal is also used to compute the value of the internal clock signal `Clock`, a decoding of a halt instruction stalls the whole processor until a reset is triggered. Besides the fact that this issue is a bug in the implementation of the superscalar DLX, for guaranteeing timing bounds, it is safe to assume that the halt signal will never be triggered. Further assumptions that might be used for timing model derivation are the absence of illegal instructions, or the absence of exceptions that are caused by the software running on the processor.

Based on the resulting model, interactive slicing coupled with the documentation of the processor is to be used to explore, how instructions flow through the several stages of a processor pipeline. This allows identification of points in the processor, where instructions leave the pipeline, i.e. where instructions retire. Within the superscalar DLX processor, instructions retire at one dedicated centralized place, namely the reorder buffer that maintains the instruction order (cf. Figure 7.1 on page 162). Since the superscalar DLX is able to retire two instructions within one clock cycle, the two oldest entries in the reorder buffer are of interest. Within the VHDL specification, the signals `RB_NextInstrToCommitA` and `RB_NextInstrToCommitB` are used to determine the two oldest instructions. Combining the backward slices for the assignment statements to both signals results in a set of nodes $S \subseteq V_{\text{VHDL}}$. All nodes $v \in V_{\text{VHDL}} \backslash S$ are guaranteed to have no effect on the behavior of the instruction pipeline, and thus can be marked as timing dead.

Following the structured methodology of timing model derivation as introduced in Chapter 5 coupled with the static analyses described in Chapter 6, the process of deriving a timing model of a modern processors suitable for its usage within a timing analyzer framework is faster, more efficient and less error-prone compared to the traditional approach based on processor documentation and measurements. Details on data-path elimination, processor state abstractions, and the tool-based generation of micro-architectural analyses can be found in [Pis12].

8

Conclusion and Outlook

> Don't cry about something that passed. Smile because it has happened.
>
> *(Gabriel García Márquez)*

An important part in the design of hard-real time systems is the proof of timeliness. Therefore, the worst-case execution times of the tasks comprising a system need to be known. There exists a bunch of tools for estimating the runtime of tasks, but for sophisticated architectures, static approaches are currently viewed as the only safe and practical technique for obtaining bounds on the execution time that are provable correct. The most promising tool that is in industrial use and whose results and methodology are accepted by certification authorities is the *aiT* WCET analyzer.

Performance-enhancing features like caches, deep pipelines, branch prediction, and speculative execution have an increasing impact on the average as well as the worst-case performance, but their history sensitive behavior make a precise and fast analysis of the hardware difficult. Timing anomalies that are mainly caused by the interaction of these features furthermore increase the search space, a static timing analyzer has to cope with.

Estimating the worst-case execution time of a task relies on a timing model of the hardware of the system. Currently, timing models used by *aiT* rely on the documentation of the system's processor and are hand-crafted by human experts. Also knowledge derived from execution traces of the real hardware is used to improve these timing models. Thus, the derivation of a timing model is a time-consuming, but also error-prone process. This is mainly caused by

missing or wrong documentation, misinterpretation of trace results, and the human involvement. Due to validation issues that are caused by the limitations of the current process, the deployment of a timing model for a sophisticated processor takes several person months, sometimes years.

The increasing demand for more and more computing power even in the area of embedded systems and the decreasing time to market has led to the development of hardware description languages that ease communication of the design teams, but also ease hardware simulation. The cycle-accurate behavior of a processor as required for guaranteeing timeliness of hard real-time systems is already part of these descriptions. Due to the complexity of current sophisticated processors, these specifications tend to be large and thus cannot be directly used within a timing analyzer.

This thesis summarizes the methodology for the derivation of timing models from hardware description languages that is detailed in [Pis12]. Based on the formal specification of a processor, the proposed derivation process is split into two phases:

- The model preprocessing phase prepares the hardware model for its use within the *aiT* timing analyzer framework and reduces its size by incorporating static available information.
- The subsequent processor state abstraction phase further reduces the size of the model by introducing abstractions that approximate the behavior of the hardware at the cost of precision.

The second phase is to be applied iteratively until the resulting model is applicable for its use within the *aiT* timing analyzer. Thus, the methodology is a consistent advance of the ideas presented in [The04].

To ease timing model derivation, the methodology is embedded into a sound derivation framework. A key part of the derivation framework are static analyses of hardware description languages. This thesis describes the novel static analysis framework that is based on abstract interpretation of hardware description languages.

At the example of VHDL, the two-level semantics that is special to many of these languages is transformed to a more comprehensible one-level semantics that is similar to imperative programming languages. The abstract semantics that is introduced within this thesis eases and supports the deployment of static analyses. The proposed framework allows for analysis of open and closed design specifications and is also not restricted to synchronous circuits.

To show the applicability of the static analysis framework, this work describes three different static analyzers supporting the process of model preprocessing. The employed analyses are not limited to a certain analysis direction, nor limited to flow-sensitive analyses. Also path- and context-sensitive data-flow analyses

can be modeled using the analysis framework. The theory of abstract interpretation underlying the framework further allows for proving the analyses to be correct.

The methods and analyses employed within this thesis have been successfully evaluated on both, academic and industrial processor (or component) specifications. Also the correctness of the analyses that form the backward slicer has been proven. The soundness of analyses results have been proven using interval property checking.

Despite the fact that all analyses presented within this thesis focus on VHDL, the methodology is applicable to other hardware description languages. Also the abstract semantics introduced by the analysis framework is not limited to VHDL itself and is applicable to other hardware description languages requiring only the development of a new compiler frontend for the semantic transformation. Although the presented analysis framework is designed to support and ease the methodology of timing model derivation, it is not limited to this specific domain. The tools developed within this thesis have been provided to the group of Prof. Steger at TU Graz in order to explore the applicability of the framework to other application domains like power estimation.

8.1 Outlook

Future work aims at different research directions, but mainly concerns the improvement of the precision of the different analyses that have been described within this thesis.

Extending the VHDL analysis-support library to support composite data structures is expected to have a major benefit. Supporting composite data structures and operations on them will result in improved results from both, the reset analyzer and the assumption-based model refiner. The ability of evaluating indexed and slice array accesses allows for more initial values being identified by the reset analyzer, and also allows to identify more timing-dead paths and smaller co-domains by the assumption-based model refiner. In the end, supporting composite data structures allows for further restricting the size of a processor specification by only using information that is statically available.

Also the domain of the assumption evaluation analysis is subject to future research. Whereas the current interval domain underlying the analysis allows for restricting the co-domain of an identifier, the introduction of an octagon domain as presented in [Min04, Min05] additionally allows for expressing relations between several identifiers. Relations that can be expressed by this special form of a polyhedral are of the kind $\pm x \pm y \leqslant c$ where x and y are identifiers of basic types and c is a constant. These relations are called octagon packs. A

simple heuristic on which identifiers to be related on each other is to group those that are used within the same expression. Their relation can than be analyzed by a separate analysis. The additional knowledge may then be used in the evaluation of expressions and might further result in more precise results of the assumption-based model refiner.

Also the VHDL compiler that has been developed as a side-product of this thesis is to be improved. Although the synthesizable substandard of VHDL defines standard idioms to be used in order to ease the synthesis process of a specification, wide-spread synthesis tools additionally support languages constructs that improve readability or re-usability. One example are user-defined attributes in VHDL that enhance readability. Especially commercial (closed-source) circuit specifications extensively use these additional features, as the synthesis tool used by the manufacturer is in use over years. In order to enhance the applicability of the compiler to these specifications, the compiler is to be extended to cope with that non-standard idioms and language features. Besides that, new VHDL language standards have been released recently requiring an adaptation of the compiler.

Processor specifications tend to be large and are distributed over many files. Compilation of a design unit requires a preloading of all units on which the current one is dependent on. Currently, the parse ordering for a given specification has to be determined manually. This can also be automated by a prior dependency analysis that provides the parse order to the compiler.

Bibliography

[Abs02] AbsInt Angewandte Informatik GmbH. Prédiction de Temps d'Exécution au Pire Cas. Technical report, 2002.

[Abs05] AbsInt Angewandte Informatik GmbH. *aiSee. Graph Visualization User's Documentation*, 2005.

[AM95] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer Verlag, September 1995.

[AMWH94] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[ARM04] ARM Limited. ARM7TDMI Technical Reference Manual. Reference Manual r4p1, ARM Limited, 2004.

[Ash01] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[BA98] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 237–251, New York, NY, USA, 1998. ACM.

[Bar95] John Barnes. *Ada 95 Rationale: The Language, The Standard Libraries*. Springer Verlag, 1995.

[BBCS77] Mario Roberto Barbacci, Gary E. Barnes, Rick G. Cattell, and Daniel P. Siewiorek. The ISPS computer description language. Technical report, Carnegie-Mellon University, Departments of Computer Science and Electrical Engineering, 1977.

[BBM+07] Jörg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg, Tim Blackmore, and Fabio Bruno. Complete formal verification of TriCore2 and other processors. In Tom Fitzpatrick, editor, *Proceedings of the Design and Verification Conference (DVCon)*, San José, California, USA, February 2007.

[BBN05] Guillem Bernat, Alan Burns, and Martin Newby. Probabilistic timing analysis: An approach using copulas. *Journal on Embedded Computing*, 1:179–194, April 2005.

[BBS96] Luciano Baresi, Cristiana Bolchini, and Donatella Sciuto. Software methodologies for VHDL code static analysis based on flow graphs. In *Proceedings of the conference on European design automation*, EURO-DAC '96/EURO-VHDL '96, pages 406–411, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7:37–61, January 1985.

[Ber06] Christoph Berg. PLRU Cache Domino Effects. In Frank Mueller, editor, *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Bor09] Jörg Bormann. *Vollständige funktionale Verifikation*. PhD thesis, Technische Universität Kaiserslautern, Kaiserslautern, Germany, June 2009.

[Byb12] Tony Bybell. GtkWave. http://gtkwave.sourceforge.net, 2012.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CC91] Patrick Cousot and Radhia Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~rcousot`.).

[CC92b] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CFR+99] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 72–72. Springer Verlag, 1999.

[CFR+02] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing for VHDL. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:125–137, 2002.

[CFS94] Bruce A. Cota, Douglas G. Fritz, and Robert G. Sargent. Control flow graphs as a representation language. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 555–559, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[Cou81] Patrick Cousot. Semantic Foundations of Program Analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou01] Patrick Cousot. Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In R. Wilhelm, editor, *Informatics*

— *10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer Verlag, 2001.

[CP00] Antoine Colin and Isabelle Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal on Real-Time Systems*, 18:249–274, 2000. 10.1023/A:1008149332687.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 469–485, London, UK, 2001. Springer Verlag.

[FKL+99] Christian Ferdinand, Daniel Kästner, Marc Langenbach, Florian Martin, Michael Schmidt, Jörn Schneider, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Run-Time Guarantees for Real-Time Systems - The USES Approach. In Kurt Beiersdörfer, Gregor Engels, and Wilhelm Schäfer, editors, *Informatik '99. Informatik überwindet Grenzen. 29. Jahrestagung der Gesellschaft für Informatik on October 5-9, 1999 at Paderborn, Germany*, pages 410–419. GI, Springer Verlag, 1999.

[FMC+07] Christian Ferdinand, Florian Martin, Christoph Cullmann, Marc Schlickling, Ingmar Stein, Stephan Thesing, and Reinhold Heckmann. New Developments in WCET Analysis. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation. Theory and Practice. Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 12–52. Springer Verlag, 2007.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 35(2-3):163–189, 1999.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[Fre01] Freescale Semiconductor, Inc. MPC750 RISC Microprocessor Family. Reference manual, Freescale Semiconductor, Inc., December 2001. Rev. 1.

[Fre02] Freescale Semiconductor, Inc. MPC603e RISC Microprocessor User's Manual. Reference manual, Freescale Semiconductor, Inc., April 2002. Rev. 3.

[Fre04] Freescale Semiconductor, Inc. PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors. Reference manual, Freescale Semiconductor, Inc., 2004. Rev. 0.1.

[Fre05a] Freescale Semiconductor, Inc. MPC7450 RISC Microprocessor Family. Reference manual, Freescale Semiconductor, Inc., January 2005. Rev. 5.

[Fre05b] Freescale Semiconductor, Inc. Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture. Reference manual, Freescale Semiconductor, Inc., September 2005. Rev. 3.

[Fre06] Freescale Semiconductor, Inc. sim_G4plus v1.1 Cycle-Accurate Simulator User's Guide. Reference manual, Freescale Semiconductor, Inc., May 2006. Rev. 2.5.

[Fre08] Freescale Semiconductor, Inc. QorIQTM P4080 Communications Processor Product Brief. Product brief, Freescale Semiconductor, Inc., September 2008. Rev. 1.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 28:237–247, June 1993.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Journal of Real-Time Systems*, 17(2-3):131–181, 1999.

[Gai] Gaisler Research. http://www.gaisler.com.

[Gai05] Gaisler Research. Leon2 Processor User's Manual. Reference Manual 1.0.30, Gaisler Research, 2005.

[Gai08] Gaisler Research. GRLIB IP Core User's Manual. Reference Manual 1.0.18, Gaisler Research, 2008.

[Geb10] Gernot Gebhard. Timing Anomalies Reloaded. In Björn Lisper, editor, *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[Gin07] Tristan Gingold. *GHDL. A VHDL compiler*, 2007.

[GK83] Dan D. Gajski and Robert Henry Kuhn. New VLSI Tools. *Computer*, 16:11–14, December 1983.

[GRW11] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. A Template for Predictability Definitions with Supporting Evidence. In *Proceedings of the Workshop on Predictability and Performance in Embedded Systems*, 2011.

[Hal95] Tom R. Halfhill. The Truth Behind the Pentium Bug. *Byte*, 1995.

[Haw03] Tom Hawkins. Declarative programming language simplifies hardware design. *EE Times Design*, 2003.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[HLS00] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-Case Execution Time Analysis for digital signal processors. In *Proceedings of the EUSIPCO 2000 Conference (European Signal Processing Conference)*, 2000.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. In *Proceedings of the IEEE*, volume 91, pages 1038–1054, 2003.

[Hor97] Joachim Horch. Entwurf eines RISC-Prozessors in der Hardwarebeschreibungssprache VHDL. Studienarbeit, Technische Universität Darmstadt, Darmstadt, Germany, June 1997.

[HPG03] John L. Hennessy, David A. Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[HRB88] Susan Beth Horwitz, Thomas William Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 23:35–46, June 1988.

[Hym03] Charles Hymans. Design and Implementation of an Abstract Interpreter for VHDL. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 263–269. Springer Verlag, 2003.

[Hym04] Charles Hymans. *Vérification de composants VHDL par interprétation abstraite*. PhD thesis, École Polytechnique, 2004.

[IEE87] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076 1987 VHDL Language Reference Manual*, 1987.

[IEE99] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard P1076.6 1999 VHDL Register Transfer Level Synthesis*, 1999.

[Inf02] Infineon Technologies. TriCore 1.3 32-bit Unified Processor Core Architecture Overview Handbook. Reference Manual 1.3.3, Infineon Technologies, May 2002.

[ISO98] ISO/IS 15408 Final Committee, International Standards Organisation. *Common Criteria for information technology security (CC)*, 1998. Version 2.0 Draft.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[KKP+81] David J. Kuck, Robert Henry Kuhn, David A. Padua, Bruce Leasure, and Michael Joseph Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[KLFP02] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/ Simulink Models. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 31–40, Washington, DC, USA, 2002. IEEE Computer Society.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language, Second Edition, ANSI C*. Bell Telephone Laboratories, Inc., 1988.

[KSP+12a] Daniel Kästner, Marc Schlickling, Markus Pister, Christoph Cullmann, Gernot Gebhard, Christian Ferdinand, and Reinhold Heckmann. Timing Predictability of Multi-Core Processors. In *Proceedings of the Embedded World 2012 Conference*, 2012.

[KSP+12b] Daniel Kästner, Marc Schlickling, Markus Pister, Christoph Cullmann, Gernot Gebhard, Reinhold Heckmann, and Christian Ferdinand. Meeting real-time requirements with multi-core processors. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 117–131. Springer Verlag, 2012.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. 10.1007/BF00290339.

[KWH+08] Daniel Kästner, Reinhard Wilhelm, Reinhold Heckmann, Marc Schlickling, Markus Pister, Marek Jersak, Kai Richter, and Christian Ferdinand. Timing Validation of Automotive Software. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA) 2008*, volume 17 of *Communications in Computer and Information Science*, pages 93–107. Springer Verlag, November 2008.

[Kä00] Daniel Kästner. *Retargetable Postpass Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, 2000.

[Lan98] Marc Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Saarland University, 1998.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, pages 254–263, Washington, DC, USA, 1996. IEEE Computer Society.

[Low06] Geoff Lowney. Why Intel is designing multi-core processors. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 113–113, New York, NY, USA, 2006. ACM.

[LS99a] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 12–21, December 1999.

[LS99b] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, 17:183–207, 1999.

[LT30] Jan Lukasiewicz and Alfred Tarski. Untersuchungen über den Aussagenkalkül. *Comptes Rendus Séances Société des Sciences et Lettres Varsovie*, 23:30–50, 1930.

[LTH+10] Xun Li, Mohit Tiwari, Ben Hardekopf, Timothy Sherwood, and Frederic T. Chong. Secure information flow analysis for hardware design: Using the right abstraction for the job. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, page 8. ACM, 2010.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, 2002.

[LWS+10] Sascha Loitz, Markus Wedler, Dominik Stoffel, Christian Brehm, and Wolfgang Kunz. 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 1–8, Southampton, United Kingdom, September 2010. IET, ECSI, Electronic Chips & Systems design Initiative.

[Mak07] Mohamed Abdel Maksoud. Generating Code from Abstracted VHDL Models. Master's thesis, Saarland University, 2007.

[Mar95] Florian Martin. Entwurf und Implementierung eines Generators für Datenflußanalysatoren. Master's thesis, Saarland University, 1995.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Saarland University, 1999.

[Mar05] Peter Marwedel. *Embedded System Design*. Springer Verlag, Berlin, 2nd edition, November 2005.

[Mar08] Michael R. Marty. *Cache coherence techniques for multicore processors*. PhD thesis, University of Wisconsin, Madison, Wisconsin, USA, 2008.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC '98), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS) on March 28-April 4, 1998 at Lisboa, Portugal*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer Verlag, 1998.

[Min04] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. *Programming Languages and Systems*, pages 3–17, 2004.

[Min05] Antoine Miné. *Weakly relational numerical abstract domains*. PhD thesis, École Polytechnique, Paris, France, 2005.

[MKC66] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Prentice-Hall Englewood Cliffs, NJ, 1966.

[MP00] Silvia M. Müller and Wolfgang Paul. *Computer Architecture: Complexity and Correctness*. Springer Verlag, Secaucus, NJ, USA, 2000.

[MPS09] Mohamed Abdel Maksoud, Markus Pister, and Marc Schlickling. An Abstraction-Aware Compiler for VHDL Models. In *Proceedings of the International Conference on Computer Engineering and Systems (ICCES '09)*, pages 3–9. IEEE Computer Society, December 2009.

[Nie82] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18(3):265–287, 1982.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[One06] OneSpin Solutions GmbH. OneSpin 360 MV. http://www.onespin-solutions.com/download.php, May 2006.

[OO84] Karl J. Ottenstein and Linda Ottenstein. The program dependence graph in a software development environment. *Proceedings of the ACM SIGPLAN '84 Conference on Programming Language Design and Implementation*, 19(5):177–184, 1984.

[PCI02] PCI Special Interest Group, 5440 SW Westgate Drive Suite 217 Portland, Oregon 97221. *PCI Local Bus Specification*, March 2002. Standard Rev. 2.3.

[Pis12] Markus Pister. *Timing Model Derivation – Pipeline Analyzer Generation from Hardware Description Languages*. PhD thesis, Saarland University, 2012.

[PN98] Peter Puschner and Roman Nossal. Testing the Results of Static Worst-Case Execution-Time Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 134–143. IEEE Computer Society Press, 1998.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Saarland University, November 2008.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007.

[RKK04] Sethu Ramesh, Aditya Rajeev Kulkarni, and V. Kamat. Slicing tools for synchronous reactive programs. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 217–220, New York, NY, USA, 2004. ACM.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[Sch98] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation*, 10(3):237–271, 1998.

[Sch03] Jörn Schneider. Combined schedulability and WCET analysis for real-time operating systems. Master's thesis, Saarland University, 2003.

[Sch05] Marc Schlickling. Generisches Slicing auf Maschinencode. Master's thesis, Saarland University, 2005.

[Sch09] Marc Schlickling. Safe Approximation of Access-Latencies to Asynchronous Memory Regions. Technical report, AbsInt Angewandte Informatik GmbH, 2009.

[Sha89] Alan Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15:875–889, 1989.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Master's thesis, Saarland University, 1997.

[SP81] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SP07] Marc Schlickling and Markus Pister. A Framework for Static Analysis of VHDL Code. In Christine Rochange, editor, *Proceedings of 7th International Workshop on Worst-case Execution Time (WCET) Analysis*, July 2007.

[SP09] Marc Schlickling and Markus Pister. Worst Case Execution Time Analyzer for PowerPC MPC7448 – Performance Counter Validation Report. Technical report, AbsInt Angewandte Informatik GmbH, October 2009. ai20090930.

[SP10] Marc Schlickling and Markus Pister. Semi-Automatic Derivation of Timing Models for WCET Analysis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 67–76. ACM, April 2010.

[SPA91] SPARC International, Inc. The SPARC Architecture Manual – Version 8. Reference manual, SPARC International, Inc., 1991. Revision SAV080SI9308.

[Spe92] Special Commitee 167. Software Considerations in Airborne Systems and Equipment Certification (DO-178B/ED-12B). Standard, Radio Technical Commission for Aeronautics, 1992.

[SPH+07] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Guillaume Borios, Victor Jégu, and Reinhold Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In Reinhard Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Ste94] Mario Stefanoni. Static analysis for VHDL model evaluation. In *Proceedings of the conference on European design automation*, EURO-DAC '94, pages 586–591, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Ste06] Ingmar Stein. Analyse von Pfadausschlüssen auf Maschinencode. Diplomarbeit, Universität des Saarlandes, Saarbrücken, February 2006.

[Ste10] Ingmar Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Saarland University, 2010.

[SWH12] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design – Analysis and Transformation*. Springer Verlag, 1st edition, July 2012. ISBN 978-3-642-17547-3.

[Syn] Synopsys, Inc. http://www.synopsys.com.

[The00] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing and Applications Symposium (RTCSA '00) on December 12-14, 2000 at Cheju Island, South Korea*, pages 23–30. IEEE Computer Society, 2000.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[The06] Stephan Thesing. Modeling a system controller for timing analysis. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 292–300, New York, NY, USA, 2006. ACM.

[Tho65] James E. Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, AFIPS '64 (Fall, part II), pages 33–40, New York, NY, USA, 1965. ACM.

[Tip95] Frank Tip. A Survey of Program Slicing Techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1995.

[TM02] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language*. Springer Verlag, 2002.

[TMAL98] Stephan Thesing, Florian Martin, Martin Alt, and Oliver Lauer. PAG User's Manual. Technical report, Saarland University, 1998. Version 1.0.

[TNN05] Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson. Information flow analysis for VHDL. *Parallel Computing Technologies*, pages 79–98, 2005.

[Tol06] Terkel K. Tolstrup. *Language-based Security for VHDL*. PhD thesis, Technical University of Denmark, Department of Informatics and Mathematical Modeling, Language-Based Technology, 2006.

[Tom67] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003) at San Francisco, CA, USA, 22-25 June 2003*, pages 625–632. IEEE Computer Society, 2003.

[TUD] TU Darmstadt http://www.rs.tu-darmstadt.de/downloads/docu/dlxdocu/superscalardlx.html.

[USB+10] Joakim Urdahl, Dominik Stoffel, Jörg Bormann, Markus Wedler, and Wolfgang Kunz. Path predicate abstraction by complete interval property checking. In Natasha Sharygina, editor, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 10, pages 207–215, Lugano, Switzerland, October 2010. IEEE.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 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, 2008.

[Weg11] Simon Wegener. Improving Static Analysis of Loops. Master's thesis, Saarland University, June 2011.

[Wei79] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[Wei81] Mark Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[Wei82] Mark Weiser. Programmers Use Slicing When Debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[WEY01] Fabian Wolf, Rolf Ernst, and Wei Ye. Path Clustering in Software Timing Analysis. *IEEE Transactions on VLSI Systems*, 9(6), December 2001.

[WFC+09] Reinhard Wilhelm, Christian Ferdinand, Christoph Cullmann, Daniel Grund, Jan Reineke, and Benoit Triquet. Designing Predictable Multicore Architectures for Avionics and Automotive Systems. In *Workshop on Reconciling Performance with Predictability (RePP)*, 2009.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 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.

[Wil12] Stephan Wilhelm. *Symbolic Representations in WCET Analysis*. PhD thesis, Saarland University, 2012.

[WKE02] Fabian Wolf, Judita Kruse, and Rolf Ernst. Timing and Power Measurement in Static Software Analysis. *Microelectronics Journal, Special Issue on Design, Modeling and Simulation in Microelectronics and MEMS*, 6(2):91–100, January 2002.

[WKRP05] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Worst-Case Execution Time Analysis. In *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 7–10, Washington, DC, USA, 2005. IEEE Computer Society.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison-Wesley, 1995. Second Printing.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.

[WW08] Reinhard Wilhelm and Björn Wachter. Abstract Interpretation with Applications to Timing Validation. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 22–36. Springer Verlag, 2008.

Index

T

V

W