Language-based Security for VHDL

Terkel K. Tolstrup

Kongens Lyngby 2006
IMM-PHD-2006-174

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

The need for reliable performance of computerised systems is well-known, yet the security verification of hardware systems is often none-existing or applied in an ad-hoc manner. To overcome this issue, standards such as the *Common Criteria* has been successful in listing points of attention for a thorough investigation of a system. Hence in this thesis it is investigated how language-based security techniques can be adapted and applied to hardware specifications. The focus of the investigation is on the information flow security that is required by the standard. Information flow security provides a strong notion of end-to-end security in computing systems. However sometimes the policies for information flow security are limited in their expressive power, and this complicates the matter of specifying policies even for simple systems. These limitations often become apparent in contexts where confidential information is released under specific conditions.

This thesis presents a novel policy language for expressing permissible information flow using expressive constraints on the execution traces for programs. Based on the policy language a security property is proposed and shown to be a generalised intransitive non-interference condition. The security property is defined on a fragment of the hardware description language VHDL that is suitable for implementing the *Advanced Encryption Standard* algorithm.

The means to verify the property is provided in the terms of a static information flow analysis. The goal of the analysis is to identify the entire information flow through the VHDL program. The result of the analysis is presented as a non-transitive directed graph that connects those nodes (representing either variables or signals) where an information flow might occur. The approach is compared to traditional approaches and shown to allow for a greater precision.

Practical implementations of embedded systems are vulnerable to side channel attacks. In particular timing channels have had a great impact on the security

verification of algorithms for cryptography. In order to address this another security property stating the absence of timing channels is presented. Again, verification is provided by a static analysis, that identifies the timing behaviour of a program. This analysis consists of a type system that identifies the delay between when a value enters the system and when it leaves it again. Programs accepted by our analysis are shown to have no timing channels.

Resumé

Vigtigheden af pålidelig opførsel i computersystemer er velkendt, men stadigvæk bliver sikkerhedsverifikation af hardware-systemer ofte ikke udført, eller verifikationen bliver udført på en ustruktureret facon. For at overkomme dette problem, har standarder som *Common Criteria* været succesfulde ved at fremhæve vigtige punkter for en fuldstændig undersøgelse af et system. Derfor vil denne afhandling undersøge, hvorledes teknikker inden for sprog-baseret sikkerhed kan tilpasses og anvendes på hardware specifikationer. Fokus i denne undersøgelse vil være på information flow sikkerhed som det kræves af standarden. Information flow sikkerhed giver en stærk opfattelse af ende-til-ende sikkerhed i et computersystem. Imidlertid sker det, at politiker for information flow sikkerhed kan være for begrænsede i deres udtrykskraft, og derfor kan de komplicere opgaven omkring at angive en politik for endda simple systemer. Disse begrænsninger bliver ofte fremhævet af systemer, hvor hemmelige oplysninger bliver frigivet ved opfyldelse af klare krav.

Denne afhandling præsenterer et nyt sprog for politiker, der muliggør at udtrykke information flow, der bliver tilladt under angivne krav til eksekveringssekvensen ved afvikling af programmet. Baseret på sproget for politikerne foreslås en sikkerhedsegenskab, som vises at generaliserer intransitiv non-interference egenskaben. Sikkerhedsegenskaben defineres på et fragment af hardware beskrivelses sproget VHDL, der er tilstrækkeligt til at implementere en *Advanced Encryption Standard* algoritme.

En information flow analyse defineres efterfølgende. Målet med analysen er at identificere, hvorledes information flyder gennem hele VHDL programmet. Resultatet af analysen er en intransitiv orienteret graf, der forbinder de knuder (repræsenterende enten variable eller signaler), hvor informationen eventuelt kan flyde. Denne tilgang sammenlignes med traditionelle metoder og vises at muliggøre en analyse med større præcision.

Virkelige implementationer af indlejrede systemer er sårbare over for angreb gennem side channels. Specielt har side channels stor indflydelse på sikkerhedsverifikationen af kryptografiskealgoritmer. Derfor foreslås en sikkerhedsegenskab, der angiver, hvornår et system er fri for timing channels. En statisk analyse til at identificere den tidsmæssige opførsel af et program bliver også præsenteret. Analysen består af et type system, der identificerer forsinkelser, mellem at en værdi kommer ind i systemet, og indtil den forlader systemer igen. Det bevises at programmer, der accepteres af analysen, ikke indeholder timing channels.

Preface

Whether you take the doughnut hole as a blank space
or as an entity unto itself is a purely metaphysical question
and does not affect the taste of the doughnut one bit
— Haruki Murakami

This thesis was prepared at the department for Informatics and Mathematical Modelling, Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering. The Ph.D. study has been carried out under the supervision of Professor Hanne Riis Nielson and Professor Flemming Nielson.

Acknowledgements

I would like to thank my supervisors Hanne Riis Nielson and Flemming Nielson for continous support and guidance. Furthermore I would like to thank them as well as the rest of the LBT Group, Han Gao, Christoffer Rosenkilde Nielsen, Henrik Pilegaard, Christian W. Probst and Ye Zhang for providing an inspiring and motivating working environment. Further gratitude must go to Henrik Pilegaard for enduring countless discussions, arguments and three years of sharing an office with me. I would to thank René Rydhof Hansen for numerous discussions and working together with me on [TNH06], likewise thanks go to Sebatian Nanz for comments and suggestions.

For my visit at the Universität des Saarlandes, Saarbrücken, Germany, I would like to thank Reinhard Wilhelm for providing me with everything I needed during the three month I visited, and for always discussing, commenting and motivating my work on timing channels. Thanks go to Stephan Thesing for

discussions and suggestions and to Jörg Bauer for comments, proof reading and never allowing me a boring weekend.

I would like to thank Andrei Sabelfeld for allowing me to visit him at Chalmers during September 2005, for helping me with numerous issues - work related and otherwise - and for being a inspiration, even after my return to Lyngby. Thanks also go to David Sands, Aslan Askarov and Daniel Hedin for welcoming me to the ProSec group, and for having daily discussions that have helped me shape an understanding and intuition for bisimulations as security properties.

Finally I would like to thank my family, friends and in particular my wife for unlimited patience, support and love, without which I could not have finished this thesis.

Lyngby, October 2006

Terkel K. Tolstrup

Contents

Chapter 1

Introduction

There is something to be learned from a rainstorm. When meeting with a sudden shower, you try not to get wet and run quickly along the road. But doing such things as passing under the eaves of houses, you still get wet. When you are resolved from the beginning, you will not be perplexed, though you still get the same soaking. This understanding extends to everything.
— Yamamoto Tsunetomo

Every individual comes into contact with over a hundred embedded computer systems every day [Mat04]. Many exist in our homes and many more operate the commonplace items in the world around us. They are now common in households through cameras, televisions, video players, refrigerators, lawn sprinkler systems, and many other items. They are in the world around us controlling our street lighting, door openers, intruder alert systems, product theft security, speed cameras, and much more. Furthermore several embedded systems perform safety critical operations where malfunction might cause the loss of human lives, for example in pacemakers, steering aid for cars and fly-by-wire systems for airplanes.

Every day additional functionality are added to existing systems and new systems are integrated into more devices. In fact the growth in the complexity of embedded systems is said to be exponential, commonly referred to as Moore's law, that state that the number of transistors in a integrated circuit is doubled every 18 months. At the same time the size and price of the circuit is halved. According to [MED04] the electronics industry totalled a $800 billion market in 2003. The main threat identified in [SEM03] to the growth of this market is the

cost connected to the design of systems. The fact that about 60% to 70% of the total development time of a circuit is spent on simulation [MED02] makes it of great concern.

The concept of security for these systems is traditionally very low because the designer has always been able to depend on the physical security of an enclosed box. However, as more of the "boxes" are connected together networks come into being and opportunities for access and malfunction, whether through poor design, unforeseen circumstances, or foul play, become possible. Their security is at risk in many cases, much of it due to "security through obscurity". The simulation phase of the development cycle is rarely concerned with verifying security properties. Instead the focus is on validating the behaviour of the system on "standard" well-formed input, to make sure that the required functionality is implemented. However the security of systems is often compromised because of non-standard input, that might be the result of unforeseen usage of the system or hostile behaviour.

The need for reliable performance of computerised systems is well-known, yet the ad-hoc design of systems that become increasingly complex makes documenting and verifying security properties hard. Therefore to document the verification of a number of well-known security properties (e.g. authenticity, confidentiality and integrity) standards and frameworks have been proposed. In this thesis the focus will be on the *Common Criteria* [CC98] standard proposed and maintained by the *International Standard Organization* (ISO). The Common Criteria supersedes the *Trusted Computer System Evaluation Criteria* (TCSEC) [DOD85], the *Information Technology Security Evaluation Criteria* (ITSEC) [ITS91] and the *Canadian Trusted Computer Product Evaluation Criteria* (CTCPEC). The Common Criteria standard greatly benefits from being an internationally accepted standard, e.g. the United Nations have chosen the standard for the documentation of the security of systems used by the military of member countries.

The Common Criteria standard is a collection of objectives describing security properties for all kinds of systems, descriptions of assurances to match each objective and assurance levels that define the kind of methods used to provide the assurances. When documenting a system's security the designer needs to identify a set of reasonable objectives with respect to the setting and usage of the system. Based on these objectives assurances need to be given and documented. Each assurance is given within a level specifying the manner of the verification. Assurances given in the lower levels need to document details on the requirements for the system and the tests performed (for example by simulation) to ensure that the system behaves accordingly. Assurances in the higher levels require that formal methods are applied to verify the system.

In contrast to simulation, formal methods can be exhaustive, even for complex systems with infinitely many states, as they deal with proving the correctness of a system. One could prove the correctness of a system in the same fashion as theorems are proved. This would demand much time and effort. Hence assistance from automated *theorem provers* could aid the designers or evaluators. Still, the amount of expertise and knowledge required often result in only core functions of real-life systems being verified.

Another option is to validate the behaviour of the system by performing an exhaustive search on all possible input, *model checking* does so. However this approach might suffer from the complexity of systems, commonly referred to as the state explosion problem. Therefore many approaches based on model checking rely on techniques for reducing the state space, heuristics and randomisation. These techniques have shown to be powerful in finding flaws, but in general they are not suitable for proving the absence of flaws. However this would be a main goal for assuring a Common Criteria objective.

Therefore important properties of the formal methods assisting in the verification of Common Criteria assurances would have to be

Automatic: Demands a minimum of effort from the designer

Correctness: Guarantees that verification implies assurance

Efficient: Handles the complexity of realistic systems

Exhaustiveness: Must cover all executions on all input

To fulfill these properties we use *static program analysis* [NNH99]. Static program analysis benefits from being developed within well-founded frameworks with clear guidelines for their implementation, resulting in tools that are automatic and exhaustive. The limitation of static program analysis is that one is forced to sacrifice precision in return for efficiency. Precision is lost due to using abstractions of the analysed system. Still analyses can be designed to perform well on real-life systems, being both efficient and sufficiently precise. To achieve the correctness of the method it is therefore important that all abstractions introduced in the analysis still allow us to verify programs that fulfill the desired objectives.

1.1 Main thesis

The main thesis of this report is therefore to show that static program analyses can be designed and implemented, 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.

For this purpose we consider the objectives *Information flow control policy* (FDP_IFC) and *Information flow control functions* (FDP_IFF) from the Common Criteria standard, that deal with the confidentiality of information manipulated by the system. In this thesis the flow of information through a system is viewed as a directed graph and the definition of information flow policies is based on this view. This is contrary to much of the literature available in the field on information flow security (see [SM03a] for references). The traditional view is that information can be divided into hierarchical levels. This matches the intuition of military systems where information is labelled with security annotations such as *Top Secret*, *Confidential* and *Unclassified*. In the more general setting of multiple systems participating in collaborative transactions (e.g. online shopping, tax auditing and service oriented computing) a natural underlying hierarchical order is not always apparent. Hence in this thesis we will investigate the usage of graph-based policies for information flow security.

The flow of information is verified by a set of analyses dealing with different facets; first an analysis verifying the flow of information due to *control-* and *data-flow* during execution of the program and second an analysis that deals with covert channels introduced by differences in the timing behaviour of the system. Together, these analyses will give a large part of the assurance for a system needed when making the Common Criteria evaluation document.

Another main goal is that the designed analyses validate specifications of integrated circuits directly, rather than analysing on a process calculi, thereby forcing the developers to translate the system, and possibly introducing flaws in the process. The presented analyses are all specified on a fragment of VHSIC[1] Hardware Description Language (VHDL). The fragment will be referred to as CoreVHDL and is expressive enough to deal with specifications of integrated circuits not designed for verification purposes.

[1] Very High Speed Integrated Circuit

1.2 Overview

In the following chapters the investigation of information flow security in the hardware setting is investigated. First the semantical model for CoreVHDL is defined in Chapter 2, which allows us to formalise the semantical meaning of the *Locality-based* security policies in Chapter 3. Chapter 3 also proposes a *generalised non-interference* property, named *History-based Release*, that permits information flows based on the execution history of the involved principals. Chapter 4 presents an *Information Flow* analysis that allows for the automatic verification of hardware specifications. The analysis is extended in Chapter 5 by a *Reaching Definitions* analysis, which tracks the *control flow* in the program. In Chapter 6 timing leaks in integrated circuits are investigated. A security property for the absence of timing leaks in a CoreVHDL specification is proposed and an automated analysis for verifying programs is presented.

During the investigations presented is this thesis much of the work has been inspired by a real communications system sought to be verified within the Common Criteria. The communications system was developed by the Danish company Maersk Data Defence, as a contractor for the military sector. The presented analyses were successful in verifying the available parts of the system that were previously verified by a *paper-and-pencil* approach. Unfortunately due to contractual obligations the system can not be discussed in this thesis. Instead we will continuously consider a reference implementation of the *Advanced Encryption Standard* (AES) [WBRF00]. This implementation will be a running example aiding in illustrating the techniques presented in the chapters.

1.3 Contributions

The main contributions of this thesis are:

Chapter 2 presents a detailed semantical model of VHDL. In particular, practical issues of the semantics for VHDL is treated formally in accordance with *state-or-the-art* simulators. These issues are previously neglected or formalised in an incorrect manner, i.e. with respect to the VHDL standard [IEE01]. Parts of this contribution has previously been published in [TNN05] and [TN06].

Chapter 3 investigates the usage of graph based security policies for information flow security. Furthermore the policies are extended with a notion of localities and execution history that results in a generalised non-

interference property, History-based Release. Parts of the graph based policies for information flow was originally published in [TNN05], while parts of the locality-based policies and the history-based release property for the process calculi *Klaim* [BBN+03] was published in [TNH06].

Chapter 4 presents the automated analysis of a VHDL subset for information flow security. Especially the treatment of synchronisation in VHDL leads to novel results. Furthermore the automated analysis for programs that comply with the History-based Release property. Parts of these contributions have previously been published in [TNN05, TNH06].

Chapter 5 motivates the need for *control flow* sensitive analyses for information flow security. The chapter presents a Reaching Definitions analysis, and discusses previously published approaches to static analysers for VHDL. The Reaching Definitions analysis was first published in [TNN05].

Chapter 6 considers timing leaks in hardware specifications. A novel property for the absence of timing leaks is presented together with an automated analysis for verification of systems. The property is the first of its kind that supports the composition with generalised non-interference properties. Parts of the work was first published in [TN06].

Furthermore nontrivial extensions and additions have been necessary to complete the work presented in this thesis. In particular the work presented in this thesis removes several simplifications made on the hardware setting in the published papers.

Chapter 2

VHDL

The difference between virtuality and life is very simple. In a construct you know everything is being run by an all-powerful machine. Reality doesn't offer this assurance, so it's very easy to develop the mistaken impression that you're in control.
— Richard K. Morgan

Very High Speed Integrated Circuit Hardware Description Language, commonly referred to as VHSIC Hardware Description Language or VHDL, is one of the most widely used hardware description languages. Development was initiated in 1981 under a research programme initiated by the US Department of Defense to accommodate the rising need for documenting integrated circuits. Hence early on VHDL focussed on describing the behaviour of hardware in a manner that allowed specifications to be decomposed hierarchically while providing a well-defined interface for composing elements. With the details available in a VHDL description the possibility of simulating specifications became apparent. This possibility of simulating hardware descriptions before synthesising the circuit had great impact, as it severely cut the development cost and time, and therefore participated in shaping a new hardware design methodology. Finally, tools for automatic synthesis of descriptions were developed. During the mid-1980s all rights to the language definition were given to IEEE, who standardised the language as IEEE standard 1076-1987 [IEE87].

A problem not solved in the original standard was that of an electrical signal's driving strength. This is a result of not only having the pure boolean values

true and *false* in digital hardware design, but also having the possibility of e.g. turning off a signal in the synthesised circuit. The IEEE standard 1164-1993 [IEE93a] introduced the *multi-valued logic* `std_logic`, not only including values for driving strengths but also an *unknown* value (primarily for simulation purposes). The VHDL standard was revised accordingly [IEE93b] while taking other issues into account as well, such as making the syntax more consistent and introducing more logical operators. Like all other IEEE standards the VHDL standard is revised every five years to ensure an ongoing relevance to the industry. The most recent revision is IEEE standard 1164-2001 [IEE01].

In this thesis we will focus on a subset of the latest revision. The subset is identified around the main concepts in *register transfer level* (RTL) VHDL, the interpretation of which common language simulators and synthesisers should agree on [IEC05]. The subset will be referred to as CoreVHDL and is presented in Section 2.1. Furthermore we will discuss some of the language constructs left out, and their pre-compilation into the CoreVHDL subset.

In order to reason about VHDL descriptions we first need to define their semantics. The semantics will be a cornerstone in the techniques presented later. Hence defining the semantics is the main goal of this chapter. The rest of the chapter will be structured as follows; the semantics is presented in Section 2.2. Section 2.4 presents the running example to be used throughout the thesis, which is based on an implementation of the *Advanced Encryption Standard*. Finally we discuss related work in Section 2.5.

2.1 CoreVHDL

Prior to the introduction of VHDL most hardware was documented in an ad-hoc manner. Hence with the introduction of VHDL it was important to have a description language with a well-defined syntax and semantics. The original syntax of VHDL was essentially a subset of the Ada programming language, where special constructs were added to handle the parallelism inherent in hardware design. A central issue in hardware design is the behaviour of the underlying circuit, which the hardware description should document with enough precision to allow simulation and automatic synthesis. However, since VHDL is primarily developed for documenting hardware, only a subset of the language can be simulated and synthesised automatically. While many simulators often support larger subsets their simulations are only required to agree on the *Register Transfer Level* subset [IEC05]. Thus our investigation is limited to a subset of RTL VHDL to give some assurance of compliance with tools for synthesis.

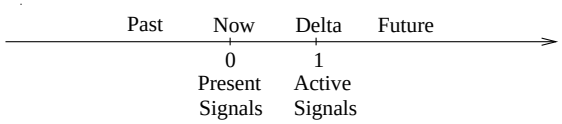


Figure 2.1: The representation of abstract time in the signal store.

A VHDL description may contain a mixture of *behavioural* and *structural* specifications. A *behavioural* specification describes the functionality in much the same way as a program in a traditional programming language. A *structural* specification describes how structures are connected into a more complex architecture. For a specification to be *synthesizable* the behaviour of the program must be completely specified, i.e. all parts of the VHDL specification must be given by a *behavioural* specification or the specifications used in the *structural* specification are all present either as library structures or constructed of smaller *behavioural* specifications.

CoreVHDL is a fragment of RTL VHDL that concentrates on the behavioural specification of models. A program in CoreVHDL consists of *entities* and *architectures*, uniquely identified by indices $i_e, i_a \in Id$. An entity describes a component's interface to the environment. An architecture comprises the behavioural or structural specification of an entity.

An entity specifies a set of signals referred to as ports ($prt \in Prt$), each port is represented by a signal ($s \in Sig$) used for reference in the specification of the architecture; furthermore a notion of the intended usage of the signal is specified by the keywords `in` and `out` determining if the signals value can be altered or read by the environment, and the type of the signal's value.

An architecture model is specified by a family of concurrent statements ($css \in Css$) running in parallel; mainly processes, here the index $i_p \in Id$ is a unique identifier in a finite set of process identifiers ($I_p \subseteq_{\text{fin}} Id$). Each process has a statement ($ss \in Stmt$) as body and may use logical values ($m \in LVal$), local variables ($x \in Var$) as well as signals ($s \in Sig$, $S \subseteq_{\text{fin}} Sig$). When accessing variables and signals we always refer to their present value and when we assign to variables it is always the present value that is modified. However, when assigning to a signal its present value is *not modified*, rather its so-called active value is modified; this representation of signal's values, as illustrated in Figure 2.1, is used to take care of the physical aspect of propagating an electrical current through a system, the time consumed by the propagation is usually called a *delta-delay*. The wait statements are synchronisation points, where the active values of signals are used to determine the new present values that will

$pgm \in Pgm$		programs
pgm	::=	$ent \mid arch \mid pgm_1\ pgm_2$
$ent \in Ent$		entities
ent	::=	`entity` i_e `is port(`prt`); end` i_e;
$prt \in Prt$		ports
prt	::=	s : `in` $type \mid s$: `out` $type \mid prt_1; prt_2$
$type \in Type$		types
$type$	::=	`std_logic`
$arch \in Arch$		architectures
$arch$	::=	`architecture` i_a `of` i_e `is begin` css; `end` i_a;
$css \in Css$		concurrent statements
css	::=	i_p : `process` $decl$; `begin` ss; `end process` i_p
		$\mid i_b$: `block` $decl$; `begin` css; `end block` i_b
		$\mid css_1 \parallel css_2$
$decl \in Decl$		declarations
$decl$	::=	`variable` x : $type := e$
		$\mid$ `signal` s : $type := e$
		$\mid decl_1; decl_2$
$ss \in Stmt$		statements
ss	::=	`null` $\mid ss_1; ss_2 \mid$ `if` e `then` ss_1 `else` ss_2
		$\mid x := e \mid s <= e$
		$\mid$ `wait on` S `until` e
$e \in Exp$		expressions
e	::=	$m \mid op^u\ e \mid e_1\ op^b\ e_2 \mid x \mid s$

Figure 2.2: The subset CoreVHDL of VHDL.

be common to all processes. This will be further discussed in Section 2.2.

Concurrent statements could also be block statements that allow local signal declarations for the use of internal communication between processes declared within the block. The index $i_b \in Id$ is a unique identifier in a finite set of block identifiers ($I_b \subseteq_{fin} Id$). The scope of the local signals declared by a block definition is the concurrent statements specified inside the block.

Since VHDL describe digital hardware we are concerned with the details of electrical signals, and it is therefore necessary to include types to represent digitally encoded values. We consider logical values ($LVal$) of the standard logic type *std_logic*, that includes traditional boolean values as well as values for electrical properties.

The formal syntax is given in Figure 2.2. In VHDL it is allowed to omit components of wait statements. Writing $FS(e)$ for the free signals in e, the effect of '`on` $FS(e)$' may be obtained by omitting the '`on` S' component, and the effect of '`until true`' may be obtained by omitting the '`until` e' component. (In other words, the default values of S and e are $FS(e)$ and `true`, respectively.) Semantically, S is the set of signals waited on, i.e. at least one of the signals of S must have a new active value, and e is a condition on the resulting present values that must be fulfilled, in order to leave the wait statement.

The syntax does not include any loops. In general the loops in VHDL can not be synthesised, as the synthesising tool needs to produce timing guarantees on finishing the loop. Instead, looping behaviour is obtained using processes and iterator signals. However, loops aid in making high-level specifications succinct. Hence in Section 2.1.1 we discuss the unrolling of loops during pre-compilation.

In CoreVHDL the notion of signals is simplified with respect to full VHDL and does not allow references further ahead in time than the following *delta-cycle*. This correspond to the choice made in RTL VHDL and not only simplifies the analysis but also simplifies the definition of the semantics: Of the many accounts to be found in the literature [Goo95, TE01] we have found the one of [TE01] to best correspond to our practical experiments, based on test programs simulated with the ModelSim SE 5.7d VHDL simulator [Men03]. Even with this restriction CoreVHDL is sufficiently expressive to deal with the programs of the AES implementation.

2.1.1 Pre-compiling RTL VHDL to CoreVHDL

The CoreVHDL subset of RTL VHDL is restricted in its expressiveness. Here we discuss some commonly used syntactical constructs and their pre-compilation to CoreVHDL. The following operations are performed right after a RTL VHDL specification has been parsed.

Hierarchy flattening The VHDL standard [IEE01] describe how the hierarchical structure of a specification can be flattened. The pre-compiler instantiates

all the modules and in-line functions. This enables us to handle structural specifications by flattening them to elaborate behavioural specifications.

Concurrent Assignment Signal assignment can be performed as a concurrent statement, this corresponds to a process that is sensitive to the *free signals* in the right-hand side expression and that has the same assignment inside [Ash02].

Loop unrolling Loops are also allowed at the level of concurrent statements, these are commonly used to produce a multitude of processes with similar behaviour. Furthermore loops can be applied within processes, with semantics similar to that of imperative languages, see e.g. [NN92]. If the specification is synthesizable all loops need to result in a finite execution [IEC05]. Hence when considering synthesizable specifications loops are unrolled.

Scalarisation of vectors Following the approach presented in [SV00] vectors are scalarized and replace the new signals named like the vector and postfixed with the index, this is possible as only statically defined vectors are available in RTL VHDL [IEC05].

2.2 Semantics

Before we can define the semantics for CoreVHDL we need to understand how integrated circuits work. Most tools for synthesising circuits from VHDL descriptions support the configuration of *Field Programmable Gate Arrays* (FPGA). A FPGA consist of Logical Blocks (LB) and Input/Output Blocks (IOB), the structure is illustrated in Figure 2.3. The blocks are connected by wires, that in switch points can be interconnected or not. The synthesising step of the hardware design methodology is to decide the configuration of the switch points, thus giving the paths between LBs and IOBs. IOBs are the interface to the environment. The IOBs can also connect an outgoing wire to an incoming wire.

One important aspect of the synthesis step is that signals are mapped down to the wires connecting LBs and IOBs. It is also important to observe that the FPGA does not *execute* the VHDL specification, but instead provides a layout of the switches. The value of each signal will therefore be carried to and from the LB on different wires. In VHDL this is captured by dividing the values of signals into an active value and an present value, these match the from and to

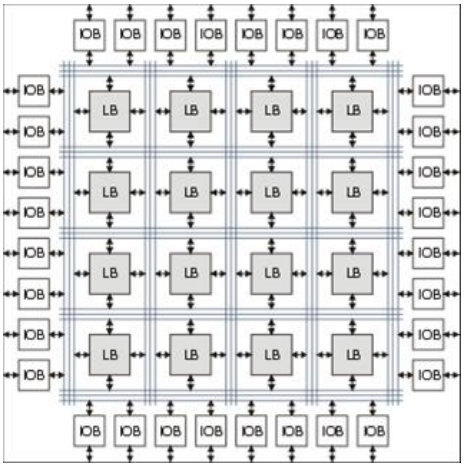


Figure 2.3: The structure of a FPGA.

wire, respectively. The time spent on stabilising and propagating the electrical current over the gates is called a *delta-delay*.

Another important aspect is that the integrated circuit does not *stop* working or otherwise end, in fact whenever the input is modified the new currents are propagated through the FPGA until it has stabilised again. To capture this behaviour when simulating hardware specifications we consider a *simulation cycle* as consisting of two phases:

- the signal update phase, and
- the process execution phase

In the signal update phase, simulation time is advanced to the time of the next scheduled active value, then all active values are applied to the corresponding signals. This may cause events to occur. In the process execution phase, all processes that wait on these events are resumed and executed until again suspended at wait statements. The simulation cycle is then repeated.

The main idea when defining the semantics for CoreVHDL programs is therefore to execute each process by itself until a synchronisation point is reached (i.e. a wait statement). When all processes of the program have reached a synchronisation point synchronisation is handled, while taking care of the resolution of signals in case a signal has been assigned different values by the processes. This

synchronisation will leave the processes in a state where they are ready either to continue execution by themselves or wait for the next synchronisation.

Basic semantic domains The syntax of programs in CoreVHDL is limited to statements operating on a state of logical values. These logical values are defined as $v \in LVal = \{\text{'U'}, \text{'X'}, \text{'0'}, \text{'1'}, \text{'Z'}, \text{'W'}, \text{'L'}, \text{'H'}, \text{'-'}\}$ where the values indicate the properties

'U'	Uninitialised	'X'	Forcing Unknown	'0'	Forcing zero
'1'	Forcing one	'Z'	High Impedance	'W'	Weak Unknown
'L'	Weak zero	'H'	Weak one	'-'	Don't care

these logical values capture the behaviour of an electrical system better than traditional boolean values. Hence they have been included in the VHDL standard from the second revision [IEE93b], see [Ash02] for further details. We have a function mapping logicals in the syntax to logical values in the semantics $\mathcal{L} : LVal \to Value$.

Constructed semantic domains CoreVHDL includes local variables and signals. The values of the local variables are stored in a local state, which is a mapping from variable names to logical values.

$$\sigma \in State = (Var \to Value)$$

The idea is that we have a local state for each process, keeping track of assignments to local variables encountered in the execution of the process so far.

For communication between the processes we have signals, the values of which are stored in local states. The processes communicate by synchronising the signals of their local signal state with other processes.

$$\varphi \in Signals = (Sig \to (\{0, 1\} \hookrightarrow Value))$$

The value assigned to a signal is available after the following synchronisation, therefore we keep the present value of a signal s in $\varphi\ s\ 0$. In $\varphi\ s\ 1$ we store the assigned value, meaning that it is available in the following *delta-cycle*. Each signal state has a time line for each signal. Values in the past are not used and therefore forgotten by the semantics; in CoreVHDL it is not possible to assign values to signals further into the future than one delta-cycle.

$\mathcal{E}[\![m]\!]\langle\sigma,\varphi\rangle$	$=$	$\mathcal{L}[\![m]\!]$	
$\mathcal{E}[\![x]\!]\langle\sigma,\varphi\rangle$	$=$	$\sigma\ x$	
$\mathcal{E}[\![s]\!]\langle\sigma,\varphi\rangle$	$=$	$\varphi\ s\ 0$	
$\mathcal{E}[\![op^u\ e]\!]\langle\sigma,\varphi\rangle$	$=$	$\overline{op^u}\ v$	where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle = v$ and $\overline{op^u}\ v$ defined
$\mathcal{E}[\![e_1\ op^b\ e_2]\!]\langle\sigma,\varphi\rangle$	$=$	$v_1\ \overline{op^b}\ v_2$	where $\mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle = v_1$ and $\mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle = v_2$ and $v_1\ \overline{op^b}\ v_2$ defined

Table 2.1: Semantics of Expressions

All signals have a present value, so $\varphi\ s\ 0$ is defined for all s. Not all signals need to be *active* meaning they have a new value waiting in the following delta-cycle, i.e. $\varphi\ s\ 1$ need not be defined. Hence we use $\{0,1\} \hookrightarrow Value$ in the definition of the signal state to indicate that it is a partial function.

The semantics handles expressions following the ideas of [NN92]. For expressions

$$\mathcal{E} : Exp \to (State \times Signals \hookrightarrow Value)$$

evaluates the expression. The function is defined in Table 2.1. Note that for signals we use the current value of the signal, i.e. $\varphi\ s\ 0$.

2.2.1 Statements

The semantics of statements and concurrent statements are specified by transition systems, more precisely by structural operational semantics [Plo04]. For statements we shall use configurations of the form:

$$\langle ss', \sigma, \varphi\rangle \in Stmt' \times State \times Signals$$

Here $Stmt'$ refers to the statements from the syntactical category $Stmt$ with an additional statement (`final`) indicating that a final configuration has been reached. Therefore the transition relation for statements has the form:

$$\langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle$$

which specifies one step of computation. The transition relation is specified in Table 2.2 and briefly commented upon below.

[Local Variable Assignment] :
$\langle x := e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto v], \varphi\rangle$
where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = v$

[Signal Assignment] :
$\langle s <= e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi^{[1]}[s \mapsto v]\rangle$
where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = v$

[Skip] :
$\langle \texttt{null}, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi\rangle$

[Composition] :
$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma', \varphi'\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi'\rangle}$$
where $ss_1' \in Stmt$

$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_2, \sigma', \varphi'\rangle}$$

[Conditional] :
$\langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \varphi\rangle \Rightarrow \langle ss_1, \sigma, \varphi\rangle$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle =$ '1'

$\langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \varphi\rangle \Rightarrow \langle ss_2, \sigma, \varphi\rangle$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle =$ '0'

Table 2.2: Statements

An assignment to a signal is defined as an update to the value at the delta-time, i.e. $\varphi\ s\ 1$. We use the notation $\varphi^{[i]}[s \mapsto v]$ to mean $\varphi[s \mapsto \varphi(s)[i \mapsto v]]$.

Because the wait statement is in fact a synchronisation point of the processes it is handled in Section 2.2.2, along with the handling of the concurrent processes.

2.2.2 Concurrent Statements

The semantics of concurrent statements handles the concurrent processes and the synchronisations of a CoreVHDL program. The transition system for concurrent statements has configurations of the form:

$$\|_{i \in I}\ \langle css_i', \sigma_i, \varphi_i\rangle$$

for $I \subseteq_{\text{fin}} Id$ and $css'_i \in Css'$, $\sigma_i \in State$, $\varphi_i \in Signals$ for all $i \in Id$. Thus each process has a local variable and signal state. Here Css' refers to the concurrent statements from the syntactical category Css with the additional option of statement from the category $Stmt$ preceding a concurrent statement. This allows for a succinct semantics for process statements.

The initial configuration of a CoreVHDL program is:

$$\|_{i \in I} \langle i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i, \sigma_i^0, \varphi_i^0 \rangle$$

The i^{th} process uses an initial state for signals defined by the semantics for declarations of signals. If no initial value is specified the following are used:

$\sigma_i^0\ x =$ 'U' and $\varphi_i^0\ s\ 0 =$ 'U' for all signals used in the process ss_i. $\varphi_i^0\ s\ 1$ is *undef* for all signals used in the process ss_i.

The transition relation for concurrent statements has the form:

$$\|_{i \in I} \langle css'_i, \sigma_i, \varphi_i \rangle \Longrightarrow \|_{i \in I} \langle {css_i}'', \sigma'_i, \varphi'_i \rangle$$

which specifies one step of computation.

The transition relation is specified in Table 2.3 and explained below.

As mentioned, the idea when defining the semantics of programs in CoreVHDL is that we execute processes locally until they have all arrived at a wait statement; this is reflected in the rule [**Handle non-waiting processes (H)**].

When all processes are ready to execute a wait statement we perform a synchronisation covered by the rule [**Active signals (A)**]. If a signal waited for is active, the processes waiting for that signal may proceed; this is expressed using the predicate $active(\varphi)$ defined by

$$active(\varphi) \equiv \exists s \exists v : \varphi\ s\ 1 = v$$

The delta-delayed values of signals will be synchronised for all processes and in order to do this we use a resolution function f_s of the form:

$$f_s : multiset(Value) \rightarrow Value$$

Thus f_s combines the *multi-set* of values assigned to a signal into one value that then will be the new (unique) value of the signal. VHDL allow a resolution function to be specified in a syntax much like that of a process. We assume that

[Handle non-waiting processes (H)] :

$$\frac{\langle ss_j, \sigma_j, \varphi_j\rangle \Rightarrow \langle ss'_j, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle ss'_i; css_i, \sigma'_i, \varphi'_i\rangle}$$

where $ss'_i = ss_i \wedge \sigma'_i = \sigma_i \wedge \varphi'_i = \varphi_i$ for all $i \neq j$.

$$\frac{\langle ss_j, \sigma_j, \varphi_j\rangle \Rightarrow \langle \texttt{final}, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle css'_i, \sigma'_i, \varphi'_i\rangle}$$

where $css'_i = css_i$ for all $i = j$ and
$css'_i = ss_i; css_i \wedge \sigma'_i = \sigma_i \wedge \varphi'_i = \varphi_i$ for all $i \neq j$.

[Active signals (A)] :

$\|_{i\in I}\ \langle \texttt{wait on}\ S_i\ \texttt{until}\ e_i; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle css'_i, \sigma_i, \varphi'_i\rangle$
if $\exists i \in I.\ active(\varphi_i)$
where

$$\varphi'_i\ s\ 0 = \begin{cases} f_s\{\!\{\mathcal{U}(\varphi_k, s) \mid k \in I\}\!\} & \text{if } \exists j \in I.\ \varphi_j\ s\ 1 \text{ is defined} \\ v & \text{otherwise, } \varphi_i\ s\ 0 = v \end{cases}$$

$$\varphi'_i\ s\ 1 = undef$$

$$css'_i = \begin{cases} css_i & \text{if } ((\exists s \in S_i.\ \varphi_i\ s\ 0 \neq \varphi'_i\ s\ 0) \\ & \wedge\ \mathcal{E}[\![e_i]\!]\langle\sigma_i, \varphi'_i\rangle =' 1') \\ \texttt{wait on}\ S_i\ \texttt{until}\ b_i; css_i & \text{otherwise} \end{cases}$$

[Processes (P)] :

$\|_{i\in I}\ \langle i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i, \sigma_i, \varphi_i\rangle \Longrightarrow$
$\quad \|_{i\in I}\ \langle ss_i; i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i, \sigma_i, \varphi_i\rangle$

Table 2.3: Concurrent statements

all signals handled in resolution functions are in the argument of the function. The resolution function is applied to the active values of a signal, however if a signal is not active in a local signal state of a process the present value is used instead. We introduce the function

$$\mathcal{U}(\varphi_i, s) = \begin{cases} v & \text{if } \varphi_i\ s\ 1 = v \\ v & \text{otherwise, } \varphi_i\ s\ 0 = v \end{cases}$$

to determine whether the active value or the present value is used.

The VHDL standard [IEE01] defines a resolution function for the type `std_logic` called `resolve`. The function is applied on a list of values for the signal. If the function is passed an empty list, it returns the value $'Z'$. If there is only one driving value, the function returns that value unchanged. Otherwise, the func-

	$'U'$	$'X'$	$'0'$	$'1'$	$'Z'$	$'W'$	$'L'$	$'H'$	$'-'$
$'U'$	$'U'$	$'U'$	$'U'$	$'U'$	$'U'$	$'U'$	$'U'$	$'U'$	$'U'$
$'X'$	$'U'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$
$'0'$	$'U'$	$'X'$	$'0'$	$'X'$	$'0'$	$'0'$	$'0'$	$'0'$	$'X'$
$'1'$	$'U'$	$'X'$	$'X'$	$'1'$	$'1'$	$'1'$	$'1'$	$'1'$	$'X'$
$'Z'$	$'U'$	$'X'$	$'0'$	$'1'$	$'Z'$	$'W'$	$'L'$	$'H'$	$'X'$
$'W'$	$'U'$	$'X'$	$'0'$	$'1'$	$'W'$	$'W'$	$'W'$	$'W'$	$'X'$
$'L'$	$'U'$	$'X'$	$'0'$	$'1'$	$'L'$	$'W'$	$'L'$	$'W'$	$'X'$
$'H'$	$'U'$	$'X'$	$'0'$	$'1'$	$'H'$	$'W'$	$'W'$	$'H'$	$'X'$
$'-'$	$'U'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$	$'X'$

Table 2.4: Resolution table rt

tion uses a resolution table to resolve driving values

$$\texttt{resolve}\{\!\{s_1, s_2, \ldots, s_n\}\!\} = rt(s_1, rt(s_2, rt(\ldots, s_n)\cdots)))$$

where the resolution table rt is defined in Table 2.4.

Notice that even though a signal, which a wait statement is waiting for becomes active, it is not enough to guarantee that it proceeds with its execution. This is because we have the side condition '`until` e'. This is reflected in the definition of the statement css'_i of the next configuration. Notice that the state of local variables is unchanged.

Recall that it is the intention to repeat the statement ss_i indefinitely in a process declaration i: `process` $decl_i$; `begin` ss_i; `end process` i , this is reflected in the rule [**Processes (P)**].

In the following chapters we will investigate security policies and mechanisms that dynamically change due to constraints on the history of execution. Therefore we will now define an execution trace. The point of interest is that specific executions take place prior to releasing specific information. Hence the semantics are annotated with the identifier of the process that gets to evaluate part of its body. Thus we write

$$\|_{i\in I}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \stackrel{i}{\Longrightarrow} \|_{i\in I}\ \langle ss'_i; css_i, \sigma'_i, \varphi'_i\rangle$$

when an evaluation step with the rule [**Handle non-waiting processes (H)**] evaluated one step of process i. As a result an execution trace is then defined as the transitive reflexive closure of the semantics, and we write

$$\|_{i\in I}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \stackrel{\omega}{\Longrightarrow}{}^{*}\ \|_{i\in I}\ \langle ss'_i; css_i, \sigma'_i, \varphi'_i\rangle$$

where ω is a string of identifiers.

2.2.3 Architectures

The Semantics for architectures basically initialises the local variable and signal stores for each process. As no actual evaluation is taking place when handling the architectures, we merely consider specifications where the variable and signal declarations are stripped. Hence no further formal definitions are considered.

2.3 Properties of the Semantics

In this section we will state properties of the Semantics that will aid us in establishing correctness results of the analyses presented in later Chapters. One property of the Semantics is that the execution of a statement ss_1 is not influenced by the statement following.

Lemma 2.1 If $\langle ss_1, \sigma, \varphi\rangle \Rightarrow^k \langle \mathtt{final}, \sigma', \varphi'\rangle$ then $\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow^k \langle ss_2, \sigma', \varphi'\rangle$.

Proof. The proof proceeds by induction in the length of the derivation sequence. If $k = 0$ the result holds vacuously. For the induction step we assume that the lemma holds for $k \leq k_0$ and we shall prove it for $k_0 + 1$. So assume that

$$\langle ss_1, \sigma, \varphi\rangle \Rightarrow^{k_0+1} \langle \mathtt{final}, \sigma', \varphi'\rangle$$

hence the derivation sequence can be written

$$\langle ss_1, \sigma, \varphi\rangle \Rightarrow \gamma \Rightarrow^{k_0} \langle \mathtt{final}, \sigma', \varphi'\rangle$$

for some configuration γ. Now we know that the rule [**Composition**] was used to obtain

$$\langle ss_1, \sigma, \varphi\rangle \Rightarrow \gamma$$

Hence we consider the two cases, first the case where statement ss_1 evaluates to ss_1' in one step

$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma'', \varphi''\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma'', \varphi''\rangle}$$

and get

$$\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma'', \varphi''\rangle \Rightarrow^{k_0} \langle ss_2, \sigma', \varphi'\rangle$$

and we can apply the induction hypothesis on the derivation sequence

$$\langle ss_1'; ss_2, \sigma'', \varphi''\rangle \Rightarrow^{k_0} \langle ss_2, \sigma', \varphi'\rangle$$

The second case is when ss_1 finish evaluation in one step

$$\frac{\langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma'', \varphi''\rangle}{\langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_2, \sigma'', \varphi''\rangle}$$

where the result follows immediately. □

2.4 Example: Advanced Encryption Standard

The techniques presented in this thesis will be applied to the standard implementation of the *Advanced Encryption Standard* [WBRF00]. We shall consider the *pipelined* 128 bit version of the NSA *Advanced Encryption Standard* implementation of the algorithm. As the specification is rather substantial, and large parts are merely concerned with control signals and registers, we will present two subprograms, which will be used to illustrate the presented analyses.

The 128 bit version of the algorithm works on blocks of 128 bit, that are encrypted through 10 rounds. Each round consists of 4 stages that substitute, shift, mix and add a *round key* to the block, respectively. Decryption is done by reversing the order of the inverse stages in each round.

The first subprogram is the shift function that is part of an encryption or decryption round.

```
-- iteration no. 1
temp(0) := a(1)(1);
temp(1) := a(1)(2);
temp(2) := a(1)(3);
temp(3) := a(1)(0);

a(1)(0) := temp(0);
```

```
   a(1)(1) := temp(1);
   a(1)(2) := temp(2);
   a(1)(3) := temp(3);

   -- iteration no. 2
   temp(0) := a(2)(2);
   temp(1) := a(2)(3);
   temp(2) := a(2)(0);
   temp(3) := a(2)(1);

   a(2)(0) := temp(0);
   a(2)(1) := temp(1);
   a(2)(2) := temp(2);
   a(2)(3) := temp(3);

   -- iteration no. 3
   temp(0) := a(3)(3);
   temp(1) := a(3)(0);
   temp(2) := a(3)(1);
   temp(3) := a(3)(2);

   a(3)(0) := temp(0);
   a(3)(1) := temp(1);
   a(3)(2) := temp(2);
   a(3)(3) := temp(3);
```

The second subprogram is the process that determines the control flow of round blocks, and decides whether the key is added before or after the 10 rounds. This is the difference between the encryption and the decryption algorithms. The implementation considered performs either encryption or decryption depending on the value of the incoming signal `ALG_ENC`. For encryption this is done before the first round instead.

```
POST_ADD_KEY   <= ALG_KEY_10;

POST_ADD: process begin
  b(0)(0) := FINAL_OUT_REG(0)(0) xor POST_ADD_KEY(0)(0);
  b(0)(1) := FINAL_OUT_REG(0)(1) xor POST_ADD_KEY(0)(1);
  b(0)(2) := FINAL_OUT_REG(0)(2) xor POST_ADD_KEY(0)(2);
  b(0)(3) := FINAL_OUT_REG(0)(3) xor POST_ADD_KEY(0)(3);

  b(1)(0) := FINAL_OUT_REG(1)(0) xor POST_ADD_KEY(1)(0);
```

```
  b(1)(1) := FINAL_OUT_REG(1)(1) xor POST_ADD_KEY(1)(1);
  b(1)(2) := FINAL_OUT_REG(1)(2) xor POST_ADD_KEY(1)(2);
  b(1)(3) := FINAL_OUT_REG(1)(3) xor POST_ADD_KEY(1)(3);

  b(2)(0) := FINAL_OUT_REG(2)(0) xor POST_ADD_KEY(2)(0);
  b(2)(1) := FINAL_OUT_REG(2)(1) xor POST_ADD_KEY(2)(1);
  b(2)(2) := FINAL_OUT_REG(2)(2) xor POST_ADD_KEY(2)(2);
  b(2)(3) := FINAL_OUT_REG(2)(3) xor POST_ADD_KEY(2)(3);

  b(3)(0) := FINAL_OUT_REG(3)(0) xor POST_ADD_KEY(3)(0);
  b(3)(1) := FINAL_OUT_REG(3)(1) xor POST_ADD_KEY(3)(1);
  b(3)(2) := FINAL_OUT_REG(3)(2) xor POST_ADD_KEY(3)(2);
  b(3)(3) := FINAL_OUT_REG(3)(3) xor POST_ADD_KEY(3)(3);

  POST_OUT_INT <= b;
end process POST_ADD;


ALG_DATA_LAST <= POST_OUT_INT when ALG_ENC = '0' else
                 FINAL_OUT;
```

2.5 Discussion and Related Work

In this thesis focus will be on VHDL. However, other hardware description languages exists. The most predominant alternative to VHDL is the Verilog Hardware Description Language [IEE95]. The choice of VHDL was made because of the wide acceptance and support of the language in the industry. Still the common foundation of hardware description languages should allow for a fairly straightforward adaption of the techniques presented in the following chapters to other languages.

Another alternative might be the *system description language* SystemC [IEE05]. The SystemC language provides a greater freedom of expressiveness as it consists of libraries and macros for C++, hence allowing the C++ language facilities to be used in hardware descriptions. However, when synthesising designs, the procedure is normally to compile the SystemC descriptions to VHDL, where further development might be necessary. Therefore the analyses presented in this thesis can be applied at the VHDL level, to achieve similar results for SystemC specifications.

2.5.1 Semantics for VHDL

There exists alternative definitions of semantics for VHDL. In the following we will discuss these and argue for some of the choices, that make our semantics differ from others. The reason for these differences is primarily the ambiguity and complexity of the IEEE standard [IEE01], which allows for varying interpretations. To validate the choices made we therefore focus on the specification of synthesising RTL VHDL [IEC05] and the ModelSim SE 5.7d VHDL simulator [Men03].

The semantics presented in this chapter is based on the structural operational semantics presented by Goossens [Goo95], which aims to formalise the simulation algorithm described in the original VHDL standard [IEE87]. Thirunarayan and Ewing [TE01] extended this semantics to comply with the 1993 standard. The presented semantics is very close to these approaches, but still there are two issues that we address differently. First the progression of time at synchronisation points that update the present values of signals was erroneously formalised. Thirunarayan and Ewing points out the problem in their paper [TE01], however they correction does not solve the issue, but instead introduces a copying of the active value of signals whenever no processes can continue past their synchronisation points. Thereafter a process might mistakenly observe the signal as being active twice. Thirunarayan has acknowledged this issue and agreed to the solution presented in this thesis [Thi03].

The second issue has to do with the resolution of signals. In [Goo95, TE01] the resolution function is applied only on the active values of signals. However there is no distinction between the active and the present value in hardware and hence the resolution function should in fact be applied on both. This observation is supported by [Ash02] and ModelSim simulations of test specifications.

There are other alternatives. Many leave out important issues in the hardware model, such as
"For simplicity, delta-delayed signal assignments are not *treated."* [BFK94],
which allows Breuer et al. to define a *clean* semantics of VHDL. Van Tassel [vT93] makes similar simplification to the considered subset of VHDL. These semantics do not match the understanding of hardware design investigated in this thesis.

Van Tassel [vT93] embeds the semantics of a VHDL subset in HOL, in this manner allowing for proof assistance. In the same line of research Russinoff [Rus95] defines the semantics of a simple hardware description language, which is similar to a subset of VHDL, in Boyer-Moore logic. Another approach that defines the semantics of VHDL in a logical language (i.e. ACL) is [RBG00].

These embeddings into proof assistants aid the formal verification of VHDL specifications.

Several hardware description languages are focused completely on finite state machines, such as Esterel [Est05]. In fact most VHDL specification were focused on describing globally synchronous specifications, that can be mapped to finite state machines. This is still one of its primary usages. Hence much work has been done on compiling VHDL descriptions into finite state machines, see e.g. [BLPV94, DB93, DB95].

Hymans [Hym02] defines a semantics that is based on Goossens' [Goo95] approach. The considered subset of VHDL is very close to the one at hand. The main differences is that Hymans introduces functions for random number generation to the language. These are not present in VHDL and are not synthesizable. Hymans' reason for introducing them is to accommodate verification of systems. Hence only a *test bench* process would use the functions. The semantics, however, has been modified to not include resolution of active signals. Instead Hymans utilises a global state that allows signal assignments to overwrite each other. This forces Hymans to assume that each signal is only assigned to in one process.

To verify that the presented definition of the semantics is accurate several test specifications were made that support or reject design choices in the semantics. The test specifications were simulated with the ModelSim simulator [Men03]. Although this does not provide any formal guarantee, it does allow us to debug the formal specifications. Below we report on two of the example specifications and the results of simulation them.

Waiting on signals In the semantics of programs the condition for allowing a process to continue execution after waiting for a signal is limited to signals that have actually changed value. I.e. using the definition of active signals is not sufficient to determine if processes continue execution in the rule [**Active signals (A)**]. We see from the below described program that we need the condition $\exists s \in S_i.\ \varphi_i\ s\ 0 \neq \varphi_i'\ s\ 0$.

To illustrate this decision in the semantics we present a program, consisting of two processes. One process will continue to set the value of a signal to a constant value. The second process will wait on the signal set by the first process, and when allowed to continue execution it will alter another signal. This allows us to detect that the process is not halted while waiting on the signal. The signal `CLK` is a clock signal set by the simulator. It is inverted with a frequency of 500 ns.

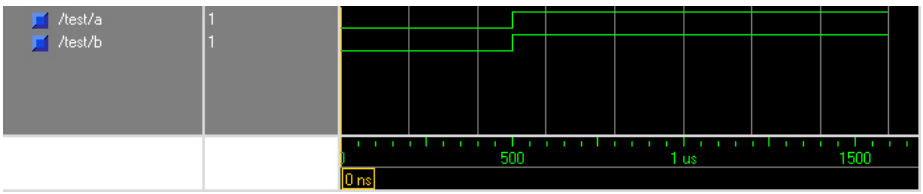


Figure 2.4: Result of simulating the program.

The result of simulating the program with the ModelSim SE 5.7d VHDL simulator is presented in Figure 2.4. Since the value of the signal `B` only changes once, namely after 500 ns and not again after 1000 ns or 1500 ns, we observe that the second process can not continue execution at the wait statement even though the first process sets an active value for the signal `A`.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test is
  signal CLK : in STD_LOGIC;
end test;

architecture model of test is
  signal A : STD_LOGIC := '0';
  signal B : STD_LOGIC := '0';
begin

  process begin
    wait on CLK;
    A <= '1';
  end process;

  process begin
    wait on A;
    B <= not B;
  end process;

end test;
```

Resolution off signals In the semantics of programs it is specified how signals are resolved with a resolution function. Different test programs have been

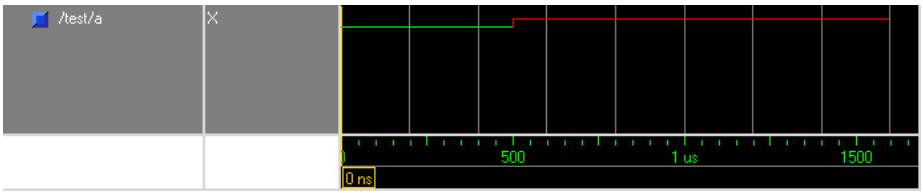


Figure 2.5: Result of simulating the program.

simulated and the behaviour corresponds to the resolution function specified in [Ash02].

The simulated programs are of a form similar to the program presented below. The program consists of two processes running in parallel, each of them setting the value of the signal `A`. When the processes synchronise at the wait statements the resolution of the signal is performed. By altering the values assigned to `A` each entry in the resolution table can be tested. As above the signal `CLK` is a clock signal set by the simulator. It is inverted with a frequency of 500 ns.

The result of simulating the test program is presented in Figure 2.5.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test is
  signal CLK : in STD_LOGIC;
end test;

architecture model of test is
  signal A : STD_LOGIC := '0';
begin

  process begin
    wait on CLK;
    A <= '1';
  end process;

  process begin
    wait on CLK;
    A <= '0';
  end process;

end model;
```

Chapter 3

Security Policies

The only thing necessary for the triumph of evil
is for good men to do nothing.
— Edmund Burke

Security verification of a system is divided into two parts; a security policy and a security mechanism. The security policy is a high-level specification of the security properties that the given system should possess; typically it consists of statements describing goals of the security verification. These goals should be driven by our understanding of threats rather than how the system is implemented. Hence a policy might express goals regarding the access to objects or how information flows through the system. On the other hand the security mechanism enforces the security properties on the given system. In this chapter we investigate security policies for information flow. Security mechanisms matching the security policies are investigated in Chapter 4 and 5.

The concept of security policies came from the military sector. The first security policy model, Bell-LaPadula [BL73], was developed in 1973 to formalise the U.S. Department of Defense *multilevel security* policy. The model describes a set of access control rules through security labels on resources and clearances for users. Informally said, the model protects the confidentiality of information through a hierarchy of security levels. A user can not read information from levels higher than the users own, *no read up*, and the user can not write information to lower levels, *no write down*. This model became the standard military approach when evaluating systems, according to the U.S. Department of Defense "Orange

Book" [DOD85]. However for general usage the model has proved to be too restrictive, in particular due to it relying on dynamic security mechanisms that have problems with enforcing the policy in the presence of *implicit* flows. Here we make the distinction between *explicit* and *implicit* information flows; where explicit information flows are statements that directly affects one resource by reading another resource, e.g.

$$x := y$$

where the value of y is copied to x. An implicit flow occurs in connection with statements that alter the *control flow* of the program based on reading a resource, e.g.

$$y := 0;\ \texttt{if}\ x\ \texttt{then}\ y := 1\ \texttt{else null}$$

where the value of x will determine which branch will be taken and finally the value of y. Observe that the implicit flow occur even when $x \neq true$ as the attacker may observe that the value of y is 0.

Due to these issues investigation began on policies and mechanisms for preserving confidentiality in a system. On one hand Denning and Denning [Den76, DD77] developed static mechanisms for verifying the security of a system, while on the other hand Lampson [Lam73] and later Cohen [Coh77, Coh78] investigated notions for the *confinement problem* and *strong secrecy*. Goguen and Meseguer [GM82, GM84] introduced the security property, *non-interference*, that has later become the most influential property of information flow policies. Informally, non-interference means that an attacker of the system, who can view low confidentiality information, is not permitted to observe any differences after two executions of a program that only differ on its confidential inputs. Another weaker property, *non-deducibility*, was proposed by Sutherland [Sut86]. It was the first nondeterministic version of non-interference. It is weaker than non-interference as it allows the two executions of the program to modify the observable part of the resources as long as no connection between confidential information and resulting low confidentiality information can be made with complete certainty. The non-deducibility property has no practical usage, since a guarantee that allows the attacker to determine a secret with 99% probability is no better than no guarantee.

At about the same time Kemmerer [Kem82, Kem02] presented the *shared resource matrix methodology* aimed at identifying *covert channels* in *real* systems. The method is centred around the notion of a *resource matrix*, which is a matrix where the rows represent the resources available in the considered system and the columns represent the actions (referred to as primitives). Each field in the matrix state the access right for the action on the resource. In this model there are two access rights, *read* and *modify*. The resource matrix model allows

the security analyst to abstract on the resources and actions, and hence include resources such as the process scheduler and the system clock, while grouping actions into processes and programs. The strength of the shared resource matrix approach is verifying real systems, which has been illustrated in several cases (e.g. [HKMY86, KT96]). One drawback of the approach is that it does not formalise a security property, instead it aims at identifying information flows (and more generally covert channels). Thus if no leaks occur we are left without any guarantees.

In this chapter we aim to combine the expressive power of the security policies in the shared resource matrix approach with the strong *end-to-end* guarantee of non-interference. Arguments regarding the inapplicability of noninterference in the setting of many real systems have been raised [RMMG01]. One approach that has been relatively successful in dealing with this issue is introducing trusted components that are allowed to break the security policies, often referred to as *intransitive noninterference* [HY86, Rus92, RG99]. The proposed policies and security property incorporate a notion of intransitive noninterference and extend it with constraints on the execution history of the program. Previously security policies for access control based on execution history [AF03, BN04, BDF05] have been investigated and shown to provide a natural approach to specifying security policies. Furthermore the proposed policies and security property must be placed within the setting of hardware descriptions. For this purpose we build on the semantics presented in Chapter 2.

3.1 Locality-based security policies

To protect confidentiality within a system, it is important to control how information flows so that secret information is prevented from being released on unintended channels. The allowed flow of information in a system is specified in a security policy. In this section we present a policy language based on graphs. Here vertices represent security domains, describing the resources available in the program. A security domain is related to sets of resources available in the considered system.

This model allows for the granularity of the mapping from resources to security domains to be very fine-grained. For example we can introduce a security domain for each resource. For improved precision we could partition the usage of a resource into lifetime periods and introduce a domain for each, hence having more than one security domain for each resource used in the program. This would allow us to abstract from e.g. reuse of limited resources in the implementation, instead of introducing new gates for each lifetime period of a signal, and

thereby potentially increasing the size of the gate layout quadratically [HS06].

In our setting the variables and signals are our resources, hence they are related to security domains. This allows us to reason about groups of resources together, as well as singling out specific resources and isolate these in their own security domains. The security policies therefore focus on the information flow between intended domains of resources. Consequently we assume that variables and signals can be uniquely mapped to security domains.

Definition 3.1 (Security Domains) For a given program we have a mapping from variables and signals $Var \cup Sig$ to security domains V

$$\underline{\cdot} : Var \cup Sig \ \rightarrow \ V$$

We write $\underline{x}$ and $\underline{s}$ for the security domain of the variable x and signal s, respectively.

As stated above the security policies are based on graphs. Therefore the edges specify permitted flows of information. Information flow between resources can be restricted subject to fulfilment of constraints with respect to certain events taking place prior to the flow. Formally we propose the following definition of security polices.

Definition 3.2 (Locality-based security policies) A security policy is a labelled graph $G = (V, \lambda)$, consisting of a set of vertices V representing security domains and a total function λ mapping pairs of vertices to labels $\lambda : V \times V \rightarrow \Delta$. We define $\mathbb{G}$ to be the set of policies. The structure, Δ, of labels is defined below.

In a flow graph the set of vertices V represent security domains. A security domain indicates the usage of a resource in the system. The edges in the flow graph describe the allowed information flow between resources. Hence an edge from the vertex for security domain v_1 to the vertex for security domain v_2 indicates that information is allowed to flow between the resources in these domains subject to the label attached to the edge.

The edges are labelled with constraints that the program must fulfill in order for the flow of information to be permitted. These constraints will include restrictions with respect to localities in the program. Hence we will consider local flows permitted in specific processes or blocks. Each process and block is uniquely referenced by identifiers Id that we map to security localities.

Definition 3.3 (Security Localities) For a given program we have a mapping

from identifiers Id to security localities L

$$\underline{\underline{\cdot}} : Id \ \rightarrow \ L$$

A policy that restrict a information flow to a security location l should permit information flows when the execution resulting in the flow takes place in a process i_p or block i_b that is in the location, i.e. $\underline{\underline{i_p}} = l$ or $\underline{\underline{i_b}} = l$, respectively. We lift the mapping of identifiers to security localities such that we write $\underline{\underline{\omega}}$ when the mapping is in fact applied to each identifier in a execution trace, thus resulting in a string of locations.

The edges in the flow graph are described by the function $\lambda : V \times V \rightarrow \Delta$. We write $v_1 \overset{\delta}{\rightsquigarrow} v_2$ for an edge from vertex v_1 to v_2 constrained by $\delta \in \Delta$, i.e. $\lambda(v_1, v_2) = \delta$. Information flow might be constrained by certain obligations that the system must fulfill before the flow can take place. Here we describe a novel constraint language that allows the security policy to be specific about intransitive information flows in the flow graph. Constraints are specified in the following syntax $\delta \in \Delta$:

$$\delta ::= true \mid false \mid l \mid \delta_1 \cdot \delta_2 \mid \delta_1 \wedge \delta_2 \mid \delta_1 \vee \delta_2 \mid \delta^*$$

A constraint may be trivially *true* or *false*. The constraint l enforces that flows occur at specific locations, thus that the flow is only permitted at locations that is part of the security location l (i.e. $\underline{\underline{i}} = l$). The constraint $\delta_1 \cdot \delta_2$ enforces that the flow is only allowed to occur if events described in δ_1 precedes events described in δ_2. Common logical operators $\wedge$ and $\vee$ are available to compose constraints. Finally Kleene's star * allows the policies to express cyclic behaviour.

We might omit the constraint, writing $v_1 \rightsquigarrow v_2$ for $v_1 \overset{true}{\rightsquigarrow} v_2$. Similarly we might omit an edge in a flow graph indicating that the constraint can never be fulfilled, i.e. $v_1 \overset{false}{\rightsquigarrow} v_2$.

Example 3.1 *To illustrate the usage of flow graphs as security policies we here discuss the three examples given in Figure 3.1. The first flow graph (a) allows a flow from v_1 to v_2 and v_2 to v_3 but not from v_1 to v_3. That is neither directly nor through v_2! In this manner the intransitive and temporal nature of the policies allow us to have constraints on the order of information flows and the further propagation of information.*

The second flow graph (b) allows the flow from v_1 to v_2, v_2 to v_3 and from v_1 to v_3 as well. The flow can be directly from v_1 to v_3 or through v_2. If we

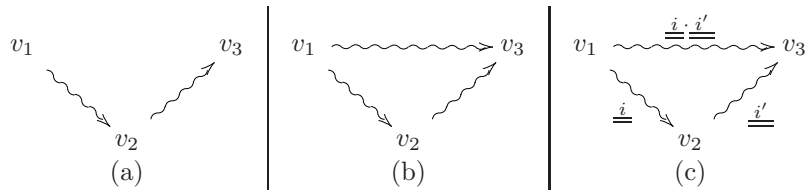


Figure 3.1: Examples of flow graphs

wish to restrict the flow to go through v_2 it could be done as in flow graph (c). We assume that $\underline{\underline{i}}$ and $\underline{\underline{i'}}$ map to security locations that no other processes or blocks are mapped to. Hence the last flow graph (c) restricts the flows between the security domains to certain locations. This ensures that for information to flow from v_1 to v_3 both locations need to participate.

To give intuition to the above example policies we relate them to a personal finance scenario. In the following we let v_1, v_2, and v_3 denote the resources of the user, the user's financial advisor, and the tax authorities respectively. The first policy (Figure 3.1(a)) states that the user's financial information may be accessed by the financial advisor but *not* by the tax authorities while still allowing the financial advisor to send (other) information to the tax authorities. The second policy (Figure 3.1(b)) then states that the user's financial information may be accessed both by the financial advisor and by the tax authorities; this may be necessary during a tax audit. Finally the policy shown in (Figure 3.1(c)) defines a situation where the user's financial information may be accessed by both the financial advisor and the tax authorities but *only* through the financial advisor; this security policy ensures that the financial advisor can review all relevant information from the user before the tax authorities gain access to it. These examples match the principle of *separation of duty* presented by Clark and Wilson [CW87], as one of the fundamental principles in the Clark-Wilson model. The principle states that in order to maintain integrity in a commercial system the concerns of each principal must be separated, and hence collaboration between principals becomes necessary to achieve the goals of the individuals, while malicious principals must not be allowed to circumvent the rights of the well-behaved principals.

A system is given by a program $\|_{i \in I}\ css_i$ and a security policy G together with a domain mapping $\underline{\cdot}$ and a location mapping $\underline{\underline{\cdot}}$ and might be written $\|_{i \in I}\ css_i$ subject to $(G, \underline{\cdot}, \underline{\underline{\cdot}})$. However, as the G, $\underline{\cdot}$ and $\underline{\underline{\cdot}}$ components are clear from context we choose not to incorporate the policies in the syntax of CoreVHDL.

3.1.1 Semantics of constraints

In this subsection we present the semantics of our security policy language. The semantics is given as a translation to regular expressions over execution traces. The intuition is that if an execution trace is in the language of the regular expression derived by the semantics, then the constraint is fulfilled.

The main idea behind the locality-based security policies is that they specify constraints that must be fulfilled rather than the total behaviour of the system. This result in succinct policies. Therefore the semantics of constraints is based on an understanding of fulfilment of a constraint whenever an execution trace produces the locations specified. Hence if a trace that fulfils a constraint is preceded or succeeded by other traces the constraint remains fulfilled.

The *true* constraint is always fulfilled and hence is translated to the regular expression L^* accepting all execution traces. Similarly the constraint *false* cannot be fulfilled for any trace, and hence the generated language is $\emptyset$. The constraint l gives the language $L^* \cdot \{l\} \cdot L^*$ as we wish to allow the flow only for executions taking place at l. The constraint $\delta_1 \cdot \delta_2$ indicates that the trace $\underline{\underline{\omega}}$ can be split into $\underline{\underline{\omega_1}}$ and $\underline{\underline{\omega_2}}$, where $\underline{\underline{\omega_1}}$ must be in the language of δ_1, and respectively $\underline{\underline{\omega_2}}$ must be in the language of δ_2. The constraints for $\delta_1 \wedge \delta_2$ and $\delta_1 \vee \delta_2$ are straightforward. One obvious definition of $[\![\delta^*]\!]$ is $[\![\delta]\!]^*$; however this choice would invalidate Lemma 3.4 below. Consequently, it is natural to define $[\![\delta^*]\!] = L^* \cdot [\![\delta]\!]^* \cdot L^*$ as then Lemma 3.4 continues to hold.

The semantics of the security policy language is given in Figure 3.2.

Lemma 3.4 The semantical interpretation of constraint δ does not change by preceding or succeeding it by other traces $\forall \delta : [\![\delta]\!] = L^* \cdot [\![\delta]\!] \cdot L^*$

We define an ordering of constraints as the relation $\leq_\Delta \subseteq \Delta \times \Delta$, i.e. we say that δ is a restriction of δ' if $(\delta, \delta') \in \leq_\Delta$, normally we write $\delta \leq_\Delta \delta'$.

Definition 3.5 We say that a constraint δ is a restriction of δ', written $\delta \leq_\Delta \delta'$ if we have $[\![\delta]\!] \subseteq [\![\delta']\!]$.

Similarly we define a restriction relation between flow graphs.

Definition 3.6 We say that a flow graph $G = (V, \lambda)$ is a restriction of $G' = (V', \lambda')$, written $G \leq G'$ if we have that

$$V = V' \ \wedge \ \forall v_1, v_2 : \lambda(v_1, v_2) \leq_\Delta \lambda'(v_1, v_2)$$

$$\begin{array}{rcl} [\![true]\!] & = & L^* \\ [\![false]\!] & = & \emptyset \\ [\![l]\!] & = & L^* \cdot \{l\} \cdot L^* \\ [\![\delta_1 \cdot \delta_2]\!] & = & [\![\delta_1]\!] \cdot [\![\delta_2]\!] \\ [\![\delta_1 \wedge \delta_2]\!] & = & [\![\delta_1]\!] \cap [\![\delta_2]\!] \\ [\![\delta_1 \vee \delta_2]\!] & = & [\![\delta_1]\!] \cup [\![\delta_2]\!] \\ [\![\delta^*]\!] & = & L^* \cdot [\![\delta]\!]^* \cdot L^* \end{array}$$

Figure 3.2: Semantics of constraints

3.2 History-based Release

To determine whether a program is secure or not, we need some condition for security. In this section we therefore present our definition of secure programs. The intuition is similar to that of non-interference, however we aim to generalise the traditional non-interference condition to permit release of confidential information, based on constraints on the history of the execution and it's present location. We call this condition *History-based Release*.

3.2.1 Security Condition

Before we formalise the main security condition we need to formalise what an attacker can observe. Consider an attacker that has access to a subset $\mathcal{V}$ of the variables and signals that are available in the program under consideration. We formalise the observable part of the resources by assuming that the attacker has knowledge of all the processes that exist in the program, and hence it is feasible to assume that two variable or signal states can be compared on all variables or signals, respectively. Thus for two programs $\|_{i \in I}\ css_{i1}$ and $\|_{i \in I}\ css_{i2}$ an attacker that observes at security domain $\mathcal{V}$ can compare the observable parts of the variable states as $\sigma_1 \sim_{\mathcal{V}} \sigma_2$.

Definition 3.7 ($\mathcal{V}$-observable equivalence) Two variable states σ_1 and σ_2 are for the resources in domain $\mathcal{V}$ observably equivalent $\sigma_1 \sim_{\mathcal{V}} \sigma_2$ iff

$$\forall x : \underline{x} = \mathcal{V} \Rightarrow \sigma_1\ x = \sigma_2\ x$$

We define the observable parts of signal states in the same manner.

Definition 3.8 ($\mathcal{V}$-observable equivalence) Two signal states φ_1 and φ_2 are for the resources in domain $\mathcal{V}$ observably equivalent $\varphi_1 \sim_{\mathcal{V}} \varphi_2$ iff

$$\forall s : \forall j \in \{0,1\} : \underline{s} = \mathcal{V} \Rightarrow \varphi_1\ s\ j = \varphi_2\ s\ j$$

Where we overload the observational equivalence to consider both variable and signal states.

We define the function $\nabla : \mathcal{P}(V) \times \mathbb{G} \times \Omega \rightarrow \mathcal{P}(V)$ for extending a set of security domains with the domains that are permitted to interfere with the observed domains due to the fulfilment of constraints by the execution trace ω. The resulting set of security domains describe the permitted information flows during the execution.

Definition 3.9 ($\mathcal{V}$ Observable) For a security policy G and a execution trace ω an observer at $\mathcal{V}$ can observe the localities

$$\nabla(\mathcal{V}, G, \omega) = \mathcal{V} \cup \{v_1 \mid v_2 \in \mathcal{V} \ \wedge \ \underline{\underline{\omega}} \in [\![\lambda(v_1, v_2)]\!]\}$$

If the considered policy is changed to a less restrictive policy, ∇ will never reduce the observable part of the variables and signals. This allows us to establish the following fact.

Fact 1. *If* $G \leq G'$ *then* $\nabla(\mathcal{V}, G, \omega) \subseteq \nabla(\mathcal{V}, G', \omega)$.

We consider a program secure if in no execution trace, neither a single step nor a series of steps, will an attacker observing the system at the level of a set of security domains $\mathcal{V}$ be able to observe a difference on any resource, when all resources permitted to interfere with the resource are observably equivalent before the evaluation. We formalise the condition as a bisimulation over execution traces on programs.

Definition 3.10 (Bisimulation) A $(G, \mathcal{V})$-bisimulation is a symmetric relation $\mathcal{R}$ on programs whenever

$$\begin{array}{l}
\|_{i\in I}\ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle \stackrel{\omega}{\Longrightarrow}{}^{*}\ \ \|_{i\in I}\ \langle css_{i1}{}'', \sigma'_{i1}, \varphi'_{i1}\rangle \ \wedge \\
\|_{i\in I}\ css'_{i1}\ \mathcal{R}\ \ \|_{i\in I}\ css'_{i2}\ \wedge \\
\sigma_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \sigma_{i2}\ \wedge \\
\varphi_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \varphi_{i2}
\end{array}$$

then there exists $\|_{i\in I}\ css''_{i2}$, σ'_{i2}, φ'_{i2} and ω' such that

$$\begin{array}{l}
\|_{i\in I}\ \langle css'_{i2}, \sigma_{i2}, \varphi_{i2}\rangle \stackrel{\omega'}{\Longrightarrow}{}^{*}\ \ \|_{i\in I}\ \langle css_{i2}{}'', \sigma'_{i2}, \varphi'_{i2}\rangle \ \wedge \\
\|_{i\in I}\ css''_{i1}\ \mathcal{R}\ \ \|_{i\in I}\ css''_{i2}\ \wedge \\
\sigma'_{i1} \sim_{\mathcal{V}} \sigma'_{i2}\ \wedge \\
\varphi'_{i1} \sim_{\mathcal{V}} \varphi'_{i2}
\end{array}$$

The bisimulation follows the approach of Sabelfeld and Sands [SS00] in utilising a *resetting of the state* between subtraces. This follows from modelling the attacker's ability to modify signals concurrently with the execution. Furthermore it accommodates the dynamically changing nature of the security policies due to the fulfilment of constraints, as seen in [MS04]. The definition is transitive but not reflexive. That the definition is not reflexive follows from observing that the program

```
p : process begin
  l := h;
end process p
```

is not self-similar whenever information is not permitted to flow from h to l.

Fact 2. *If $G \leq G'$ and $\mathcal{R}$ is a $(G, \mathcal{V})$-bisimulation then $\mathcal{R}$ is also a $(G', \mathcal{V})$-bisimulation.*

Definition 3.11 A G-bisimulation is a relation $\mathcal{R}$ such that for all $\mathcal{V}$, $\mathcal{R}$ is a $(G, \mathcal{V})$-bisimulation. Denote the largest G-bisimulation $\approx_G$.

Now we can define the security condition as a program being bisimilar to itself.

Definition 3.12 (History-based Release) A program $\|_{i \in I}\ css_i$ is secure wrt. the security policy G if and only if we have $\|_{i \in I}\ css_i \ \approx_G \|_{i \in I}\ css_i$.

Example 3.2 *In the following we will consider a number of example programs that illustrate the strength of History-based Release. First consider the program*

```
p : process begin
  y <= x;
end process p
```

which reads the signal x and modifies y. With the policy $\underline{x} \overset{p}{\rightsquigarrow} \underline{y}$, the program is secure, while changing the policy to $\underline{x} \overset{p'}{\rightsquigarrow} \underline{y}$ makes the program insecure. The reason that the program is insecure with the second policy is because the bisimulation forces us to consider all possible traces, so even if the above program was modified to execute a process named p' concurrently with the considered one, p, the result would be the same. This corresponds to intransitive non-interference *[MB05] (or* lexically scoped flows *according to [BS06]).*

Example 3.3 *History-based Release goes beyond lexically scoped flows as the policy might constrain the history of a process. This is illustrated by the following example. Consider the security policy in Fig. 3.1(c) and assume that* $\underline{x} = v_1$, $\underline{y} = v_2$ *and* $\underline{z} = v_3$, *for which the program*

```
i : process begin
  y <= x;
end process i

i' : process begin
  z <= y;
end process i'
```

is secure. On the other hand the program

```
i : process begin
  y <= x;
end process i

i' : process begin
  z <= x;
end process i'
```

is insecure because the process i' *might evaluate prior to the process* i.

Example 3.4 *Another concern is the handling of indirect flows. Consider the program*

```
p : process begin
  if x then y <= not y else null
end process p
```

and an attacker observing whether the signal y *is modified or not. Based on this the attacker will know when the the signal* x *carries the value '1'. Therefore History-based Release allows the program for the policy* $\underline{x} \overset{\underline{p}}{\rightsquigarrow} \underline{y}$, *but not for* $\underline{x} \overset{false}{\rightsquigarrow} \underline{y}$.

Example 3.5 *Finally we wish to look at an example program that is insecure in the traditional setting where lattices are used as security policies. Consider the program*

```
p : process begin
  y := '0';
  z := y;
  y := x;
end process p
```

which is secure for the policy in Fig. 3.1(a) when $\underline{x} = v_1$, $\underline{y} = v_2$ *and* $\underline{z} = v_3$. *This is because evaluating the program does not result in information flowing from* x *to* z.

3.2.2 Consistency of History-based Release

In this subsection we argue the consistency of the definition of History-based Release. In particular we will discuss two of the principles presented by Sabelfeld and Sands in [SS05]. In the following we consider declassification to refer to constraints that are not trivially evaluated to *true* or *false*.

Conservativity: *Security for programs with no declassification is equivalent to non-interference.*

Limiting all constraints on edges in the flow graphs to only being of the simple form *true*, *false* or l gives us intransitive non-interference. Removing all non-trivial constraints (i.e. only having the constraints *true* and *false*) results in traditional non-interference.

Monotonicity of release: *Adding further declassifications to a secure program cannot render it insecure.*

Adding declassifications to a program corresponds to making our security policies less restrictive. Hence we aim to show that a program will be secure for any policy at least as restrictive as the original policy, for which it can be shown secure.

Lemma 3.13 (Monotonicity) If $G \leq G'$ then

$$\|_{i \in I}\ css_{i1} \ \approx_G \|_{i \in I}\ css_{i2}$$

implies

$$\|_{i \in I}\ css_{i1} \ \approx_{G'} \|_{i \in I}\ css_{i2}.$$

Proof. It follows from Fact 1 that

$$\forall i \in I : (\sigma_{i1} \sim_{\nabla(\mathcal{V},G',\omega)} \sigma_{i2}) \Rightarrow (\sigma_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \sigma_{i2})$$

and

$$\forall i \in I : (\varphi_{i1} \sim_{\nabla(\mathcal{V},G',\omega)} \varphi_{i2}) \Rightarrow (\varphi_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \varphi_{i2}).$$

The Lemma follows from Fact 2 and observing that to show $\|_{i\in I}\ css_{i1} \approx_{G'} \|_{i\in I}\ css_{i2}$ we have either $\forall i : \sigma_{i1} \sim_{\nabla(\mathcal{V},G',\omega)} \sigma_{i2}$ and $\forall i : \varphi_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \varphi_{i2}$, in which case we can reuse the proof for $\|_{i\in I}\ css_{i1} \approx_G \|_{i\in I}\ css_{i2}$, and otherwise the result holds trivially. □

3.3 Transitive Security Policies

In this section we shall investigate a limited version of the locality-based security policies. The investigation will focus on transitively closed policies and compare them to the previously presented policies. The goal of the transitively closed policies is to find a subset of the locality-based policies that allows for a more direct approach to automatic analysis and verification of program security. The transitively closed policies will be utilised in the correctness result for the type based approach to verification of CoreVHDL programs presented in Chapter 4.

First let us define what a transitively closed Locality-based security policy is. The intuition is that if information is allowed to flow from one node to a second node and from the second node to a third node then information should also be allowed to flow from the first node to the third node. Formally this is defined in the following definition.

Definition 3.14 (Transitively Closed Locality-based Security Policies) A security policy G is transitively closed whenever

$$\forall\ n_0 \overset{\delta_1}{\rightsquigarrow} n_1 \overset{\delta_2}{\rightsquigarrow} n_2 \overset{\delta_3}{\rightsquigarrow} \cdots \overset{\delta_k}{\rightsquigarrow} n_k\ \in G$$

implies

$$n_0 \overset{\delta}{\rightsquigarrow} n_k\ \in G \wedge$$
$$[\![\delta_1 \cdot \delta_2 \cdots \delta_k]\!] \subseteq [\![\delta]\!]$$

Observe that this goes against our previous comments on information flows between principals, who are not allowed to forward each others information without explicit permission from the originating principal.

The transitively closed policies merely propose a restriction on the policies, and hence all previously presented results still hold. In particular we will apply Lemma 3.13 to give guarantees for policies that are not transitively closed whenever our analysed system can be verified to conform to History-based Release for a more restrictive transitively closed policy.

The goal of the transitively closed policies was to accommodate a simplified and more direct approach to analysing the considered programs. Hence we propose a simplified security condition, which match the intuition of the restricted policies. Furthermore for transitive closed policies, the condition will provide a guarantee that is equally strong to History-based Release. The condition will be centred around the following bisimulation.

Definition 3.15 (Transitive Bisimulation) A $(G, \mathcal{V})$-bisimulation is a symmetric relation $\mathcal{R}$ on programs whenever

$$\begin{array}{l} \|_{i\in I}\ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle \stackrel{\omega}{\Longrightarrow} \|_{i\in I}\ \langle css_{i1}{}'', \sigma'_{i1}, \varphi'_{i1}\rangle\ \wedge \\ \|_{i\in I}\ css'_{i1}\ \mathcal{R}\ \|_{i\in I}\ css'_{i2}\ \wedge \\ \sigma_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \sigma_{i2}\ \wedge \\ \varphi_{i1} \sim_{\nabla(\mathcal{V},G,\omega)} \varphi_{i2} \end{array}$$

then there exists $\|_{i\in I}\ css''_{i2}$, σ'_{i2}, φ'_{i2} and ω' such that

$$\begin{array}{l} \|_{i\in I}\ \langle css'_{i2}, \sigma_{i2}, \varphi_{i2}\rangle \stackrel{\omega'}{\Longrightarrow}{}^{*}\ \|_{i\in I}\ \langle css_{i2}{}'', \sigma'_{i2}, \varphi'_{i2}\rangle\ \wedge \\ \|_{i\in I}\ css''_{i1}\ \mathcal{R}\ \|_{i\in I}\ css''_{i2}\ \wedge \\ \sigma'_{i1} \sim_{\mathcal{V}} \sigma'_{i2}\ \wedge \\ \varphi'_{i1} \sim_{\mathcal{V}} \varphi'_{i2} \end{array}$$

Here we have limited the bisimulation game to one step evaluations of the first program. This will simplify the proof burden on programs under verification and aid us in establishing the correctness result of the static analyses presented in the following chapters.

Again the bisimulation follows the approach of Sabelfeld and Sands [SS00] in utilising a *resetting of the state* between subtraces. This follows from modelling the attacker's ability to modify signals concurrently with the execution. The definition is transitive but not reflexive.

Definition 3.16 A transitive G-bisimulation is a relation $\mathcal{R}$ such that for all $\mathcal{V}$, $\mathcal{R}$ is a transitive $(G, \mathcal{V})$-bisimulation. Denote the largest transitive G-bisimulation $\approx_G^+$.

Now we can define the transitive security condition as a program being bisimilar to itself.

Definition 3.17 (Transitive History-based Release) A program $\|_{i\in I}\ css_i$ is secure wrt. the security policy G if and only if we have $\|_{i\in I}\ css_i \ \approx_G^+ \|_{i\in I}\ css_i$.

One should observe that the temporal properties of having information flows that are due to collaboration of several principals are lost. This is due to the resetting of the state between execution steps. Hence the Transitive History-based Release condition is a weakened condition compared to History-based Release. We also observe that for a transitively closed security policies the two conditions coincide and are equally strong. We investigate this further in the following two lemmas. The first lemma states that History-based Release and Transitive History-based Release are equally strong for a transitively closed policy.

Lemma 3.18 If a program conforms with *Transitive History-based Release* for a *Transitively Closed Policy* G then the program also conforms with *History-based Release* for G.

Proof. The proof proceeds by induction in the length of the evaluation sequence of the program $\|_{i\in I}\ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle$. Consider the evaluation sequence

$$\|_{i\in I}\ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle \stackrel{\omega}{\Longrightarrow}^k \ \|_{i\in I}\ \langle css_{i1}{}'', \sigma'_{i1}, \varphi'_{i1}\rangle$$

If $k = 0$ the result follows vacuously. For the induction step we assume that the result holds for $k \leq k_0$ and we shall prove it for $k_0 + 1$. Divide the evaluation sequence into the first step and the rest. Applying Definition 3.15 gives us that the program conforms with *Transitive History-based Release* after evaluating one step. The remaining evaluation sequence is shorter, hence we apply the induction hypothesis and get that the last steps conform with *History-based Release*. The evaluation sequence has the form

$$\|_{i\in I}\ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle \stackrel{\omega'}{\Longrightarrow} \|_{i\in I}\ \langle css_{i1}{}''', \sigma''_{i1}, \varphi''_{i1}\rangle \stackrel{\omega''}{\Longrightarrow}^{k_0} \ \|_{i\in I}\ \langle css_{i1}{}'', \sigma'_{i1}, \varphi'_{i1}\rangle$$

Thus we need to consider the changes of the states. If a variable or signal has changed its value in the final state, one of the following cases must hold; first if

the variable or signal is unobservable, the result follows. Second if the variable or signal is observable then we know that any differences between the resulting states are due to the first step modifying the states used in the rest of the evaluation. Now as the policy is a *Transitively Closed Policy* we have that there exists an edge annotated with a constraint δ such that $\underline{\underline{\omega'}} \cdot \underline{\underline{\omega''}} \in [\![\delta]\!]$ and hence there can be no difference in the states that are observable. □

The second lemma state that there exist programs that are secure by Transitive History-based Release and insecure by History-based Release with respect to a (non-transitive) policy.

Lemma 3.19 If a program conforms with *Transitive History-based Release* for a policy G that is not transitively closed then the program might not conform with *History-based Release* for G.

Proof. The proof proceeds by counter example. The first program presented in Example 3.3 conforms with Transitive History-based Release for the policy given in Figure 3.1(a). However it does not conform with History-based Release for the same policy. □

3.4 Discussion

Traditionally, policies for information flow security have been of the form of security lattices [BL73, Den76] where an underlying hierarchical structure on the principals in the system is assumed and reflected in the security lattice. Hence the principals are tied to security levels and an ordering of security levels indicate what information is observable to a principal. Security lattices have found a widespread usage in language-based approaches to information flow security, see e.g. [VSI96, SM03a].

The security policies presented in this chapter are base on labelled graphs, i.e. without assigning an underlying ordering. Due to the lack of underlying ordering the expressiveness of the policies is increased, allowing for simplified specifications of security policies for systems. One example is systems that let a principal act on resources in different security domains without causing a flow of information in between. The expressiveness gained is due to the lack of the transitive nature of the ordering in a lattice. These policies relate back to resource matrices applied for e.g. covert channel identification [Kem82, McH95].

A comparison [HKMY86] on the verification of a real systems, the Honeywell Secure Ada Target (SAT), using the non-interference approach [GM82] and the shared resource matrix approach [Kem82] highlighted issues with the transitivity of the non-interference property. Further investigation illustrated that information flow policies are in general not transitive [HY86, Rus92].

Clearly the translation of a policy specified as a lattice to a labelled graph is straightforward. For each security level a security domain is introduced. Edges (labelled *true*) are added between security domains according to the ordering of the corresponding security levels.

The *Decentralized Label Model* by Myers and Liskov [ML97, Mye99a] is a framework for security policies that allows owners of information to specify who is permitted to read that information. The basis model is still a lattice and does not provide expressiveness similar to what is presented in this chapter.

3.4.1 Semantical security conditions

The goal of specifying whether a system complies with an information flow policy has been formally stated as *non-interference* [Coh77, GM82]. Informally non-interference states that for all input to a system, that varies only on information not observable to the attacker, the resulting output will only vary on information not observable to the attacker. Since the original formalisation several generalised non-interference properties have been published. In Section 3.2.2 we showed that History-based Release generalises non-interference, so in this Section other properties will be discussed and compared to History-based Release.

Non-Disclosure by Matos and Boudol [MB05] proposes extending the syntax of a ML-like calculus with specific constructs for loosening the security policy. These constructs have the form

$$\mathsf{flow}\ A \prec B\ \mathsf{in}\ M$$

where M is an expression and A and B are security levels. The construct extends the security relation to permit information to flow from A to B in M and thereby permits *disclosure* of confidential information in lexically scoped parts of programs. The policies presented in this chapter allow for flows to be scoped within specified locations, i.e. locations associated with a security domain. Clearly, by introducing a security domain tied to a *fresh* location for each flow construct and constraining the information flow to only happen in

execution traces containing the security location we get scoped flows. Finally due to the transitive nature of underlying lattice structure in [MB05], we need to perform a transitive closure on the resulting graph to achieve the same effect in our policies. In this manner the *Non-Disclosure* property can be seen as a specialisation of the History-based Release property. Obviously the *Non-Disclosure* property does not have the expressiveness to handle constraints on execution traces.

Intransitive non-interference. Goguen and Meseguer [GM84] generalised non-interference to *intransitive non-interference* while observing that information flows might be permitted when properly filtered at specified security levels. The property was further investigated in [HY86, Rus92] and adapted to a language-based setting by Mantel and Sands [MS04]. Mantel and Sands [MS04] formalise intransitive non-interference so that two goals are achieved. First, the place in the program where information flow is limited through a syntactical extension of the language. Second the security level where information flows through is specified through an extension of the security lattice by an intransitive component.

History-based Release incorporates these concerns. The place in the program where information flow is guaranteed in the same way as described above for the non-disclosure property. Furthermore the locality-based security policies are intransitive due to being based on graphs rather than lattices.

Non-interference until by Chong and Myers [CM04] propose adding conditions to security policies based on lattices. This is done by introducing a special annotated flow of the form $\ell_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} \ell_k$ into the security policies, which states that information can be gradually downgraded along with the fulfilment of the conditions $c_1, \ldots, c_k$. It is straightforward to represent the downgrading condition with History-based Release.

However, observe that once the conditions are fulfilled, information can flow directly from ℓ_0 to any of the security levels $\ell_1, \ldots, \ell_k$. Therefore *non-interference until* does not provide the intransitive guarantees of History-based Release. Another point is the temporal constraints that History-based Release enforce on execution traces. *Non-interference until* provides simple temporal guarantees, namely that conditions are fulfilled prior to downgrading. However neither the order of the fulfilment nor the conditions allow for temporal constraints.

Chong and Myers [CM05] extend the policies to consider erasure. This extension go beyond the scope of this chapter. Yet, one can consider the erasure policies

to be lifetime partitionings of variables and signals, where the program must guarantee that the variable or signal that is to be erased is overwritten along with the fulfilment of the constraints in the security policy. Chapter 5 present further investigation and discussion of of partitioning of security domains.

Flow Locks. Recently Broberg and Sands [BS06] introduced the novel concept of *flow locks* as a framework for dynamic information flow policies. The policies are specified in syntactical constructs introduced in an ML-like calculus. The constructs are utilised in the opening and closing of flow locks, these locks are used in constraining the information flows in the policies. The policies have the form $\{\sigma_1 \Rightarrow A_1; \ldots; \sigma_n \Rightarrow A_n\}$ and are used to annotate declarations of variables in the programs. These policies correspond to ours, where a policy needs to specify where information might flow globally during execution and is hence not transitively closed. The major difference is that our policies can include temporal constraints which cannot be expressed in the flow locks policies.

Another major difference between [BS06] and the present work is the intuition behind the security condition. In the flow lock setting information can flow to security levels as specified in the policies, as long as necessary locks are opened beforehand. This differs from our definition in not being tied to the actual flow of information. E.g. once a lock is open information can flow from several levels and several times. Furthermore flow locks have no way of observing if a lock has been closed and opened again between two flows. In our setting the constraints on the execution trace must be fulfilled for every single flow, hence it is not sufficient that another process is executed at the right location, just before or after considered flow.

Chapter 4

Information Flow Analysis

It's impossible to move, to live, to operate at any level without leaving traces, bits, seemingly meaningless fragments of personal information
— William Gibson

A main concern in computer security is *confidentiality*, the assurance that information is only available to authorised principals. The first approaches to enforcing confidentiality were *access control* mechanisms granting access to information only to authorised principals. Often *discretionary access control* mechanisms will not track the usage of information after access clearance is granted, and hence trusts the principal (or process acting for the principal) to propagate the information in a confidentiality preserving manner [Lam73]. This has lead to an increased attention on mechanisms for *mandatory access control* [BL73] and *information flow analysis* [Den76]. While dynamic approaches enforcing information flow control has been proposed [DOD85] their approaches to determining dependencies, including implicit ones, often make them overly restrictive and impractical [Mye99b]. Thus much investigation of static analysis for information flow control has been done. Volpano et al. [VSI96] developed an automatic static analysis and showed the coupling to the security property of non-interference. This resulted in numerous investigations of static information flow analysis for *real* programming languages, e.g. [Mye99a] and [HR98], concurrency, e.g. [SV98] and [SS01], covert channels, e.g. [Aga00], and declassification, e.g. [ML97], [ZM01] and [SM03b] — for a thorough overview see [SM03a].

This chapter develops a static information flow analysis for CoreVHDL. The approach taken is that of Kemmerer [Kem82, Kem02], where the identification of local dependencies and the end-to-end dependencies are separated and computed in two steps. This is different from the approach taken in [VSI96, Mye99a] (and others [SM03a]) where the transitivity of the underlying lattice-based security policies makes checking of local dependencies sufficient. Taking this view we develop the analysis and prove that it enforces the security properties presented in Chapter 3. Hence the analysis presented in this chapter is *flow insensitive* meaning that the execution order is not taken into account by the analysis [NNH99]. In Chapter 5 we will investigate a *flow sensitive* analysis in order to strengthen the information flow analysis by guiding the transitive closure through the removal of spurious control flows.

4.1 Local dependencies in CoreVHDL

The Information Flow analysis is performed in two steps. First we identify the flow of information to a variable or signal locally at each assignment; this is specified in this section. However instead of performing a transitive closure of this information, as specified by Kemmerer [Kem82], we base our correctness result on the transitively closed security policies and the matching property, transitive History-based Release.

The result of the Information Flow analysis is given in the form of a directed graph. The graph has a node for each variable or signal used in the program, and an edge from the node n_1 to the node n_2 if information *might* flow from n_1 to n_2 in the program.

The analysis is based on an understanding of which statements cause information to flow between signals. It is clear that an assignment of a variable to another variable will cause a flow of information. As an example,

$$\texttt{a} := \texttt{b}$$

causes a flow of information from `b` to `a`. We also need to consider implicit flows due to conditional statements. As an example,

$$\texttt{if c then a} := \texttt{b else null}$$

has an implicit flow from `c` to `a` because an observer could use the resulting value of `a` to gain information about the value of `c`. Observe that the presented security

properties does not capture information leaking through covert channels. Thus we here ignore issues regarding e.g. termination and timing of the statements. These issues will be discussed in detail in Chapter 5.

In the presented analyses we are going to use a labelling scheme. The labelling scheme is defined so that each block has a label which is initially unique for the program. During execution the labels might not be unique within the processes, but the same label is not found in two different processes. Hence, we shall sometimes implicitly use that to each label ($l \in Lab$) there is a unique process identifier ($i \in Id$) in which it occurs. In fact we shall freely use labels instead of process identifiers. The syntax of statements with labelled blocks

$$\begin{array}{rcl} ss & ::= & [\texttt{null}]^l \mid ss_1; ss_2 \mid \texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2 \\ & & \mid [x := e]^l \mid [s <= e]^l \mid [\texttt{wait on } S \texttt{ until } e]^l \end{array}$$

Often we will annotate the evaluation with the structural operational semantics of a statement by the label of the block that has been evaluated. Hence we write

$$\langle ss, \sigma, \varphi \rangle \stackrel{l}{\Rightarrow} \langle ss', \sigma', \varphi' \rangle$$

when the block labelled l is evaluated.

Analysing a CoreVHDL program is done by checking that neither *explicit* nor *implicit* flow are present. In this fashion we must consider all the statements of CoreVHDL and determine how information might flow. For a CoreVHDL program we define a set of structural rules that define the set of dependencies between local variables and signals. The analysis is defined using judgements of the form

$$B \vdash ss : RM$$

where $B \subseteq (\mathbf{Var} \cup \mathbf{Sig})$, $ss \in \mathbf{Stmt}$ and $RM \subseteq ((\mathbf{Var} \cup \mathbf{Sig}) \times \mathbf{Lab} \times \{M_0, M_1, R_0, R_1\})$. Here ss is the statement analysed under the assumption that it is only reachable when values of variables and signals in B have certain values. The result is the set RM containing entries (n, l, M_j) if the variable or signal n might be modified at label l in ss; we use M_0 for variables and present values of signals and M_1 for active values of signals. Similarly, RM contains entries (n, l, R_j) if the variable or signal n might be read at label l in ss; we use R_0 for variables and present values of signals and R_1 for the synchronisation of active values of signals.

The local dependency analysis of the flow between variables and signals is specified in Table 4.1 and is explained below. Assignments to variables result in local

[Local Variable Assignment] :
$B \vdash [x := e]^l : \quad \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$

[Signal Assignment] :
$B \vdash [s <= e]^l : \quad \{(s, l, M_1)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$

[Skip] :
$B \vdash [\texttt{null}]^l : \emptyset$

[Composition] :
$$\frac{B \vdash ss_1 : RM_1 \qquad B \vdash ss_2 : RM_2}{B \vdash ss_1; ss_2 : RM_1 \cup RM_2}$$

[Conditional] :
$$\frac{B' \vdash ss_1 : RM_1 \qquad B' \vdash ss_2 : RM_2}{B \vdash \texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2 : RM_1 \cup RM_2}$$
where $B' = B \cup FV(e) \cup FS(e)$

[Synchronization] :
$B \vdash [\texttt{wait on } S \texttt{ until } e]^l : \{(s, l, R_1) \mid s \in FS(ss_i)\} \cup$
$\{(n, l, R_0) \mid n \in B \cup S \cup FV(e) \cup FS(e)\}$
where ss_i is the body of process i in which l resides

Table 4.1: Structural rules for constructing a Resource Matrix for the process i : `process` $decl_i$ `begin` ss_i; `end process` i

dependencies, there are no other statements that causes information to flow into variables.

Observe that for the active signals in a program information can only flow to the signal through signal assignment. Hence only the signal assignment contributes dependencies to the resulting set. Notice that the information flowing to active signals (M_1) might come from both local variables and the present value of signals, but never from the active value of signals.

The variables and signals used in the evaluation of conditions within `if` statements are collected in the *block-set* B as they might implicitly flow into assigned variables or signals in the branches. This is taken care of in rule [**Conditional**]. Again notice that this rule does not handle timing channels that might occur, timing channels will be discussed in Chapter 6.

The synchronisation statement (i.e. `wait`) causes information about the active signals to flow to the present value of the same signals. Hence we will update

(writing R_1) all signals present in the process considered.

4.2 Soundness

Before showing that the analysis provides the necessary static guarantee for the security properties presented in Chapter 3 the typical soundness result of *subject reduction* [NNH99] is shown. This result give some insight in the technical details of semantical soundness for CoreVHDL as well as serving as a sanity check on the specification of the type system. When stating a subject reduction result it is imperative to determine a relation between typing environments before and after a semantical evaluation of a considered specification. In this matter the issues of the block component B for tracking implicit flows in the static analysis is seen to be neither covariant nor contravariant. This is seen by observing that when analysing a conditional statement the block component is extended with the variables and signals branched on. However, when leaving a branch the block components is reduced accordingly. To overcome this issue the semantics is instrumented with a similar component for tracking the branching context. The instrumented semantics is defined on configurations of the form

$$B \vdash \langle ss^B, \sigma, \varphi \rangle \in (Var \cup Sig) \times Stmt^B \times State \times Signals$$

where $Stmt^B$ is an extension of the syntactical category for statements including the special construct $[B]ss^B$ that impose a lexically scoped block component on the statement ss^B. The $Stmt'$ category is extended in the same manner to $Stmt^{B\prime}$.

The instrumented semantics for CoreVHDL is presented in Table 4.2 where we include the locality labels and the context in the notation of the statements. The instrumented semantics introduce two new rules, [**Branch**] for evaluating statements within a context. Notice that the instrumented semantics is limited to evaluate specifications that do not contain wait statements nested within conditionals.

Observe that the block component in the instrumented semantics collect precise information, opposite to the block component in the type system. Furthermore the block component has no influence on the evaluation of the statement.

Lemma 4.1 If $ss = \lceil ss^B \rceil_B$ and $ss' = \lceil ss^{B\prime} \rceil_B$ then

$$\langle ss, \sigma, \varphi \rangle \Rightarrow \langle ss', \sigma', \varphi' \rangle \text{ iff. } B \vdash \langle ss^B, \sigma, \varphi \rangle \Rightarrow \langle ss^{B\prime}, \sigma', \varphi' \rangle$$

[**Local Variable Assignment**] :
$B \vdash \langle [x := e]^l, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto v], \varphi\rangle$ where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = v$

[**Signal Assignment**] :
$B \vdash \langle [s <= e]^l, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, 1^{[s]}[v \mapsto\rangle]$ where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = v$

[**Skip**] :
$B \vdash \langle [\texttt{null}]^l, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi\rangle$

[**Composition**] :
$$\frac{B \vdash \langle ss_1^B, \sigma, \varphi\rangle \Rightarrow \langle ss_1^{B\prime}, \sigma', \varphi'\rangle}{B \vdash \langle ss_1^B; ss_2^B, \sigma, \varphi\rangle \Rightarrow \langle ss_1^{B\prime}; ss_2^B, \sigma', \varphi'\rangle} \text{ where } ss_1^{B\prime} \in Stmt^B$$

$$\frac{B \vdash \langle ss_1^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{B \vdash \langle ss_1^B; ss_2^B, \sigma, \varphi\rangle \Rightarrow \langle ss_2^B, \sigma', \varphi'\rangle}$$

[**Conditional**] :
$B \vdash \langle \texttt{if } [e]^l \texttt{ then } ss_1^B \texttt{ else } ss_2^B, \varphi\rangle \Rightarrow \langle [FV(e) \cup FS(e)]ss_1^B, \sigma, \varphi\rangle$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle =$ '1'
$B \vdash \langle \texttt{if } [e]^l \texttt{ then } ss_1^B \texttt{ else } ss_2^B, \varphi\rangle \Rightarrow \langle [FV(e) \cup FS(e)]ss_2^B, \sigma, \varphi\rangle$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle =$ '0'

[**Branch**] :
$$\frac{B \cup \tilde{B} \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle ss^{B\prime}, \sigma', \varphi'\rangle}{B \vdash \langle [\tilde{B}]ss^B, \sigma, \varphi\rangle \Rightarrow \langle [\tilde{B}]ss^{B\prime}, \sigma', \varphi'\rangle}$$

$$\frac{B \cup \tilde{B} \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{B \vdash \langle [\tilde{B}]ss^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}$$

Table 4.2: Statements

The function $\lceil\cdot\rceil_B$ removes locally scoped block components.

Proof. The proof proceeds by a two way induction in the shape of the derivation tree and follows by simply inspecting the semantical rules and applying the induction hypothesis. □

To handle the new syntactical construct introduced above we add the following

two rules to the Local Dependencies Analysis for CoreVHDL

$$B \vdash \texttt{final} : \emptyset \qquad \frac{B \cup \tilde{B} \vdash ss^B : RM}{B \vdash [\tilde{B}]ss^B : RM}$$

The instrumented semantics allows us to keep the block component invariant during evaluation of specifications. Therefore we are now ready to show the subject reduction result.

Lemma 4.2 For every statement ss^B of CoreVHDL, where the Semantics changes the state in one step as $B \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle ss^{B\prime}, \sigma', \varphi'\rangle$, we have

$$B \vdash ss^B : RM \Rightarrow \begin{cases} \exists B' : \exists RM' : \\ \quad B' \vdash ss^{B\prime} : RM' \\ \quad \wedge\ RM' \subseteq RM \\ \quad \wedge\ B' = B \end{cases}$$

Proof. The proof proceeds by induction on the shape of the derivation tree for the Local Dependencies analysis on statement ss^B.

Case $[x := e]^l$**:** For assignment the Semantics evaluate the statement as

$$B \vdash \langle [x := e]^l, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto v], \varphi\rangle \text{ where } \mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = v$$

and the Local Dependencies analysis gives

$$B \vdash [x := e]^l : RM$$

where $RM = \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$ according to Table 4.1. Now choose $B = B'$. Then applying the analysis to the statement after evaluating one step of the Semantics

$$B' \vdash \texttt{final} : RM' = \emptyset$$

Now it follows

$$RM' \subseteq RM$$

Case $[s <= e]^l$**:** Straightforward as above.

Case $[\texttt{null}]^l$**:** Evaluating a null statement in the Semantics gives

$$B \vdash \langle [\texttt{null}]^l, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi \rangle$$

and the result of the analysis is

$$B \vdash [\texttt{null}]^l : RM$$

where $RM = \emptyset$. Again we choose $B = B'$, for which applying the analysis gives

$$B' \vdash \texttt{final} : RM' = \emptyset$$

From which it follows that

$$RM' \subseteq RM$$

Case $ss_1^B; ss_2^B$**:** There are two possibilities for the Semantics, we consider them separately. First we consider the case where ss_1^B is not evaluated in a single step

$$\frac{B \vdash \langle ss_1^B, \sigma, \varphi \rangle \Rightarrow \langle ss_1^{B\prime}, \sigma', \varphi' \rangle}{B \vdash \langle ss_1^B; ss_2^B, \sigma, \varphi \rangle \Rightarrow \langle ss_1^{B\prime}; ss_2^B, \sigma', \varphi' \rangle} \text{ where } ss_1^{B\prime} \in Stmt^B$$

So the result of the analysis when applied to the original statement

$$\frac{B \vdash ss_1^B : RM_1 \qquad B \vdash ss_2^B : RM_2}{B \vdash ss_1^B; ss_2^B : RM_1 \cup RM_2}$$

we apply the induction hypothesis on $B \vdash \langle ss_1^B, \sigma, \varphi \rangle \Rightarrow \langle ss_1^{B\prime}, \sigma', \varphi' \rangle$ which gives

$$B \vdash ss_1^B : RM_1 \Rightarrow \begin{cases} \exists B_1' : \exists RM_1' : \\ B_1' \vdash ss_1^{B\prime} : RM_1' \\ \wedge\ RM_1' \subseteq RM_1 \\ \wedge\ B_1' = B \end{cases}$$

Now choose $B' = B_1'$. The result of applying the analysis after evaluating one step of the Semantics is

$$\frac{B' \vdash ss_1^{B\prime} : RM_1' \qquad B' \vdash ss_2^B : RM_2}{B' \vdash ss_1^{B\prime}; ss_2^B : RM_1' \cup RM_2}$$

and it follows

$$RM_1' \cup RM_2 \subseteq RM_1 \cup RM_2$$

Secondly we consider the case where ss_1^B is completely evaluated in a single step

$$\frac{B \vdash \langle ss_1^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{B \vdash \langle ss_1^B; ss_2^B, \sigma, \varphi\rangle \Rightarrow \langle ss_2^B, \sigma', \varphi'\rangle}$$

and for the original statement the analysis gives

$$\frac{B \vdash ss_1^B : RM_1 \qquad B \vdash ss_2^B : RM_2}{B \vdash ss_1^B; ss_2^B : RM_1 \cup RM_2}$$

Choose $B = B'$. Then applying the analysis gives

$$B' \vdash ss_2^B : RM'$$

where $RM' = RM_2$. Therefore it follows

$$RM' \subseteq RM_1 \cup RM_2$$

Case $\texttt{if } [e]^l \texttt{ then } ss_1^B \texttt{ else } ss_2^B$**:** For a given σ and φ we consider the cases where e evaluates to '1' and '0'.

If $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle =$ **'1':** One step of the Semantics give

$$B \vdash \langle \texttt{if } e \texttt{ then } ss_1^B \texttt{ else } ss_2^B, \sigma, \varphi\rangle \Rightarrow \langle [FV(e) \cup FS(e)]ss_1^B, \sigma, \varphi\rangle$$

Applying the analysis gives

$$\frac{B'' \vdash ss_1^B : RM_1 \qquad B'' \vdash ss_2^B : RM_2}{B \vdash \texttt{if } [e]^l \texttt{ then } ss_1^B \texttt{ else } ss_2^B : RM_1 \cup RM_2}$$

where $B'' = B \cup FV(e) \cup FS(e)$. We choose $B' = B$ and apply the analysis

$$\frac{B' \cup FV(e) \cup FS(e) \vdash ss_1^B : RM'}{B' \vdash [FV(e) \cup FS(e)]ss_1^B : RM'}$$

where $RM' = RM_1$, it follows that

$$RM' \subseteq RM_1 \cup RM_2$$

If $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle =$ **'0':** One step of the Semantics give

$$B \vdash \langle \texttt{if } e \texttt{ then } ss_1^B \texttt{ else } ss_2^B, \sigma, \varphi\rangle \Rightarrow \langle ss_2^B, \sigma, \varphi\rangle$$

Applying the analysis gives

$$\frac{B'' \vdash ss_1^B : RM_1 \qquad B'' \vdash ss_2^B : RM_2}{B \vdash \texttt{if } [e]^l \texttt{ then } ss_1^B \texttt{ else } ss_2^B : RM_1 \cup RM_2}$$

where $B'' = B \cup FV(e) \cup FS(e)$. We choose $B' = B$ and apply the analysis

$$\frac{B' \cup FV(e) \cup FS(e) \vdash ss_2^B : RM'}{B' \vdash [FV(e) \cup FS(e)]ss_2^B : RM'}$$

where $RM' = RM_2$ therefore

$$RM' \subseteq RM_1 \cup RM_2$$

Case $[\tilde{B}]ss^B$: There are two cases for evaluating the statement in the Semantics. First assume that the Semantics give

$$\frac{B \cup \tilde{B} \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle ss^{B\prime}, \sigma', \varphi'\rangle}{B \vdash \langle [\tilde{B}]ss^B, \sigma, \varphi\rangle \Rightarrow \langle [\tilde{B}]ss^{B\prime}, \sigma', \varphi'\rangle}$$

applying the analysis on the statement before taking one step in the Semantics gives us

$$\frac{B \cup \tilde{B} \vdash ss^B : RM}{B \vdash [\tilde{B}]ss^B : RM}$$

now we choose $B' = B$, then applying the analysis on the statement after the step in the Semantics gives us

$$\frac{B' \cup \tilde{B} \vdash ss^{B\prime} : RM'}{B' \vdash [\tilde{B}]ss^{B\prime} : RM'}$$

if we apply the Induction Hypothesis on $B \cup \tilde{B} \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle ss^{B\prime}, \sigma', \varphi'\rangle$ we get

$$B \cup \tilde{B} \vdash ss^B : RM \Rightarrow \begin{cases} \exists B'' : \exists RM' : \\ B'' \vdash ss^{B\prime} : RM' \\ \wedge\ RM' \subseteq RM \\ \wedge\ B'' = B \cup \tilde{B} \end{cases}$$

and therefore

$$RM' \subseteq RM$$

Now assume that the Semantics give

$$\frac{B \cup \tilde{B} \vdash \langle ss^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{B \vdash \langle [\tilde{B}]ss^B, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}$$

applying the analysis on the statement before taking one step in the Semantics gives us

$$\frac{B \cup \tilde{B} \vdash ss^B : RM}{B \vdash [\tilde{B}]ss^B : RM}$$

no we choose $B' = B$, then applying the analysis on the statement after the step in the Semantics gives us

$$B' \vdash \texttt{final} : \emptyset$$

clearly

$$\emptyset \subseteq RM$$

This proves the Lemma. □

Similarly we wish to prove the following Lemma for the correctness of CoreVHDL programs.

Lemma 4.3 For every set of concurrent statements *css* of CoreVHDL, where the Semantics changes the state in one step as $\|_{i\in I}\ \langle ss_i^B, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle ss_i^{B\prime}, \sigma_i', \varphi_i'\rangle$, we have

$$B_i \vdash ss_i^B : RM_i \Rightarrow \left\{ \begin{array}{l} \exists B_1', \ldots, B_n' : \exists RM_1', \ldots, RM_n' : \\ \quad B_i' \vdash ss_i^{B'} : RM_i' \\ \quad \wedge\ RM_i' \subseteq RM_i \\ \quad \wedge\ B_i' = B_i \end{array} \right.$$

Proof. The proof proceeds by inspection of the two cases in the Semantics of Concurrent statements.

Case Handle non-waiting processes (H): If none of the processes are waiting the Semantics evaluates one step as

$$\frac{\emptyset \vdash \langle ss_j^B, \sigma_j, \varphi_j\rangle \Rightarrow \langle ss_j^{B\prime}, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i^B; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle ss_i^{B\prime}; css_i, \sigma'_i, \varphi'_i\rangle}$$

where $ss_i^{B\prime} = ss_i^B$, $\sigma'_i = \sigma_i$ and $\varphi'_i = \varphi_i$ for all $i \neq j$. For the j^{th} statement the result follows from Lemma 4.2. For the i^{th} statement, i.e. $i \neq j$, $ss_i^B = ss_i^{B\prime}$, so assume

$$B_i \vdash ss_i^B : RM_i$$

now choose $B'_i = B_i$, then the analysis of $ss_i^{B\prime}$ gives

$$B'_i \vdash ss_i^{B\prime} : RM'_i$$

it follows that $RM'_i \subseteq RM_i$.

Case Active signals (A): If all processes are waiting the Semantics considers the active signals

$$\|_{i\in I}\ \langle \texttt{wait on}\ S_i\ \texttt{until}\ e_i; ss_i^B, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle ss_i^{B\prime}, \sigma_i, \varphi'_i\rangle$$

where $ss_i^{B\prime}$ is defined depending on

$$ss_i^{B\prime} = \begin{cases} ss_i^B & \text{if} \\ & ((\exists s \in S_i.\ \varphi_i\ s\ 0 \neq \varphi'_i\ s\ 0) \wedge \\ & \mathcal{E}[\![e_i]\!]\langle\sigma'_i, \varphi'_i\rangle =' 1') \\ \texttt{wait on}\ S_i\ \texttt{until}\ b_i; ss_i^B & \text{otherwise} \end{cases}$$

We consider the two cases in turn.

If $((\exists s \in S_i.\ \varphi_i\ s\ 0 \neq \varphi_i'\ s\ 0) \wedge \mathcal{E}[\![e_i]\!]\langle\sigma_i', \varphi_i'\rangle =' 1')$**:** The Semantics continues execution, i.e. $ss_i^{B\prime} = ss_i^B$. Assume that the analyses of the process before evaluating the wait statement is

$$\frac{B_i \vdash [\texttt{wait on}\ S\ \texttt{until}\ e]^l : RM_{i1} \qquad B_i \vdash ss_i^B : RM_{i2}}{B_i \vdash [\texttt{wait on}\ S\ \texttt{until}\ e]^l; ss_i^B : RM_{i1} \cup RM_{i2}}$$

where $RM_{i1} = \{(s, l, R_1) \mid s \in FS(ss_i^{B\prime\prime})\} \cup \{(n, l, R_0) \mid n \in B \cup FV(e) \cup FS(e)\}$ and $ss_i^{B\prime\prime}$ is the body of process i in which l resides. Choose $B_i' = B_i$ then

$$B_i' \vdash ss_i^B : RM_i'$$

where $RM_i' = RM_{i2}$. Now it follows

$$RM_i' \subseteq RM_{i1} \cup RM_{i2}$$

Otherwise: Trivial as $ss_i^{B\prime} = \texttt{wait on}\ S_i\ \texttt{until}\ e_i; ss_i^B$.

Case Processes (P): Holds vacuously. □

4.3 History-based Release

The main result in this chapter is to show that the developed information flow analysis provides static guarantees for the security property History-based Release for a considered Locality-based security policy. In the formal results we make the assumption that the security policies are transitively closed. This restriction and how to lift it will be discussed in detail in Chapter 5. Hence we aim to show that analysing a program with the above presented analysis result in a static guarantee that match the transitive History-based release property presented in Section 3.3. Thus it follows from Lemma 3.18 that for a transitively closed security policy the analysis guarantee History-based Release.

First we state when the result of the Information Flow Analysis of a program is secure wrt. a security policy. We need to do this because the analysis will not

reject insecure programs (as it is often done by other information flow analyses, e.g. [VSI96, SM03a]), instead we choose to analyse the flow of information in a program, and afterwards compare the result with the security policy. This is merely a technicality as the related analyses could most often be modified to infer the security types.

Definition 4.4 (Secure Analysis Result) A CoreVHDL program $\|_{i\in I}\ ss_i$ has a *secure analysis result* wrt. to a security policy (V, λ) if the result of the Information Flow analysis RM_{lo} satisfies

$$\forall n, n', l, i : \{(n, l, M_i), (n', l, R_0)\} \subseteq RM_{lo} \Rightarrow l \leq_\Delta \lambda(\underline{n'}, \underline{n})$$

and

$$\forall n, n', l : \{(n, l, R_1), (n', l, R_0)\} \subseteq RM_{lo} \Rightarrow l \leq_\Delta \lambda(\underline{n'}, \underline{n})$$

When verifying a program we sometimes need to consider conditionals that depend on variables and signals which are not observable to an attacker, in this case it is imperative that branches do not give away information about the result of evaluating the condition, thus resulting in an implicit flow of information. Therefore we consider the class of statements that have the property of never modifying the observable part of the states. We shall classify this class of statements as being *operationally unobservable*.

Definition 4.5 (Operationally unobservable statement) A statement ss is said to be *operationally unobservable* for an observer $\mathcal{V}$ if

$$\langle ss, \sigma, \varphi\rangle \Rightarrow^* \langle ss', \sigma', \varphi'\rangle$$

implies

$$\sigma \sim_\mathcal{V} \sigma' \text{ and } \varphi \sim_\mathcal{V} \varphi'$$

Observe that this definition allows the statement to contain assignments to observable variables and signals, as long as they are either never evaluated or guaranteed to result in the same states (e.g. $x := x$).

In the resource matrix used when analysing a statement we can gain information that allows us to determine whether the statement might modify the observable part of the states. Hence we say that a statement is *syntactically unobservable* when the resource matrix used by the analysis of the statement does not contain any modifications (M_0 or M_1) of an observable variable or signal.

Definition 4.6 (Syntactically unobservable statement) A statement ss is said to be *syntactically unobservable* for an observer $\mathcal{V}$ if

$$B \vdash ss : RM$$

implies

$$\{(n, l, M_i) \mid n \in \mathcal{V} \wedge l \in Lab(ss) \wedge i \in \{0, 1\}\} \cap RM = \emptyset$$

Now we can show that syntactically unobservable statements are also operationally unobservable. First we show that evaluating one step of the statement is not observable.

Lemma 4.7 If a statement ss is *syntactically unobservable* for $\mathcal{V}$ then we have

$$\langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle$$

implies

$$\sigma \sim_{\mathcal{V}} \sigma' \text{ and } \varphi \sim_{\mathcal{V}} \varphi'$$

Proof. The proof proceeds by induction in the shape of the analysis.

Case $[x := e]^l$**:** Analysing the statement gives

$$B \vdash [x := e]^l : RM$$

where $RM = \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$. Applying the semantics on the assignment gives

$$\langle [x := e]^l, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto v], \varphi\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma_1, \varphi_1\rangle = v$. Because the statement is *syntactically unobservable* it follows from Definition 4.6 that $x \notin \mathcal{V}$. Hence we have that $\sigma \sim_{\mathcal{V}} \sigma'$ and $\varphi \sim_{\mathcal{V}} \varphi'$ holds vacuously.

Case $[s <= e]^l$**:** Straightforward as above.

Case $[\texttt{null}]^l$**:** Holds trivially.

Case $ss; ss''$**:** The analysis of the statement gives

$$\frac{B \vdash ss_1 : RM_1 \qquad B \vdash ss_2 : RM_2}{B \vdash ss_1; ss_2 : RM_1 \cup RM_2}$$

Now RM_1 is a valid resource matrix for ss_1. Hence from Definition 4.6 and by observing that $RM_1 \subseteq RM_1 \cup RM_2$ we can conclude that ss_1 is syntactically unobservable for $\mathcal{V}$. Now assume that applying the semantics on the statement ss_1 gives

$$\langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle ss_1', \sigma', \varphi' \rangle$$

hence applying the induction hypothesis gives $\sigma \sim_{\mathcal{V}} \sigma'$ and $\varphi \sim_{\mathcal{V}} \varphi'$. We need to consider two cases. First if $ss_1 = \texttt{final}$ we can apply the rule

$$\frac{\langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi' \rangle}{\langle ss_1; ss_2, \sigma, \varphi \rangle \Rightarrow \langle ss_2, \sigma', \varphi' \rangle}$$

and the case follows. Similar if $ss_1 \neq \texttt{final}$ we can apply the rule

$$\frac{\langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle ss_1', \sigma', \varphi' \rangle}{\langle ss_1; ss_2, \sigma, \varphi \rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi' \rangle}$$

which concludes the proof of this case.

Case `if` $[e]^l$ `then` ss_1 `else` ss_2**:** Holds trivially as evaluating one step in the semantics does not modify the states. □

We can now show that evaluating any number of steps does not result in the attacker gaining information.

Lemma 4.8 . If a statement ss is *syntactically unobservable* for $\mathcal{V}$ then it is also *operationally unobservable* for $\mathcal{V}$.

Proof. The Lemma follows from the transitive reflexive closure of Lemma 4.7. □

To show the main result we can build a symmetric relation between statements. The binary relation is defined such that two statement when evaluated will

modify the same observable variables and signals. The relation is called $\mathcal{T}_{G,\mathcal{V}}$, and state that two statements can either be equal or have prefixes that are syntactically unobservable.

Definition 4.9 ($\mathcal{T}_{G,\mathcal{V}}$) Define $\mathcal{T}_{G,\mathcal{V}}$ as a relation on statement where we have that $ss_1 \ \mathcal{T}_{G,\mathcal{V}} \ ss_2$ if one of the following holds

- $ss_1 = ss_2$, or
- $ss_1 = \overline{ss}_1; ss_1'$ and $ss_2 = \overline{ss}_2; ss_2'$ where $\overline{ss}_1$ and $\overline{ss}_2$ are operationally unobservable for $\mathcal{V}$ and $ss_1' \ \mathcal{T}_{G,\mathcal{V}} \ ss_2'$, or
- ss_1 and ss_2 are operationally unobservable for $\mathcal{V}$.

Now we are ready to show that two statements, which are related by $\mathcal{T}_{G,\mathcal{V}}$ and are secure by Definition 4.4, will also be related during each step of computation.

Lemma 4.10 If two CoreVHDL statements ss_1 and ss_2 are *secure* wrt. G and $\mathcal{V}$, when analysed in context B and we have that

$$\begin{array}{l}\langle ss_1, \sigma_1, \varphi_1\rangle \overset{l}{\Rightarrow} \langle ss_1', \sigma_1', \varphi_1'\rangle \ \wedge \\ ss_1 \ \mathcal{T}_{G,\mathcal{V}} \ ss_2 \ \wedge \\ \sigma_1 \sim_{\nabla(\mathcal{V},G,l)} \sigma_2 \ \wedge \\ \varphi_1 \sim_{\nabla(\mathcal{V},G,l)} \varphi_2\end{array}$$

then there exists ss_2', σ_2' and φ_2' such that

$$\begin{array}{l}\langle ss_2, \sigma_2, \varphi_2\rangle \overset{l'}{\Rightarrow} \langle ss_2', \sigma_2', \varphi_2'\rangle \ \wedge \\ ss_1' \ \mathcal{T}_{G,\mathcal{V}} \ ss_2' \ \wedge \\ \sigma_1' \sim_{\mathcal{V}} \sigma_2' \ \wedge \\ \varphi_1' \sim_{\mathcal{V}} \varphi_2'\end{array}$$

Proof. We need to consider each of the cases following from the clauses of Definition 4.9. First consider the case where $ss_1 = \overline{ss}_1; ss$ and $\overline{ss}_1$ is operationally unobservable for $\mathcal{V}$. Assume that semantics evaluate $\overline{ss}_1$ as

$$\langle \overline{ss}_1, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1', \varphi_1'\rangle$$

Applying the semantical rule [**Composition**] gives us

$$\frac{\langle \overline{ss}_1, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1', \varphi_1'\rangle}{\langle \overline{ss}_1; ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss, \sigma_1', \varphi_1'\rangle}$$

We know from Definition 4.9 that $ss_2 = \overline{ss}_2; ss$, Now choose the evaluation sequence

$$\langle \overline{ss}_2, \sigma_2, \varphi_2 \rangle \Rightarrow^k \langle \texttt{final}, \sigma_2', \varphi_2' \rangle$$

and from Lemma 2.1 we get that

$$\langle \overline{ss}_2; ss, \sigma_2, \varphi_2 \rangle \Rightarrow^k \langle ss, \sigma_2', \varphi_2' \rangle$$

and the result follows from Definition 4.5.

Now, assume that the semantics did not evaluate $\overline{ss}_1$ completely in one step then we choose not to evaluate ss_2. Hence Definition 4.5 and observing that the relation between the resulting statements are preserved gives the result.

If ss_1 is operationally unobservable for $\mathcal{V}$, we execute one step as

$$\langle ss_1, \sigma_1, \varphi_1 \rangle \Rightarrow \langle ss_1', \sigma_1', \varphi_1' \rangle$$

now if $ss_1' = \texttt{final}$ then we evaluate ss_2 completely and the result follows from Definition 4.5, otherwise we do not apply the semantics on ss_2 and the result follows.

Finally we need to show the result when $ss_1 = ss_2$. The proof proceeds by induction in the shape of the derivation tree for the Structural Operational Semantics on the statement ss_1.

Case $[x := e]^l$**:** There are two subcases. First if $x \in \mathcal{V}$, we have that the semantics evaluate one step as

$$\langle [x := e]^l, \sigma_1, \varphi_1 \rangle \Rightarrow \langle [\texttt{final}, \sigma_1[x \mapsto v], \varphi_1 \rangle$$

where $\mathcal{E}[\![e]\!]\langle \sigma_1, \varphi_1 \rangle = v$. Applying the analysis gives us

$$B \vdash [x := e]^l : RM$$

where $RM = \{(x, l, M_0)\} \cup \{(n, l, R_0) \mid n \in FV(e) \cup FS(e) \cup B\}$. Because the statement is *secure* and following from Definition 3.9 and Fact 1 we know that

$$\mathcal{E}[\![e]\!]\langle \sigma_1, \varphi_1 \rangle = \mathcal{E}[\![e]\!]\langle \sigma_2, \varphi_2 \rangle$$

Therefore we get $\sigma_1 \sim_{\mathcal{V}} \sigma_2$, while $\varphi_1 \sim_{\mathcal{V}} \varphi_2$ and $\texttt{final}\ \mathcal{T}_{G,\mathcal{V}}\ \texttt{final}$ follows vacuously.

If $x \notin \mathcal{V}$ the case follow immediately by observing that evaluating one step in the semantics on both ss_1 and ss_2 in respective states fulfils the requirement directly.

Case $[s <= e]^l$**:** Straightforward as above.

Case $[\texttt{null}]^l$**:** Holds trivially.

Case $ss; ss''$**:** Evaluating one step in the semantics results in two cases first assume that the evaluation has the form

$$\frac{\langle ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_1', \sigma_1', \varphi_1'\rangle}{\langle ss; ss'', \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_1'; ss'', \sigma_1', \varphi_1'\rangle}$$

Now it follows from applying the induction hypothesis that there exists ss_2', σ_2' and φ_2' such that

$$\begin{array}{l}\langle ss, \sigma_2, \varphi_2\rangle \Rightarrow^* \langle ss_2', \sigma_2', \varphi_2'\rangle \wedge \\ ss_1'\ \mathcal{T}_{G,\mathcal{V}}\ ss_2' \wedge \\ \sigma_1' \sim_{\mathcal{V}} \sigma_2' \wedge \\ \varphi_1' \sim_{\mathcal{V}} \sigma_2'\end{array}$$

The case follows from observing that $ss_1'; ss''\ \mathcal{T}_{G,\mathcal{V}}\ ss_2'; ss''$.

Now assume that the semantics evaluate one step as

$$\frac{\langle ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1', \varphi_1'\rangle}{\langle ss; ss'', \sigma_1, \varphi_1\rangle \Rightarrow \langle ss'', \sigma_1', \varphi_1'\rangle}$$

Hence it follows from applying the induction hypothesis that there exists ss_2', σ_2' and φ_2' such that

$$\begin{array}{l}\langle ss, \sigma_2, \varphi_2\rangle \Rightarrow^* \langle ss_2', \sigma_2', \varphi_2'\rangle \wedge \\ \texttt{final}\ \mathcal{T}_{G,\mathcal{V}}\ ss_2' \wedge \\ \sigma_1' \sim_{\mathcal{V}} \sigma_2' \wedge \\ \varphi_1' \sim_{\mathcal{V}} \sigma_2'\end{array}$$

Definition 4.9 gives us that $ss_2' = \texttt{final}$ or ss_2' is operationally unobservable for $\mathcal{V}$. In the second case we apply the semantics as

$$\langle ss_2', \sigma_2', \varphi_2' \rangle \Rightarrow^* \langle \texttt{final}, \sigma_2'', \varphi_2'' \rangle$$

Where we have that $\sigma_2' \sim_{\mathcal{V}} \sigma_2''$ and $\varphi_2' \sim_{\mathcal{V}} \varphi_2''$. Now we know the $ss''\ \mathcal{T}_{G,\mathcal{V}}\ ss''$ and the case follows.

Case $\texttt{if}\ [e]^l\ \texttt{then}\ ss\ \texttt{else}\ ss'$**:** We apply the semantics as

$$\langle \texttt{if}\ [e]^l\ \texttt{then}\ ss\ \texttt{else}\ ss', \sigma_1, \varphi_1 \rangle \Rightarrow \langle ss_1', \sigma_1, \varphi_1 \rangle$$

We need to consider two cases. First if

$$\forall n : n \in FV(e) \cup FS(e) \Rightarrow \underline{n} \in \nabla(\mathcal{V}, G, l)$$

then we know that

$$\mathcal{E}[\![e]\!]\langle \sigma_1, \varphi_1 \rangle = \mathcal{E}[\![e]\!]\langle \sigma_2, \varphi_2 \rangle$$

and hence we can choose to apply the semantics as

$$\langle \texttt{if}\ [e]^l\ \texttt{then}\ ss\ \texttt{else}\ ss', \sigma_2, \varphi_2 \rangle \Rightarrow \langle ss_1', \sigma_2, \varphi_2 \rangle$$

and the result follows immediately.

However, if

$$\exists n : n \in FV(e) \cup FS(e) \wedge \underline{n} \notin \nabla(\mathcal{V}, G, l)$$

we know from Definition 4.4 and Lemma 4.8 that ss and ss' are operationally unobservable for $\mathcal{V}$ and hence $ss\ \mathcal{T}_{G,\mathcal{V}}\ ss'$ so no matter which branch we choose by applying the semantics the result holds. □

Finally we can show the main result. To do so we lift the relation $\mathcal{T}_{G,\mathcal{V}}$ to CoreVHDL programs.

Definition 4.11 ($\mathcal{U}_{G,\mathcal{V}}$) Define $\mathcal{U}_{G,\mathcal{V}}$ as a relation on programs where we have that $\|_{i \in I}\ css_{i1}\ \mathcal{U}_{G,\mathcal{V}}\ \|_{i \in I}\ css_{i2}$ if for all $i \in I$ one of the following holds

- $css_{i1} = css_{i2}$, or
- $css_{i1} = ss_1; css'_{1i}$ and $css_{i2} = ss_2; css'_{i2}$ where $\overline{ss}_1 \ \mathcal{T}_{U,\mathcal{V}} \ \overline{ss}_2$ and $css'_{i1} \ \mathcal{U}_{U,\mathcal{V}} \ css'_{i2}$.

Now, when two programs are analysed and considered secure, and each statement in the body of the processes can be related by $\mathcal{U}_{G,\mathcal{V}}$, we have that the statements and states resulting of evaluating one step by the semantical relation preserves the secrecy of unobservable information.

Theorem 4.12 If two CoreVHDL programs $\|_{i\in I} \ css_{i1}$ and $\|_{i\in I} \ css_{i2}$ are *secure* wrt. G and $\mathcal{V}$, when analysed in context B and we have that

$$\begin{array}{l} \|_{i\in I} \ \langle css'_{i1}, \sigma_{i1}, \varphi_{i1}\rangle \stackrel{i}{\Longrightarrow} \|_{i\in I} \ \langle css_{i1}{}'', \sigma'_{i1}, \varphi'_{i1}\rangle \ \wedge \\ \|_{i\in I} \ css_{i1} \ \mathcal{U}_{G,\mathcal{V}} \ \ \|_{i\in I} \ css_{i2} \ \wedge \\ \sigma_{i1} \sim_{\nabla(\mathcal{V},G,l)} \sigma_{i2} \ \wedge \\ \varphi_{i1} \sim_{\nabla(\mathcal{V},G,l)} \varphi_{i2} \end{array}$$

then there exists ss'_{i2}, σ'_{i2} and φ'_{i2} such that

$$\begin{array}{l} \|_{i\in I} \ \langle css'_{i2}, \sigma_{i2}, \varphi_{i2}\rangle \stackrel{i'}{\Longrightarrow} \|_{i\in I} \ \langle css_{i2}{}'', \sigma'_{i2}, \varphi'_{i2}\rangle \ \wedge \\ \|_{i\in I} \ css'_{i1} \ \mathcal{U}_{G,\mathcal{V}} \ \ \|_{i\in I} \ css'_{i2} \ \wedge \\ \sigma'_{i1} \sim_{\mathcal{V}} \sigma'_{i2} \ \wedge \\ \varphi'_{i1} \sim_{\mathcal{V}} \varphi'_{i2} \end{array}$$

Proof. The proof proceeds by induction in the shape of the derivation tree for the Structural Operational Semantics on the concurrent statement css_{i1}.

Case Handle non-waiting processes (H): This case follows from Lemma 4.10.

Case Active signals (A): The states are modified by clearing the active values, while moving the active values to the present values. Finally the condition on the synchronisation point (`until` e) must evaluate to the same because the analysis result for the program is secure.

Case Processes (P): This case holds vacuously. □

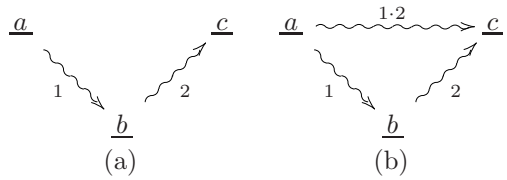


Figure 4.1: Result of the Information Flow Analysis for programs (a) and (b).

A result of Theorem 4.12 is that $\mathcal{U}_{G,\mathcal{V}}$ is a bisimulation of satisfying Definition 3.15 for programs that can be analysed as secure. Furthermore $\mathcal{U}_{G,\mathcal{V}}$ can be seen as satisfying Definition 3.10 for transitively closed security policies.

4.4 Discussion

In this chapter transitively closed security policies have been investigated. However, a security policy is given in the form of a directed graph. This graph will in general be non-transitive. To illustrate this point consider the following programs:

$$\textit{(a)}: \quad [\mathtt{c} := \mathtt{b}]^1; [\mathtt{b} := \mathtt{a}]^2 \qquad\qquad \textit{(b)}: \quad [\mathtt{b} := \mathtt{a}]^1; [\mathtt{c} := \mathtt{b}]^2$$

In program (a) there is a flow from b to c and a flow from a to b and therefore the security policy shown in Figure 4.1(a) has an edge from node b to node c and an edge from node a to node b. There is no flow from a to c and indeed there is no edge from a to c. In program (b) on the other hand there is a flow from a to c and the security policy shown in Figure 4.1(b) indeed has an edge from a to c. Analysing the two program (a) and (b) result in the same flow graph (i.e. Figure 4.1(b)), hence not allowing us to verify program (a) with respect to the security policy given in Figure 4.1(a). This is due to the flow insensitivity in the presented information flow analysis. Keeping this in mind we observe most investigations of information flow analyses are flow-insensitive e.g. [Den76, VSI96, Mye99a] and many more, see [SM03a] for an overview. In comparison the flow-sensitivity approaches are far more limited in numbers: [CHH02, AB04, HS06]. However all except for [AB04] rely on lattice-based (and hence transitively closed) security policies. This seems surprising as one can not express a transitively closed policy for program (a) without having to transform the program. Therefore we will continue the investigation of the information flow analysis in the following chapter, and discuss a flow sensitive extension and the benefit in expressing policies with it.

Chapter 5

Reaching Definitions Analysis

Reviling a locust tree when pointing at a mulberry tree.
— Chinese proverb

The previous chapter presented an flow insensitive analysis, that provided static guarantees sufficient for verifying systems when their security policies were transitively closed. This approach follows the tradition of information flow analysis [VSI96, ML97, Mye99a, Mye99b, SS00, SS01, SM03a]. Yet, investigations by Kemmerer [Kem82] have consider the information flows found by these approaches to be only the first step in the analysis of information flow. Indeed Kemmerer divides the analysis into two phases; first the local dependencies are identified, and second the global dependencies are calculated. In the previous chapter we proved that the analysis of local dependencies are sufficient to verify a program compliance with a transitively closed policy. Which match the results for the earlier presented analyses, due to their lattice based policies. However for our graph based policies this limitation seems to hinder the full expressiveness. In fact, previous investigations have shown the need for control flow sensitive mechanisms in information flow analyses [McH83]. Hence we aim to lift this constraint by extending the analysis by a flow sensitive component.

The approach taken will not be to modify the existing analysis, instead we choose to develop an analysis tracking the data flow in the program, and use it to remove the unwanted edges introduced in the security policy by restricting it to be transitively closed. The investigation will focus on the classical data flow analysis Reaching Definitions. In the following we will adapt a Reaching Definitions analysis to the setting of hardware descriptions. Some of the main issues

concerning the synchronisation and timing in hardware specifications make this to be a nontrivial task.

5.1 Analysing CoreVHDL

The main purpose of the Reaching Definitions analysis is to gather information about which assignments **may** have been made and not overwritten, when the execution reaches each point in the program.

The semantics divides signal states into two parts, namely the present value of a signal and the active value of a signal. Following this the analysis is divided into two parts as well, one for the active value of a signal and one for local variables and the present value of a signal. The two parts are connected since the active values of a signal influence the present value of the signal after the following synchronisation. Therefore we will first define the analysis of active signals in Section 5.1.2, and then, that of the local variables and present values of signals in Section 5.1.3.

The analysis for active signals is concerned only with a single process, and thus has no information about the other processes. It collects information about which signals **might** be active in order to gather all the influences on the present value; this information is gathered for the process i by an over-approximation analysis of the active signals $RD^{\cup\ i}_{\varphi}$. It also collects information about which signals **must** be active so that the overwritten signals can be removed from the analysis result; this information is gathered for the process i by an under-approximation analysis of the active signals $RD^{\cap\ i}_{\varphi}$.

The analysis of the local variables and present values of signals will be an over-approximation. It is concerned with the entire program and thus collects information for all processes at the same time.

5.1.1 Common analysis domains

The analyses use a labelling scheme, a block definition and a flow relation, similar to the ones described in [NNH99], the only difference being the wait statements which are given labels and treated as blocks. For each process i in a program

$$\|_{i \in I}\ i : \texttt{process}\ decl_i;\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i$$

the set of blocks is denoted $blocks(ss_i)$ and the flow relation is denoted $flow(ss_i)$. Similarly we use $init(ss_i)$ to denote the label of the initial block when executing the process i.

Data flow analyses are defined by pairs of functions that map labels to the appropriate sets; one function in each pair specifies information that is true on the entry to block, the second specifies information that is true at the exit. To aid us in defining these functions we will first define a number of simple operations on programs and labels. These are all simple adaptations of the operations specified in the classical data flow analyses [NNH99].

First we define a operation that returns the initial label of a statement

$$init : \mathbf{Stmt} \to \mathbf{Lab}$$

$$\begin{array}{rcl} init([x := e]^l) & = & l \\ init([s <= e]^l) & = & l \\ init([\texttt{null}]^l) & = & l \\ init(ss_1; ss_2) & = & init(ss_1) \\ init(\texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2) & = & l \\ init([\texttt{wait on } S \texttt{ until } e]^l) & = & l \end{array}$$

We also need to define a operation which returns the set of final labels in a statement; contrary to the fact that a sequence of statements has an isolated initial label, it may have multiple final labels, this is due to the conditionals.

$$final : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

$$\begin{array}{rcl} final([x := e]^l) & = & \{l\} \\ final([s <= e]^l) & = & \{l\} \\ final([\texttt{null}]^l) & = & \{l\} \\ final(ss_1; ss_2) & = & final(ss_2) \\ final(\texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2) & = & final(ss_1) \cup final(ss_2) \\ final([\texttt{wait on } S \texttt{ until } e]^l) & = & \{l\} \end{array}$$

The **Blocks** in a program or statement is the labelled elements, e.g. $[x := e]^l]$ or $[e]^l$. Therefore to access the Blocks, i.e. the statements, tests or synchronisation

points associated with a label, in a program we define the operation

$$blocks : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Blocks})$$

$$\begin{array}{rcl} blocks([x := e]^l) & = & \{[x := e]^l\} \\ blocks([s <= e]^l) & = & \{[s <= e]^l\} \\ blocks([\texttt{null}]^l) & = & \{[\texttt{null}]^l\} \\ blocks(ss_1; ss_2) & = & blocks(ss_1) \cup blocks(ss_2) \\ blocks(\texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2) & = & \{[e]^l\} \cup blocks(ss_1) \cup blocks(ss_2) \\ blocks([\texttt{wait on } S \texttt{ until } e]^l) & = & \{[\texttt{wait on } S \texttt{ until } e]^l\} \end{array}$$

Finally we define an operation on the flow between labels in a statement

$$flow : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

$$\begin{array}{rcl} flow([x := e]^l) & = & \emptyset \\ flow([s <= e]^l) & = & \emptyset \\ flow([\texttt{null}]^l) & = & \emptyset \\ flow(ss_1; ss_2) & = & flow(ss_1) \cup flow(ss_2) \\ & & \cup\{(l, init(ss_2)) \mid l \in final(ss_1)\} \\ flow(\texttt{if } [e]^l \texttt{ then } ss_1 \texttt{ else } ss_2) & = & flow(ss_1) \cup flow(ss_2) \\ & & \cup\{(l, init(ss_1)), (l, init(ss_2))\} \\ flow([\texttt{wait on } S \texttt{ until } e]^l) & = & \emptyset \end{array}$$

We define the cross flow relation cf for a program as the set of all possible synchronisations, i.e. cf is the Cartesian product of the set of labels of wait statements in each process.

The analyses are presented in a simplified way, following the tradition of the literature (see [NNH99]), where all programs considered are assumed to have so-called isolated entries (meaning that the entry nodes cannot be reentered once left). This is reasonable as each process in CoreVHDL can be considered as a skip statement followed by a loop with an always true condition around the statement defining the process. We shall write $FV(ss)$ for the set of free variables of the statement ss and similarly $FS(ss)$ for the set of free signals.

5.1.2 Analysis of active signals

The Reaching Definitions analysis takes the form of a Monotone Framework as given in [NNH99]. It is a *forward* Data Flow analysis, with both an over-

($RD^{\cup\ i}_{\varphi}$) and an under- ($RD^{\cap\ i}_{\varphi}$) approximation part; it operates over a complete lattice $\mathcal{P}(Sig_\star \times Lab_\star)$ where $Sig_\star$ is the set of signals and $Lab_\star$ is the set of labels present in the program.

In both cases we shall introduce functions recording the required information at the *entry* and at the *exit* of the program points. So for the over-approximation we have

$$RD^{\cup\ i}_{\varphi entry}, RD^{\cup\ i}_{\varphi exit} : Lab_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

and similarly for the under-approximation

$$RD^{\cap\ i}_{\varphi entry}, RD^{\cap\ i}_{\varphi exit} : Lab_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

To define the analysis we define in Table 5.1 a function

$$kill^i_{RD\varphi} : Blocks_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

which produces a set of pairs of signals and labels corresponding to the assignments that are killed by the block. A signal assignment can be killed for two reasons: Another block in the same process assigns a new value to the already active signal, or a wait statement in the same process will synchronise all active signals, and therefore kill all signal assignments.

In Table 5.1 we also define the function

$$gen^i_{RD\varphi} : Blocks_\star \to \mathcal{P}(Sig_\star \times Lab_\star)$$

that produces a set of pairs of signals and labels corresponding to the assignments generated by the block.

The over-approximation part of the analysis is defined in terms of the information that *may* be available at the entry of the statement. Therefore the over-approximation part considers a union of the information available at the exit of all statements that have a flow directly to the statement considered. In terms of the scenario of Figure 5.1(a) we have:

$$RD^{\cup\ i}_{\varphi entry}(l) = RD^{\cup\ i}_{\varphi exit}(l_1) \cup RD^{\cup\ i}_{\varphi exit}(l_2)$$

The under-approximation part of the analysis is defined in terms of the information that *must* be available at the entry of the statement. Therefore the under-approximation part considers an intersection of the information available

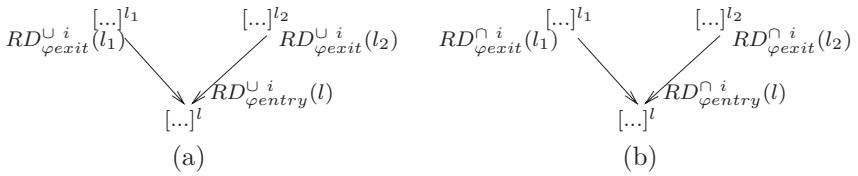


Figure 5.1: Relation between $RD^{\cup\ i}_{\varphi entry}(l)$ and $RD^{\cup\ i}_{\varphi exit}(l)$ (a), and $RD^{\cap\ i}_{\varphi entry}(l)$ and $RD^{\cap\ i}_{\varphi exit}(l)$ (b) for a process p_i

at the exit of all statements that have a flow directly to the statement considered. In terms of the scenario of Figure 5.1(b) we have:

$$RD^{\cap\ i}_{\varphi entry}(l) = RD^{\cap\ i}_{\varphi exit}(l_1) \cap RD^{\cap\ i}_{\varphi exit}(l_2)$$

The full details are presented in Table 5.1.

For the under-approximation analysis we define a special intersection operator; $\dot{\bigcap}\emptyset = \emptyset$, and $\dot{\bigcap}X = \bigcap X$ for $X \neq \emptyset$, to guarantee that $RD^{\cap\ i}_{\varphi entry} \subseteq RD^{\cup\ i}_{\varphi entry}$, will hold for the smallest solution to the equation systems.

5.1.3 Analysis of local variables and present values of signals

The Reaching Definitions analysis for the local variables corresponds to the Reaching Definitions analysis given in [NNH99]. For the present value of signals it will use the result of the Reaching Definitions analysis for active signals. The idea is that if a signal has an active value when execution of the program arrives at a synchronisation point, then the active value of the signal will become the present value of the signal after the synchronisation.

The result of the Reaching Definitions analysis for active signals can be computed *before* we perform the Reaching Definitions analysis for local variables and signals. Hence the result can be considered a static set, and therefore the Reaching Definitions analysis for local variables and signals remains an instance of a Monotone Framework.

kill and gen functions for the process
i : `process` $decl_i$; `begin` ss_i; `end process` i

$$
\begin{array}{rcl}
kill^i_{RD\varphi}([s <= e]^l) & = & \{(s,l')|B^{l'} \text{ assigns to } s \text{ in process } i\} \\
kill^i_{RD\varphi}([\texttt{wait on } S \texttt{ until } e]^l) & = & \{(s,l')|B^{l'} \text{ assigns to } s \text{ in process } i\} \\
kill^i_{RD\varphi}([\ldots]^l) & = & \emptyset \text{ otherwise} \\
 & & \\
gen^i_{RD\varphi}([s <= e]^l) & = & \{(s,l)\} \\
gen^i_{RD\varphi}([\ldots]^l) & = & \emptyset \text{ otherwise}
\end{array}
$$

data flow equations: RD_φ for the process
i : `process` $decl_i$; `begin` ss_i; `end process` i

$$
\begin{array}{rcl}
RD^{\cup\ i}_{\varphi entry}(l) & = & \begin{cases} \emptyset & \text{if } l = init(ss_i) \\ \bigcup\{RD^{\cup\ i}_{\varphi exit}(l')|(l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases} \\
 & & \\
RD^{\cup\ i}_{\varphi exit}(l) & = & (RD^{\cup\ i}_{\varphi entry}(l) \backslash kill^i_{RD\varphi}(B^l)) \cup gen^i_{RD\varphi}(B^l) \\
 & & \\
RD^{\cap\ i}_{\varphi entry}(l) & = & \begin{cases} \emptyset & \text{if } l = init(ss_i) \\ \dot{\bigcap}\{RD^{\cap\ i}_{\varphi exit}(l')|(l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases} \\
 & & \\
RD^{\cap\ i}_{\varphi exit}(l) & = & (RD^{\cap\ i}_{\varphi entry}(l) \backslash kill^i_{RD\varphi}(B^l)) \cup gen^i_{RD\varphi}(B^l)
\end{array}
$$

Table 5.1: Reaching Definitions Analysis for active signals; labels l are implicitly assumed to occur in process i : `process` $decl_i$; `begin` ss_i; `end process` i

The Reaching Definitions analysis for present values of signals operates over the complete lattice $\mathcal{P}(Sig_\star \times Lab_\star)$ and is a *forward* data flow analysis. It yields an over-approximation of the assignments that **might** have influenced the present value of the signal. Its goal is to define two functions holding the information at the *entry* and *exit* of a given label in the program:

$$RD^{cf}_{entry}, RD^{cf}_{exit} : Lab_\star \to \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)$$

The Reaching Definitions analysis for local variables and signals is given in Table 5.2 and makes use of two auxiliary functions. One is

$$kill^{cf}_{RD} : Blocks_\star \to \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)$$

kill and *gen* functions

$$
\begin{array}{rcl}
kill^{cf}_{RD}([x := e]^l) & = & \{(x,?)\} \cup \\
 & & \{(x,l') | B^{l'} \text{ assigns to } x \text{ in process } i\} \\
kill^{cf}_{RD}([\texttt{wait on } S \texttt{ until } e]^l) & = & \dot{\bigcap}_{(l_1,\ldots,l_n)\in cf, s.t.\ l_i=l} \\
 & & \quad \bigcup_{j=1}^{n} fst(RD^{\cap\ i}_{\varphi entry}(l_j)) \times wS(ss_i) \\
kill^{cf}_{RD}([\ldots]^l) & = & \emptyset \text{ otherwise} \\
 & & \\
gen^{cf}_{RD}([x := e]^l) & = & \{(x,l)\} \\
gen^{cf}_{RD}([\texttt{wait on } S \texttt{ until } e]^l) & = & \bigcup_{(l_1,\ldots,l_n)\in cf, s.t.\ l_i=l} \\
 & & \quad \bigcup_{j=1}^{n} fst(RD^{\cup\ i}_{\varphi entry}(l_j)) \times \{l\} \\
gen^{cf}_{RD}([\ldots]^l) & = & \emptyset \text{ otherwise}
\end{array}
$$

data flow equations: RD

$$
\begin{array}{rcl}
RD^{cf}_{entry}(l) & = & \\
 & & \begin{cases} \{(x,?) \mid x \in FV(ss_i)\} \cup \{(s,?) \mid s \in FS(ss_i)\} & \text{if } l = init(ss_i) \\ \bigcup \{RD^{cf}_{exit}(l') | (l',l) \in flow(ss_i)\} & \text{otherwise} \end{cases} \\
 & & \text{where } B \text{ and } i \text{ is uniquely given by } B^l \in blocks(ss_i) \\
 & & \\
RD^{cf}_{exit}(l) & = & RD^{cf}_{entry}(l) \backslash kill^{cf}_{RD}(B^l) \cup gen^{cf}_{RD}(B^l)
\end{array}
$$

Table 5.2: Reaching Definitions Analysis for the local variables and present value of signals, for all labels l in the program $\|_{i\in I}\ i$: `process` $decl_i$; `begin` ss_i; `end process` i.

that produces a set of pairs of variables or signals and labels corresponding to assignments that are overwritten by the block. An assignment to a local variable will overwrite all previous assignments on the execution path. A signal value can only be overwritten by a wait statement where at least one of the synchronising processes has an active value for the signal. To guarantee that an active value for a signal is available, the under-approximation analysis ($RD^{\cap\ i}_{\varphi}$) described above in Section 5.1.2 is used.

Since the active signal has to be present in all possible processes the considered wait statement could synchronise with, an intersection over the set cf of cross flow information is needed.

The other auxiliary function is

$$gen^{cf}_{RD} : Blocks_\star \to \mathcal{P}((Var_\star \cup Sig_\star) \times Lab_\star)$$

that produces a set of pairs of variables or signals, and labels corresponding to the assignments generated by the block. Only assignments to local variables generate definitions of a variable. Only wait statements are capable of changing a signal's value at present time. This means that in our analysis signals will get their present value at wait statements in the processes. The result of the over-approximation analysis ($RD^{\cup\ i}_{\varphi}$) contains all the signals that might be active and thus defines the present value after the synchronisation. Therefore we perform a union over all the signals that might be active in any process, that might be synchronised with.

Finally, all signals are considered to have an initial value in CoreVHDL hence a special label (?) is introduced to indicate that the initial value might be the one defining a signal at present time. The operator *fst* is defined by $fst(D) = \{s \mid (s, l) \in D\}$ and extracts the first components of pairs.

5.2 Global dependencies

Using the local dependencies defined above we can construct a Resource Matrix specifying for each point in the program which resources (i.e. a variable or signal) was modified and which resources were read meanwhile [McH95]. First we apply the local dependency analysis on each process in the considered program the result is collected in $RM_{lo} = \bigcup_i RM_i$ where $\emptyset \vdash ss_i : RM_i$. Then we need to compute the global dependencies; one way to do this is to take the transitive closure of the local dependencies; this method is attributed to Kemmerer and is described in [McH95] in case of traditional programming languages.

Let us evaluate the traditional method for constructing the Resource Matrix. For this we consider the program (a) presented in Section 4.4. The result of the transitive closure will correspond to the graph presented in Figure 4.1(b), but not to the true behaviour of the program as depicted in the graph in Figure 4.1(a). This is due to the flow-insensitivity of the transitive closure method: The imprecision is a result of the method failing to consider information about the flow between labels in the programs.

[RD for active signals]

$$\frac{(s, l_i, R_1) \in RM_{lo} \quad (s, l) \in RD^{\cup\, i}_{\varphi entry}(l_i)}{(s, l) \in RD^{\dagger}_{\varphi}(l_i)}$$

if $\exists\, \overrightarrow{l} \in cf : l_i$ occurs in $\overrightarrow{l}$

[RD for present signals and local variables]

$$\frac{(n, l', R_0) \in RM_{lo} \quad (n, l) \in RD^{cf}_{entry}(l')}{(n, l) \in RD^{\dagger}(l')}$$

Table 5.3: Specialisation of $RD^{\cup\, i}_{\varphi entry}$ and RD^{cf}_{entry}

[Initialisation]

$$\frac{(n, l, A) \in RM_{lo}}{(n, l, A, [\![l]\!]) \in RM_{gl}} \qquad \text{where } A \in \{R_0, R_1, M_0, M_1\}$$

[Present values and local variables]

$$\frac{(n', l, R_0, \delta'') \in RM_{gl} \quad (n', l') \in RD^{\dagger}(l) \quad (n, l', R_0, \delta') \in RM_{gl} \quad \delta' \cdot \delta'' \subseteq \delta}{(n, l, R_0, \delta) \in RM_{gl}}$$

[Synchronised values]

$$\frac{(s', l_i) \in RD^{\dagger}(l) \quad (s', l'') \in RD^{\dagger}_{\varphi}(l_j) \quad (s, l'', R_0, \delta') \in RM_{gl} \quad \delta' \subseteq \delta}{(s, l, R_0, \delta) \in RM_{gl}}$$

if $\exists\, \overrightarrow{l} \in cf : l_i$ and l_j occur in $\overrightarrow{l}$

Table 5.4: Transitive closure of Resource Matrix, based on $RD^{\dagger}$ and $RD^{\dagger}_{\varphi}$

5.2.1 Closure based on Reaching Definitions.

This motivates modifying the closure condition to make use of Reaching Definitions information. Indeed the Reaching Definitions analysis specified in this Chapter supplies us with the needed information to exclude some of the "spurious flows" when performing the transitive closure.

Before doing so we specialise in Table 5.3 the result of the Reaching Definitions analysis to allow a better precision in the closure of the Resource Matrix. The specialisation ensures that definitions are only considered to reach a labelled construct if they are actually used in the labelled construct. This is done by

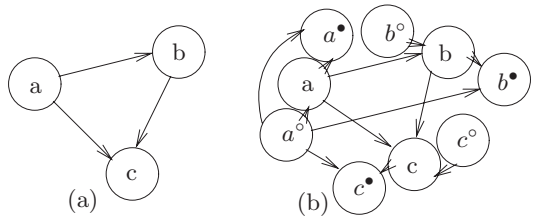


Figure 5.2: Result of the Information Flow Analysis for program (b).

considering the result of the local dependency analysis; notice the usage of the cross flow relation in rule [**RD for active signals**] which determines if the signal might in fact be synchronised.

We can now update the specification of the transitive closure using the result of the Reaching Definitions analysis, as is done in Table 5.4. We specify a rule for initialising the Global Resource Matrix [**Initialisation**]. Observe that we add an extra element in the resource matrix. This element is a regular language over execution traces (where we take the freedom of expressing them by their program labels rather than process identifiers). The regular language define the execution traces that might lead to a flow of information.

The closure is done by rule [**Present values and local variables**] considering the result of the Reaching Definitions analyses for the program. For the present value of a signal and for local variables we consider each entry in the Resource Matrix, if the present value of a variable or signal is read (R_0) we can use the information of the Reaching Definitions analysis to find the label where the variable or signal was defined. Therefore we copy all the entries about variables and signals read at this label in the Resource Matrix. This rule also handles the case where information flows from the variables and signals in a condition on a synchronisation point.

The rule [**Synchronised values**] uses the result of the Reaching Definitions analysis to determine which signals were read in the Resource Matrix and follow them to their definition. When synchronising signals the matter is complicated as the signal is defined at a synchronisation point, therefore the rule needs to consider all the information about signals flowing into the synchronisation points that might be synchronised with. Which synchronisation points the definition point synchronises with is gathered in the cf predicate, hence we apply the Reaching Definitions analysis for active signals on all the synchronisation points and copy all the entries indicating variables and signals being read from the

point the signal could be defined.

5.2.2 Improvement of the Information Flow Analysis

For the example program (b) (i.e. $[\texttt{b} := \texttt{a}]^1; [\texttt{c} := \texttt{b}]^2$) we previously described how the Information Flow Analysis would yield the result presented in Figure 5.2(a). In fact the resulting graph indicates that the resulting value of the variable `b` can be read from the resulting value of the variable `c`, which is entirely correct. However, the initial value of the variable `b` cannot be read from the variable c; to see this consider a scenario where `b` initially contained a value, this value would never flow to `c`, as the first assignment would overwrite the variable.

The Information Flow Analysis based on the Reaching Definitions Analysis can be improved to handle the initial and outgoing values of signals with greater accuracy. The idea is to add a node to the graph for each incoming signal, annotating the incoming node of a variable with a $\circ$, and for each outgoing signal, annotating the outgoing node with a $\bullet$. Using this scheme a more precise result for program (b) can be constructed as shown in Figure 5.2(b), where we consider the last statement to be outcoming and therefore update the Resource Matrix in the same fashion as for wait statements.

The extension of the analysis is based on adding special variables and signals for incoming and outgoing values. The rules for improving the information flow analysis are presented in Table 5.5 and explained below.

In a traditional sequential programming language the improvement could be handled by adding assignments of the form $x := x^\circ$ for each variable read in front of the program, and similarly adding assignments $x^\bullet := x$ in the end for handling the outgoing values. Having this in mind we introduce the rule [**Initial values**] that uses the special symbol (?) from the Reaching Definitions analysis to propagate the initial value of a variable or locally defined signal.

CoreVHDL consists of processes running as infinite loops in parallel with other processes and under the influence of the environment. Therefore signals might carry incoming values at any synchronisation point, similarly a process might communicate values out of the system at any synchronisation point. We introduce a new process π to illustrate how the incoming and outgoing signals are handled. The process has the form

π : `process begin` $[s_1^{in} <= s_1^\circ]; \ldots [\texttt{wait on } S^\pi]; [s_1^\bullet <= s_1^{out}]^{l_{s_1}^\bullet}; \ldots$
`end process` π

[Initial values]

$$\frac{(n, ?) \in RD^{\dagger}(l)}{(n^{\circ}, l, R_0, [\![true]\!]) \in RM_{gl}}$$

[Incoming values]

$$\frac{(n, l') \in RD^{\dagger}(l) \qquad l' \in WS}{(n^{\circ}, l, R_0, [\![true]\!]) \in RM_{gl}}$$

[Outgoing values]

$$\frac{n \in Sig^{out}}{(n^{\bullet}, l_{n^{\bullet}}, M_1, [\![true]\!]) \in RM_{gl}}$$

[Outcoming values]

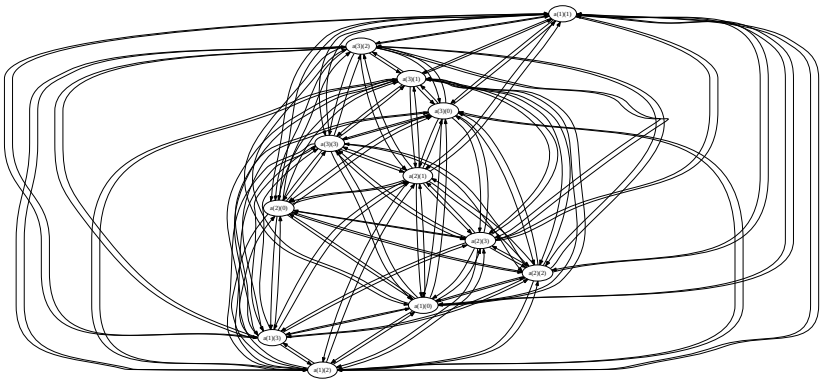
$$\frac{l \in WS \qquad (n, l') \in RD^{\dagger}_{\varphi}(l) \quad (n', l', R_0, \delta') \in RM_{gl} \quad \delta' \subseteq \delta}{(n', l_{n^{\bullet}}, R_0, \delta) \in RM_{gl}}$$

Table 5.5: Rules for the improved Information Flow Analysis

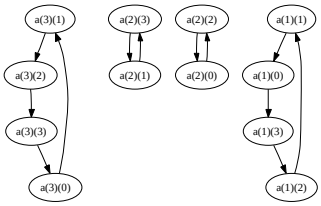
where $s_1^{in}, s_2^{in}, \ldots$ are the incoming signals, $s_1^{out}, s_2^{out}, \ldots$ are the outgoing signals and S^{π} is the set of all incoming and outgoing signals, as specified in the entity declaration of the program.

The assignments prior to the synchronisation point in process π can be synchronised into the system at each wait statement and this is handled by rule **[Incoming values]** where $WS = \bigcup_i WS(ss_i)$ are all the wait statements in the program.

We add two rules for the outgoing values to the closure method. The first rule **[Outgoing values]** specifies a special label for each signal (i.e. the label of the assignment to $n^{\bullet}$ in process π) used on the left-hand side in an assignment, at which the signal is set to be modified in the Resource Matrix. The second rule **[Outcoming values]** handles the right-hand side of the outgoing assignments in process π by considering all active signals coming into a wait statement, the values read when these active signals where modified are the signals that influence the outgoing value. Sig^{out} is the set of signals that are declared as outgoing (i.e. with the keyword `out` in the entity declaration).



(a)



(b)

Figure 5.3: Result of the Information Flow Analysis; by Kemmerer's method (a) and by the presented method (b).

5.3 Analysing Advanced Encryption Standard

For comparison between Kemmeres method and the one presented in this paper we consider part of the *NSA Advanced Encryption Standard* test implementation of the 128 bit version of the encryption algorithm Rijndael (i.e. `rijndael_pkg.vhdl`) [WBRF00]. In particular we consider the part of the encryption algorithm performing the shift of a state. It is implemented using a number of temporary local variables. These are overwritten several times and therefore Kemmeres method mixes all the values up as shown in figure 5.3(a). Our implementation of the presented method computes the precise result as shown in figure 5.3(b).

The *NSA Advanced Encryption Standard* test implementation use temporary

local variables in many parts of the program, therefore the correct handling of overwritten signals presented in our analysis is needed to analyse the implementation precise enough.

5.4 Discussion

Modern technical equipment often depends on the reliable performance of embedded systems. The verification of these systems have often relied on adapting existing techniques developed for software in to the setting of hardware specifications. The following three papers are some examples where static program analyses have been adapted for hardware, Hsieh and Levitan use data flow analyses for semantical extraction [HL98], Clarke et al. adapt a program slicing tool for verification of VHDL descriptions [CFR+99], and Hymans present an abstract interpretation for verification of safety properties [Hym02].

The paper by Hsieh and Levitan [HL98] considers a similar fragment of VHDL and is concerned with optimising the synthesis process by avoiding the generation of circuits needed to store values of signals. One component of the required analyses is a Reaching Definitions analysis with a similar scope to ours although specified in a rather different manner. Comparing the precision of their approach (to the extent actually explained in the paper) with the present one, it is clear that the present analysis is more precise in that it allows also to kill signals being set in other processes than where they are used. Furthermore the presented analysis is only correct for processes with one synchronisation point, because definition sets are only influenced by definitions in other processes at the end (or beginning) of a process. Therefore definitions are lost if they are present at a synchronisation point within the process but overwritten before the end of the process.

The paper by Clarke et al. [CFR+99] adapt the program slicing tool for verification of VHDL descriptions. The approach taken is mapping the language constructs to constructs in a traditional programming language. This approach allows the authors to consider an almost complete subset of the VHDL language, and provides professional user interfaces on the developed tool. However the mapping seems to disregard certain important aspects of the VHDL execution model, ignoring e.g. the subtleties regarding the timing behaviour of signals.

The paper by Hymans [Hym02] uses abstract interpretation to give an over-approximation of the set of reachable configurations for a fragment of VHDL not unlike ours. However there are some issues with the presented semantics as mentioned in Section 2.5. The presented analyses suffices for checking safety

properties: if the safety property is true on all states in the over-approximation it will be true for all executions of the VHDL program. Hence when synthesising the VHDL specification one does not need to generate circuits for enforcing the reference monitor (called an observer in [Hym02]).

The presented approach is based around adapting a Reaching Definitions analysis (along the lines of [NNH99]) to the setting of CoreVHDL. A novel feature of the analysis is that it has two components for tracking the flow of values of active signals: one is the traditional over-approximation whereas the other is an under-approximation. Finally, a Reaching Definitions analysis tracks the flow of variables and present values of signals. This combination of analyses enables us to deal with the special semantics found in VHDL and other hardware description languages due to the underlying timing model of the synthesised circuits.

The author is unaware of any other approach that using static analyses investigate Kemmerer's approach, including the method of identifying local and global dependencies. Previous investigations have considered using theorem provers for identifying information flow and covert channels, see e.g. [McH95] for an overview.

Furthermore extending the approach to include execution traces has not been presented before. The constraints on execution traces found in the security policy can be used in the closure condition. However, the constraints can be automatically inferred using the textbook approach for transforming a finite automaton in to a regular expression. The approach, commonly referred to as the *Pigeon Hole Principle* [HMU01], is applied to each set of nodes in the resource matrix, and the resulting regular expression is the constraint on the resulting edge between the two nodes. In a similar fashion the prototype implementation of the system uses a textbook algorithm for determining the inclusion of constraints in the resource matrix and the related constraints used in the security policy.

Chapter 6

Timing Leaks

Defer no time, delays have dangerous ends
— William Shakespeare

A security issue closely related to information flow and noninterference is *separability* [Lam73], originally dating back before noninterference. Separability is a very strong confidentiality property, that deal not only with the information flow through overt channels but also with information flow through *covert channels*. A covert channel is a channel that in the systems specification is meant included for a given purpose but can be exploited to leak information contrary to the security policy for the system. Traditionally two kinds of covert channels were identified; i.e. timing channels and storage channels. In this chapter we will investigate the first kind in the hardware setting. In cryptographic algorithms these channels are often referred to as *side channels*. Side channels focus only on how information flow occur outside the semantical specification of a system, rather than covert channels that focus on the information flows that occurs in the semantical model as well. Since the flows that occur in the semantical model have already been investigated in the previous chapter, this investigation will be focused on the side channels.

Practical implementations of embedded systems and synthesised circuits are vulnerable to side channel attacks. Side channels are often introduced at time of implementation, where information flow is added that was not originally described in the specification of the system. A key component is the security of the cryptographic algorithms often used in embedded systems. Much work

has been done to verify the security of such algorithms at specification level. However, Kocher [Koc96] showed that by observing physical aspects of implementations, like input-dependent differences in the execution time of a program, one could determine information that was considered secure in the specifications of cryptographic algorithms.

Several other side-channel attacks have been presented and shown to leak enough information to break commonly used cryptographic algorithms: power consumption [KJJ99], fault-based [KWMK01] and electro magnetic [QS01] channels. Timing channels have gotten most attention as they have been shown to be relevant in many practical settings, e.g. in smart cards [DKL+98].

This chapter presents a method for automatically verifying whether specifications in CoreVHDL contain timing leaks. We consider a subset of VHDL allowing synthesizable synchronous specifications. Meaning that the specification will have a global synchronisation signal, or a *clock* signal. The subset is similar to CoreVHDL except for the restriction on the synchronisation points. This restriction is motivated by observing that most VHDL specifications are of this kind, including our test specification of the AES algorithm. We present a structural operational semantics for the considered subset of VHDL in Section 6.1. The semantics differs from the previously presented semantics for CoreVHDL by a novel component for recording timing delays.

The problem of eliminating timing leaks has already been addressed in the literature. Several informal methods and guidelines for implementing algorithms without timing leaks have been presented, but here we focus on the ones presenting an automatic method for identifying the timing leaks.

Volpano and Smith [VS99] adapt a notion of protected branches of conditionals with atomic execution time. To do so, the programs considered are not allowed to include loops in conditionals depending on secret values. This approach guarantees the removal of timing channels leaking internally in the program; however Volpano and Smith note that for an external observer timing leaks might still occur.

Agat [Aga00] presents a method that transform programs so that both branches of a conditional depending on secret values will have the same execution time. This is based on a *filler-statement* that has the same timing properties as the corresponding statement, but that has no effect on the state of the system. The programs are transformed such that in each branch, the original branch and a filler-statement corresponding to the other branch is executed. Sabelfeld and Sands [SS00, Sab01] extend the method for a concurrent language with synchronisation points by running the statement and filler-statement in parallel

and synchronising them in the end.

Guaranteeing the absence of timing leaks in hardware introduces different issues. The differences between timing leaks in software and hardware systems are discussed in Section 6.2 where we propose a security condition for the absence of timing leaks in hardware specifications. In Section 6.3 we present a type based static analysis, that computes the different paths from input to output, allowing us to determine the absence of timing leaks in a VHDL specification.

In Section 6.5 we report on our experience from using the method on the *Advanced Encryption Standard* (AES).

6.1 CoreVHDL with Timing Behaviours

CoreVHDL$^{\circ}$ is a fragment of VHDL that concentrates on synthesizable specifications only. Hence all processes synchronise on the global clock.

The syntax is similar to that of CoreVHDL where a program is specified by an indexed family of processes or concurrent statements ($css \in Css$) running in parallel; here the index $i \in Id$ is a unique identifier in a finite set of process identifiers ($I \subseteq_{fin} Id$). Each process has a statement ($ss \in Stmt$) as body and may use logical values ($m \in LVal$), local variables ($x \in Var$) as well as signals ($s \in Sig$). The main difference from CoreVHDL is that in CoreVHDL$^{\circ}$ all synchronisation points are at the end of processes and synchronisation is with respect to the clock edge (clk).

Observe that VHDL processes, although seeming sequential in their specification, are concurrent in their execution. Because of the delay from a signal assignment to the new value taking effect (i.e. the delta-cycle), one can in fact freely exchange the order of execution of the signal assignments within a process, as long as they are not dependent on variables assigned in the process. For signals the value to be used is held in memory cells, which are updated along with the clock edge. Hence signal assignments are not dependent on the other signal assignments in the process, but rather the signal assignments executed in the previous cycle of the process.

The synthesising tools guarantee that processes will be ready for synchronisation with the clock edge. In reality the clock frequency is determined by measuring the worst case stabilising time of the electrical current through the gates synthesised from the VHDL specification. This worst case time can be measured by tools like SPICE [Tui92], IRSIM [SH89] and VSIM [VSI93]. Since the future

$css \in Css$		concurrent statements
css	::=	i : `process` (clk) `begin` ss; `end process` i
		$\mid \quad (css_1 \parallel css_2)$
$ss \in Stmt$		statements
ss	::=	`null` $\mid x := e \mid s <= e \mid ss_1; ss_2$
		$\mid$ `if` e `then` ss_1 `else` $ss_2 \mid [t]ss$
$e \in Exp$		expressions
e	::=	$m \mid x \mid s \mid$ `not` $e \mid e_1$ `and` e_2
		$\mid e_1$ `or` $e_2 \mid e_1$ `xor` e_2

Table 6.1: The subset CoreVHDL◔ of VHDL.

values are stored in memory cells not accessible before the synchronisation we can measure the delay from a value enters the system until a certain point by counting the synchronisation points on its execution path.

The syntax of CoreVHDL◔ is given in Table 6.1 where we omit the structural part of the syntax as it is the same as for CoreVHDL.

6.1.1 Semantics of CoreVHDL◔

We adapt the approach used in Chapter 2 when defining the structural operational semantics for CoreVHDL◔ where we add a notion of timing of signals to the semantics; this will prove crucial for developing our analysis.

The main idea is to execute each process by itself until a synchronisation point is reached. When all processes of the program have reached a synchronisation point we perform the system-wide synchronisation, while taking care of the resolution of signals in case a signal has been assigned different values by different processes.

Basic semantic domains. The syntax of specifications in CoreVHDL◔ operate on a state of logical values:

$$v \in LVal = \{\text{'U'}, \text{'X'}, \text{'0'}, \text{'1'}, \text{'Z'}, \text{'W'}, \text{'L'}, \text{'H'}, \text{'-'}\}$$

We assume the existence of a function mapping logicals in the syntax to logical

values in the semantics $\mathcal{L} : LVal \to Value$.

One goal of the semantics is to capture the timing delays of the evaluated systems, to do so we introduce a notion of time. Timing delays are specified relative to incoming signals (in the set Sig_{IN}) and the time measured as delays (in integers $\mathbb{Z}$) they have flowed through the system

$$t \in Time = \mathcal{P}(Sig_{IN} \times \mathbb{Z})$$

If a timing delay t includes the pair (s, z) then t indicates that there is a dependency on the incoming signal s with a delay of $|z|$ clock cycles (where $z \leq 1$).

Constructed semantic domains. CoreVHDL$^{\textcircled{\scriptsize C}}$ includes local variables and signals. The values and the timing delays of the local variables are stored in a local state. The local state is a mapping from variable names to logical values and timing delays.

$$\sigma \in State = (Var \to Value \times Time)$$

Signals are used for communication between processes and their values and timing delays are stored in local states.

$$\varphi \in Store = (Sig \to (\{0, 1\} \hookrightarrow Value \times Time))$$

The value assigned to a signal is available after the following synchronisation, therefore we keep the present value and timing delay of a signal s in $\varphi\ s\ 0$. In $\varphi\ s\ 1$ we store the assigned value and its timing delay, meaning that it is available after a *delta-cycle*.

All signals have a present value, so $\varphi\ s\ 0$ is defined for all s. Not all signals need to be active meaning they have a new value waiting in the following delta-cycle, thus $\varphi\ s\ 1$ need not be defined; hence we use $\{0, 1\} \hookrightarrow Value \times Time$ in the definition of the signal state to indicate that it is a partial function.

The semantics handles expressions following the ideas of [NN92]. For expressions

$$\mathcal{E} : Expr \to (State \times Signals \to Value \times Time)$$

evaluates the expression. The function is defined in Table 6.2. Note that for signals we use the current value and timing delay of the signal, i.e. $\varphi\ s\ 0$.

$\mathcal{E}[\![m]\!]\langle\sigma,\varphi\rangle$	$=$	$(\mathcal{L}[\![m]\!],\emptyset)$	
$\mathcal{E}[\![x]\!]\langle\sigma,\varphi\rangle$	$=$	$\sigma\ x$	
$\mathcal{E}[\![s]\!]\langle\sigma,\varphi\rangle$	$=$	$\varphi\ s\ 0$	
$\mathcal{E}[\![\texttt{not}\ e]\!]\langle\sigma,\varphi\rangle$	$=$	$(\overline{\texttt{not}}\ v,t)$	where $\mathcal{E}[\![e]\!]\langle\sigma,\varphi\rangle=(v,t)$
$\mathcal{E}[\![e_1\ \texttt{and}\ e_2]\!]\langle\sigma,\varphi\rangle$	$=$	$(v_1\ \overline{\texttt{and}}\ v_2,t_1\cup t_2)$	where $\mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle=(v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle=(v_2,t_2)$
$\mathcal{E}[\![e_1\ \texttt{or}\ e_2]\!]\langle\sigma,\varphi\rangle$	$=$	$(v_1\ \overline{\texttt{or}}\ v_2,t_1\cup t_2)$	where $\mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle=(v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle=(v_2,t_2)$
$\mathcal{E}[\![e_1\ \texttt{xor}\ e_2]\!]\langle\sigma,\varphi\rangle$	$=$	$(v_1\ \overline{\texttt{xor}}\ v_2,t_1\cup t_2)$	where $\mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle=(v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle=(v_2,t_2)$

Table 6.2: Semantics of Expressions

6.1.2 Statements

The semantics of statements and concurrent statements is given as a structural operational semantics. For statements we shall use configurations of the form

$$t \vdash \langle ss', \sigma, \varphi\rangle \in Time \times Stmt' \times State \times Signals$$

Here $Stmt'$ refers to the statements from the syntactical category $Stmt$ with an additional statement (`final`) indicating that a final configuration has been reached. Therefore the transition relation for statements has the form

$$t \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle$$

which specifies one step of computation. Here t is the context that might influence the timing delay of evaluated statements. The transition relation is specified in Table 6.3 and briefly commented upon below.

An assignment to a signal is defined as an update to the value and timing delay at the delta-time, i.e. $\varphi\ s\ 1$. We use the notation $\varphi^{[i]}[s \mapsto (v,t)]$ to mean $\varphi[s \mapsto \varphi(s)[i \mapsto (v,t)]]$. Conditionals influence the timing delay of signals assigned in the branches, therefore we use $[t]$ in front of a statement to indicate that the execution of the statement depends on the timing delay t.

6.1.3 Concurrent Statements

The semantics for concurrent statements handles the concurrent processes and their synchronisations of a CoreVHDL$^{\circ}$ program. The transition system for

concurrent statements has configurations of the form

$$\|_{i\in I}\ \langle css_i', \sigma_i, \varphi_i\rangle$$

for $I \subseteq_{fin} Id$ and $css_i' \in \{ss'; css \mid ss' \in Stmt' \wedge css \in Css\} \cup Css$, $\sigma_i \in State$, $\varphi_i \in Signals$ for all $i \in Id$. Thus each process has a local variable and signal state. The transition relation for concurrent statements has the form

$$\|_{i\in I}\ \langle css_i', \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle css_i'', \sigma_i', \varphi_i'\rangle$$

which specifies one step of computation. The transition relation is specified in Table 6.4 and explained below.

As mentioned before we execute processes locally until they have all arrived at a synchronisation point; this is reflected in the rule [**Handle non-waiting processes (H)**]. The body of a process is executed in an empty timing context, as the execution does not depend on the timing delay of any signal.

When all processes finish execution we perform a synchronisation covered by the rule [**Active signals (A)**]. Applications of this rule indicate the coming clock edge. This rule is explained in the sequel.

The delta-time values and timing delays of signals will be synchronised for all processes and in order to do this we use a resolution function f_s of the form:

$$f_s : multiset(Value \times Time) \to Value \times Time$$

Thus f_s combines the *multi-set* of values assigned to a signal and their timing delays into one value and timing delay that then will be the new (unique) value and timing delay of the signal. VHDL allow a resolution function to be specified in a syntax much like that of a process. We assume that all signals (and their timing delays) handled in resolution functions are in the argument of the function

$$f_s(\mathcal{R}_s) = (v, t) \quad \wedge \quad t = \bigcup_{(v',t')\in\mathcal{R}_s} t'$$

where $\mathcal{R}_s$ is the multi-set of values and timing delays used to perform the resolution of the new present value for s. The resolution function is applied to the active values of a signal, however if a signal is not active in a local signal state of a process the present value is used instead. We introduce the function

$$\mathcal{U}(\varphi_i, s) = \begin{cases} (v,t) & \text{if } \varphi_i\ s\ 1 = (v,t) \\ (v,t) & \text{otherwise, } \varphi_i\ s\ 0 = (v,t) \end{cases}$$

[Local Variable Assignment] :

$$t \vdash \langle x := e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma[x \mapsto (v, t \cup t')], \varphi\rangle$$
where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = (v, t')$

[Signal Assignment] :

$$t \vdash \langle s <= e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi^{[1]}[s \mapsto (v, t \cup t')]\rangle$$
where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = (v, t')$

[Skip] :

$$t \vdash \langle \texttt{null}, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi\rangle$$

[Composition] :

$$\frac{t \vdash \langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma', \varphi'\rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi'\rangle}$$
where $ss_1' \in Stmt$

$$\frac{t \vdash \langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_2, \sigma', \varphi'\rangle}$$

[Conditional] :

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t']ss_1, \sigma, \varphi\rangle$$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = (\text{'1'}, t')$

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t']ss_2, \sigma, \varphi\rangle$$
if $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = (\text{'0'}, t')$

[Branch] :

$$\frac{t \cup t' \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle}{t \vdash \langle [t']ss, \sigma, \varphi\rangle \Rightarrow \langle [t']ss', \sigma', \varphi'\rangle}$$

$$\frac{t \cup t' \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{t \vdash \langle [t']ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}$$

Table 6.3: Semantics of Statements

to determine whether the active value or the present value is used.

Incoming signals interact with the processes at synchronisation points. The incoming values can be modelled as a signal state of the environment which has the timing delay $\{(s_{in}, 1)\}$ for all active values of incoming signals s_{in} because after synchronisation we then obtain the desired timing delay $\{(s_{in}, 0)\}$ for the incoming signals.

[Handle non-waiting processes (H)] :

$$\frac{\emptyset \vdash \langle ss_j, \sigma_j, \varphi_j\rangle \Rightarrow \langle ss'_j, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle ss'_i; css_i, \sigma'_i, \varphi'_i\rangle}$$

where $ss'_i = ss_i \wedge \sigma'_i = \sigma_i \wedge \varphi'_i = \varphi_i$ for all $i \neq j$.

$$\frac{\emptyset \vdash \langle ss_j, \sigma_j, \varphi_j\rangle \Rightarrow \langle \texttt{final}, \sigma'_j, \varphi'_j\rangle}{\|_{i\in I\cup\{j\}}\ \langle ss_i; css_i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I\cup\{j\}}\ \langle css'_i, \sigma'_i, \varphi'_i\rangle}$$

where $css'_i = css_i$ for all $i = j$ and $css'_i = ss_i; css_i \wedge \sigma'_i = \sigma_i \wedge \varphi'_i = \varphi_i$ for all $i \neq j$.

[Active signals (A)] :

$$\|_{i\in I}\ \langle i : \texttt{process}\ (clk)\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle css'_i, \sigma_i, \varphi'_i\rangle$$

where

$$\varphi'_i\ s\ 0 = \begin{cases} f_s\{\!\{(v, t^{--}) \mid \mathcal{U}(\varphi_k, s) = (v, t) \wedge k \in I\}\!\} & \text{if } \exists j \in I.\ \varphi_j\ s\ 1 \text{ is defined} \\ (v, t^{--}) & \text{otherwise, } \varphi_i\ s\ 0 = (v, t) \end{cases}$$

$$\varphi'_i\ s\ 1 = \mathit{undef}$$

$$css'_i = ss_i; i : \texttt{process}\ (clk)\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i$$

Table 6.4: Semantics of Concurrent statements

Synchronisation of signals will modify the time-line in the signal states, therefore we need to change the timing delays accordingly. Thus if a timing delay includes (s, z) at the time of synchronisation then the signal s is also synchronised moving the referred value to $(s, z-1)$. Therefore we introduce the function $^{--}$

$$t^{--} = \{(s, z-1) \mid (s, z) \in t\}$$

Notice that the state of local variables is unchanged.

6.2 Absence of Timing Leaks

To argue whether a program is secure or not, we need some condition for security. Many of the related methods for handling timing leaks base their security condition on a kind of non-interference. This deals with all kinds of information flow including side channels but its view is usually limited to considering

only the input and output of programs. Timing channels are often exhibited by cryptographic algorithms, for which non-interference between the secret plain text and the public cipher text is hard to establish. Here we focus on timing channels, and therefore we concentrate on formulating our security condition for this problem. Before we can define the security condition let us consider some requirements for programs.

Timing channels are defined as differences in the timing delay introduced by handling of secret information. In software programming languages an example of such a channel could be

$$\texttt{if } h = 1 \texttt{ then sleep } 100 \qquad (*)$$

where h is a secret variable. The timing channel allows an observer to gain information about the value of h, by measuring the execution time of the program.

In hardware the setting is different. Any statement is encapsulated in a process synchronised with time. Therefore if a statement as $(*)$ is placed within a process, the synthesis of the specification must guarantee that the process is ready for synchronisation at the time of the clock edge. Since the memory cells hold the previous value for each signal, the value will always be present at exactly the same time. Furthermore processes loop infinitely, so measuring the overall execution time of a VHDL specification makes no sense.

In our setting we consider an external observer that can measure the time (measured in clock cycles) between modifications of ingoing and outgoing signals. By doing this the observer might gain secret information. As an example, consider a network of sensors where the sensors communicate with each other by encrypted messages. The timing delay of the encryption scheme depends on the value of the plain text as well as the secret key. Messages will be routed through other sensors to allow communication between all sensors. Now assume that one sensor is hostile, and wishes to gain information about the messages passed between other sensors. The hostile sensor takes the following actions: it sends a message to another sensor, that it shares a key with, then it measures the time spent by the receiving sensor on decrypting the message and encrypting it again and passing it on to yet another sensor. Since the hostile sensor knows the key with which the messages were originally encrypted it could also calculate the time spent on the decryption, and hence get some idea of the time spent on the encryption. By continuously sending messages that vary only on single bits the hostile sensor can use the timing channel to learn the key shared by the other two sensors.

```
buffer: process (clk) begin
  b <= a;
end process;

selector: process (clk) begin
  if sel then
    o <= a;
  else
    o <= b;
end process;
```

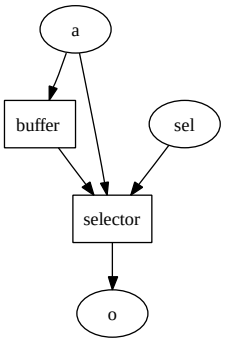


Figure 6.1: The example program selector with process dependencies

Example 6.1 *(Selector) Consider a program that selects between the newest and the previous value of the incoming signal* `a`*, based on the incoming signal* `sel`*. The specification consists of two processes, i.e.* `buffer` *and* `selector`*; the first is a buffer setting the previous incoming signal on the internal signal* `b`*. The second process branches on the incoming signal* `sel` *and assigns either the incoming signal* `a` *or the internal signal* `b` *to the outgoing signal* `o`*. The* CoreVHDL *specification is given in Figure 6.1.*

Now the timing delay of `o` *depends on* `sel`*. If* `sel` *is true then the value of* `a` *flows to* `o` *after one clock cycle, and we write* $o \mapsto \{(a, -1)\}$*, however if* `sel` *is false then* `a` *is delayed two clock cycles before flowing to* `o`*, and we write* $o \mapsto \{(a, -2)\}$*. Since only one branch is executed the timing delay of* `o` *can be used to determine the value of* `sel`*. Therefore the specification is considered to have a timing leak.*

The timing delays can also be found by considering dependency diagrams, like the one in Figure 6.1 for the selector program. Here the ovals containing signal names are the incoming and outgoing signals (i.e. `a`*,* `sel` *and* `o`*). However* `b` *is not drawn in a oval, as it is an internal signal and therefore does not appear in the timing delays. Instead the arrow between the boxes* `buffer` *and* `selector` *represent* `b`*. The processes in the specification are illustrated as boxes (i.e.* `buffer` *and* `selector`*). Finding the timing delay of the outgoing signal* `o` *can be done by finding the paths through the diagram, increasing the delay (i.e. decreasing the timing delay) whenever passing a process box. The specification has a timing leak because there are paths of different length from the incoming signal* `a` *to the outgoing signal* `o`*. One is through both* `buffer` *and* `selector`*, and the other is only through* `selector`*.*

Unfortunately finding paths of different length in a dependency diagram is not enough. Modify the `selector` *to perform the computation* `o <= a xor b xor sel`*. The dependency graph is the same, but no timing leak is present. The difference between the two specifications is that in the unmodified specification the signal* `o` *can depend on only one of the paths, while in the modified specification* `o` *depends on both paths.*

We say a program is *timing secure* if on no input should an observer be able to differentiate between two executions of the program, by considering the number of clock cycles elapsed until the outgoing signals change their values. We formalise this condition by using the timing delay computed in the semantics for CoreVHDL◔. Therefore the program is *timing secure* if two executions of a program in different states always have the same timing delay for all signals.

In the definition of a timing secure program we write $\Longrightarrow_H^*$ as the transitive reflexive closure of applying rule [**Handle non-waiting processes (H)**] and we write $\Longrightarrow_A$ for applying rule [**Active signals (A)**]. The projection on the time component $|_{Time}$ is defined as $(v,t)|_{Time} = t$. Later we will similarly write $|_{Value}$ for the projection on the value component $(v,t)|_{Value} = v$.

Definition 6.1 (Timing secure program) A CoreVHDL◔ program $\|_{i\in I}\ css_i$ is *timing secure* if

$$\begin{array}{l}\|_{i\in I}\ \langle css_i, \sigma_{1i}, \varphi_{1i}\rangle \Longrightarrow_H^* \Longrightarrow_A \|_{i\in I}\ \langle css_i', \sigma_{1i}', \varphi_{1i}'\rangle \wedge \\ \|_{i\in I}\ \langle css_i, \sigma_{2i}, \varphi_{2i}\rangle \Longrightarrow_H^* \Longrightarrow_A \|_{i\in I}\ \langle css_i', \sigma_{2i}', \varphi_{2i}'\rangle \wedge \\ \forall i, x : \sigma_{1i}\ x|_{Time} = \sigma_{2i}\ x|_{Time} \wedge \\ \forall i, j, s : \varphi_{1i}\ s\ j|_{Time} = \varphi_{2i}\ s\ j|_{Time}\end{array}$$

where for each index i the two branches produce the same statement css_i' implies

$$\forall i, j, s : \varphi_{1i}'\ s\ j|_{Time} = \varphi_{2i}'\ s\ j|_{Time}$$

Observation. The definition of *timing secure* programs follows from the observation that processes execute their body and then wait for synchronisation. Therefore for a program to be *timing secure* all execution traces of a process' body must change the timing delay of the affected signals uniformly. Notice that the timing delay of a variable or signal is changed even when the value is unmodified, see [**Local Variable Assignment**] and [**Signal Assignment**].

In Example 6.1 we presented a program and showed how an observer could gain knowledge from observing the timing delays of assignments to a signal. Now

we introduce a notion of security thresholds, that allow us to argue whether an observer can gain information outside some threshold $\mathcal{L} : Sig \rightarrow \mathcal{P}(Sig)$ from timing delays, where $\mathcal{L}(s) \subseteq Sig_{IN}$ for all $s \in Sig$. This corresponds to the observer knowing the values of the signals in the threshold. Hence the above given program will be timing secure for observers that hold the signal `sel` in their threshold.

This leads to considering a program secure for an observer of signal $s^\star$ with threshold $\mathcal{L}(s^\star)$ if the observer cannot observe different values of the signal when no signal within the threshold differ on their incoming value by observing the timing delays.

Definition 6.2 (($\mathcal{L}, s^\star$)-Timing secure program) A CoreVHDL program $\|_{i\in I}$ css_i is $(\mathcal{L}, s^\star)$-*timing secure* if

$$\begin{array}{l} \|_{i\in I} \; \langle css_i, \sigma_{1i}, \varphi_{1i}\rangle \Longrightarrow_H^* \Longrightarrow_A \|_{i\in I} \; \langle css_i', \sigma_{1i}', \varphi_{1i}'\rangle \; \wedge \\ \|_{i\in I} \; \langle css_i, \sigma_{2i}, \varphi_{2i}\rangle \Longrightarrow_H^* \Longrightarrow_A \|_{i\in I} \; \langle css_i', \sigma_{2i}', \varphi_{2i}'\rangle \; \wedge \\ \forall i, x : \sigma_{1i} \; x|_{Time} = \sigma_{2i} \; x|_{Time} \; \wedge \\ \forall i, j, s : \varphi_{1i} \; s \; j|_{Time} = \varphi_{2i} \; s \; j|_{Time} \; \wedge \\ \forall i', j', s' : s' \in \mathcal{L}(s^\star) \Rightarrow \varphi_{1i'} \; s' \; j'|_{Value} = \varphi_{2i'} \; s' \; j'|_{Value} \end{array}$$

where for each index i the two branches produce the same statement css_i' implies

$$\forall i, j : \varphi_{1i}' \; s^\star \; j|_{Time} = \varphi_{2i}' \; s^\star \; j|_{Time}$$

A specification is timing secure if it is secure for observation on all signals. Hence the security property of a timing secure specification with respect to the threshold $\mathcal{L}$ is defined as follows.

Definition 6.3 ($\mathcal{L}$-Timing secure program) A CoreVHDL program $\|_{i\in I}$ css_i is $\mathcal{L}$-*timing secure* if it is *$(\mathcal{L}, s^\star)$-timing secure* for all $s^\star \in Sig$.

The definition presented resembles that of [SS00] as we apply a kind of mutation or reseting of the states after each synchronisation point for longer execution sequence (i.e. between each clock cycle). This allows us to focus on the observable values without dealing with the values that flow directly from higher to lower security classes. It follows that we permit controlled declassification without making more information observable through timing channels.

6.3 Execution Path Analysis

This section presents an automatic type-based analysis for certifying secure programs. The analysis is an over-approximation, where a type describing the timing delay of variables and signals manipulated can only be inferred for programs that can be guaranteed to comply with the security condition presented in Section 6.2. We prove the correctness of the analysis in Section 6.4.

The aim of the analysis is to determine the different paths along which an input signal can flow through a system until it reaches an outgoing signal. With this knowledge one can decide if all the paths have the same length with respect to timing delays. For each point in the program the analysis checks that all information flows introduced have the same timing delay. As said, the timing of different execution paths in a CoreVHDL program is determined by the number of synchronisation points passed. The analysis developed here is limited to handling programs where all paths are of finite length, i.e. no process depends on its own output. This limitation follow the tradition of [VS99, Aga00, SS00, Sab01].

The analysis of an expressions e has the form

$$\Gamma \vdash e : \delta$$

where Γ is an environment that holds the timing effect of variables and signals ($\Gamma : (Sig \cup Var) \hookrightarrow \mathcal{P}(Time)$). The timing effect δ ($\delta \in \mathcal{P}(Time)$) is the set of timing delays of the evaluated expression. The analysis of expressions is presented in Table 6.5.

For the environment Γ it holds that

$$\forall s \in Sig_{IN} : \Gamma(s) = \{\{(s, 0)\}\}$$

indicating that for all in-coming signals, the delay from their entry to the system and to the point where used is always zero clock cycles. This matches the notion of time presented in the semantics, where the incoming signals are introduced as active values in the environment, and then become present values when synchronised. Notice that a program is not allowed to write to an in-coming signal in VHDL [IEE87].

For combining paths we introduce the operator $\otimes$ as

$$\delta \otimes \delta' = \{t \cup t' \mid t \in \delta \wedge t' \in \delta'\}$$

$$\Gamma \vdash m : \emptyset \qquad \Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash s : \Gamma(s)$$

$$\frac{\Gamma \vdash e : \delta}{\Gamma \vdash \texttt{not}\ e : \delta} \qquad \frac{\Gamma \vdash e_1 : \delta_1 \qquad \Gamma \vdash e_2 : \delta_2}{\Gamma \vdash e_1\ \texttt{and}\ e_2 : \delta_1 \otimes \delta_2}$$

$$\frac{\Gamma \vdash e_1 : \delta_1 \qquad \Gamma \vdash e_2 : \delta_2}{\Gamma \vdash e_1\ \texttt{or}\ e_2 : \delta_1 \otimes \delta_2} \qquad \frac{\Gamma \vdash e_1 : \delta_1 \qquad \Gamma \vdash e_2 : \delta_2}{\Gamma \vdash e_1\ \texttt{xor}\ e_2 : \delta_1 \otimes \delta_2}$$

Table 6.5: Typing rules for expressions

The analysis of a statement ss has the form

$$\Gamma, \delta \vdash ss : \tau$$

where Γ is the environment that holds the timing effect of variables and signals, identical to the one used for analysing expressions. The timing effect δ ($\delta \in \mathcal{P}(Time)$) might affect the manipulated signals' timing delays due to control flow in the program. The timing behaviour τ is a map of variables and signals manipulated by ss, and their timing effect ($\tau : (Var \cup Sig) \hookrightarrow \mathcal{P}(Time)$)).

When analysing assignments to variables and signals we need to take into account the timing delays of variables and signals that determine the control flow of the process. Hence we collect these timing delays in the δ component. The types of variables and signals therefore need to include this component, and we check that the Cartesian product of the paths are equal to the type (i.e. $\delta \otimes \delta' = \Gamma(x)$ for the variable x). Furthermore we need to handle the delay from assignment of a signal to the value becomes the present value of the signal. The modification of a signal cannot be observed before the following clock edge, so in the rule for signal assignment the type is modified accordingly. This is done by overloading the function $^{--}$ to apply on all timing delays in a timing effect.

For the composition of statements and concurrent statements we introduce the operator $\curlyvee$ on two timing behaviours by

$$(\tau_1 \curlyvee \tau_2)(n) = \begin{cases} \tau_2(n) & \text{if } n \in dom(\tau_2) \\ \tau_1(n) & \text{otherwise} \end{cases}$$

[**Local Variable Assignment**] :

$$\frac{\Gamma \vdash e : \delta' \qquad \delta \otimes \delta' = \Gamma(x)}{\Gamma, \delta \vdash x := e : [x \mapsto \delta \otimes \delta']}$$

[**Signal Assignment**] :

$$\frac{\Gamma \vdash e : \delta' \qquad (\delta \otimes \delta')^{--} = \Gamma(s)}{\Gamma, \delta \vdash s <= e : [s \mapsto (\delta \otimes \delta')^{--}]}$$

[**Skip**] :

$$\Gamma, \delta \vdash \texttt{null} : \bot$$

[**Final**] :

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

[**Composition**] :

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

[**Low Conditional**] :

$$\frac{FV(e) \cup FS(e) \subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau_1 \Cup \tau_2}$$

[**High Conditional**] :

$$\frac{FV(e) \cup FS(e) \not\subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau}$$

where $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$

[**Branch**] :

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss : \tau}{\Gamma, \delta \vdash [t']ss : \tau}$$

where $dom(\tau_1) \cap dom(\tau_2) = \emptyset$

[**Process**] :

$$\frac{\Gamma, \emptyset \vdash ss_1 : \tau}{\Gamma \vdash_c i : \texttt{process } S \texttt{ begin } ss_1; \texttt{ end process } i : \tau}$$

[**Concurrency**] :

$$\frac{\Gamma \vdash_c css_1 : \tau_1 \qquad \Gamma \vdash_c css_2 : \tau_2}{\Gamma \vdash_c css_1 \parallel css_2 : \tau_1 \curlyvee \tau_2}$$

where $dom(\tau_1) \cap dom(\tau_2) = \emptyset$

Table 6.6: Typing rules for statements and concurrent statements

Conditionals might introduce differences in the timing effect of signals, therefore if the condition contains signals outside the security threshold $\mathcal{L}$, the analysis verifies that both branches modify signals based on the same timing behaviours. We add to constraint $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$ to make sure nested conditionals do not mask timing channels through statements that is unreachable. If the condition does not contain any signals above the observers threshold ($FV(e) \cup FS(e) \subseteq \mathcal{L}$, where FV and FS are the free variables and signals in the expression e, respectively) we know that information does not become observable. Thus we can allow both paths even when they are different. We unite the paths by the operator $\Cup$ defined as

$$(\tau_1 \Cup \tau_2)\ n = (\tau_1\ n) \cup (\tau_2\ n)$$

The analysis of concurrent statements has the form

$$\Gamma \vdash_c css : \tau$$

using the same elements as the analysis of statements, except for the timing effects introduced by conditionals. The analysis verifies the body of each process. The analysis of statements and concurrent statements is presented in Table 6.6.

6.4 Soundness

We prove the soundness of the analysis with respect to the semantics presented in Section 6.1. The analysis checks that the points in time where a signal (or variable) is modified matches the timing effect, so we prove that for any program evaluated in the semantics, and if a type effect can be verified with the analysis then the variable and signal states hold the same timing behaviour as the type for all variables and signals.

First, we define an ordering $\sqsupseteq$ over the types as

$$\tau \sqsupseteq \tau' \equiv \forall : n \in dom(\tau') \Rightarrow \tau\ n \supseteq \tau'\ n$$

The first result we aim to show is that expressions evaluated in states, that have well-formed timing behaviours for all variables and signals, also have resulting timing behaviours that match the result of the analysis.

Lemma 6.4 (Secure Expressions) For every expression e where the semantics evaluates the expression in the states σ_1 and φ_1 as $\mathcal{E}[\![e]\!]\langle\sigma_1, \varphi_1\rangle = (v_1, t_1)$, and

in the states σ_2 and φ_2 as $\mathcal{E}[\![e]\!]\langle\sigma_2, \varphi_2\rangle = (v_2, t_2)$ we have

$$\left.\begin{array}{ll} \Gamma \vdash e : \delta \ \wedge & \\ \sigma_1\ x|_{Time} \in \Gamma\ x\ \wedge\ \sigma_2\ x|_{Time} \in \Gamma\ x \ \wedge & (6.4.1) \\ \varphi_1\ s\ 0|_{Time} \in \Gamma\ s\ \wedge\ \varphi_2\ s\ 0|_{Time} \in \Gamma\ s & (6.4.2) \end{array}\right\} \Rightarrow t_1 \in \delta\ \wedge\ t_2 \in \delta$$

Proof. The proof proceeds by induction on the shape of the derivation sequence.

Case m : The semantics evaluates the expression as

$$\mathcal{E}[\![m]\!]\langle\sigma_1, \varphi_1\rangle = (\mathcal{L}[\![m]\!], \emptyset)$$

and

$$\mathcal{E}[\![m]\!]\langle\sigma_2, \varphi_2\rangle = (\mathcal{L}[\![m]\!], \emptyset)$$

The analysis gives

$$\Gamma \vdash m : \emptyset$$

and the case follows.

Case x : The semantics evaluates the expression as

$$\mathcal{E}[\![x]\!]\langle\sigma_1, \varphi_1\rangle = \sigma_1\ x$$

and

$$\mathcal{E}[\![x]\!]\langle\sigma_2, \varphi_2\rangle = \sigma_2\ x$$

The analysis gives

$$\Gamma \vdash x : \Gamma(x)$$

and the case follows from condition *(6.4.1)*.

Case s : The semantics evaluates the expression as

$$\mathcal{E}[\![s]\!]\langle\sigma_1, \varphi_1\rangle = \varphi_1\ s\ 0$$

and

$$\mathcal{E}[\![s]\!]\langle\sigma_2,\varphi_2\rangle = \varphi_2\ s\ 0$$

The analysis gives

$$\Gamma \vdash s : \Gamma(s)$$

and the case follows from condition *(6.4.2)*.

Case `not e` : The semantics evaluates the expression as

$$\mathcal{E}[\![\texttt{not}\ e]\!]\langle\sigma_1,\varphi_1\rangle = (\overline{\texttt{not}}\ v_1, t_1)$$

where $\mathcal{E}[\![e]\!]\langle\sigma_1,\varphi_1\rangle = (v_1,t_1)$, and

$$\mathcal{E}[\![\texttt{not}\ e]\!]\langle\sigma_2,\varphi_2\rangle = (\overline{\texttt{not}}\ v_2, t_2)$$

where $\mathcal{E}[\![e]\!]\langle\sigma_2,\varphi_2\rangle = (v_2,t_2)$. The analysis gives

$$\frac{\Gamma \vdash e : \delta}{\Gamma \vdash \texttt{not}\ e : \delta}$$

the case follows from the induction hypothesis.

Case e_1 `and` e_2 : The semantics evaluates the expression as

$$\mathcal{E}[\![e_1\ \texttt{and}\ e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1\ \overline{\texttt{and}}\ v_1', t_1 \cup t_1')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_1,\varphi_1\rangle = (v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1',t_1')$, and

$$\mathcal{E}[\![e_1\ \texttt{and}\ e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2\ \overline{\texttt{and}}\ v_2', t_2 \cup t_2')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_2,\varphi_2\rangle = (v_2,t_2)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2',t_2')$. The analysis gives

$$\frac{\Gamma \vdash e_1 : \delta \qquad \Gamma \vdash e_2 : \delta'}{\Gamma \vdash e_1\ \texttt{and}\ e_2 : \delta \otimes \delta'}$$

We have $t_1 \in \delta$, $t_2 \in \delta$, $t_1' \in \delta'$ and $t_2' \in \delta'$ from the induction hypothesis. Hence $t_1 \cup t_1' \in \delta \otimes \delta'$ and $t_2 \cup t_2' \in \delta \otimes \delta'$.

Case e_1 `or` e_2 **:** The semantics evaluates the expression as

$$\mathcal{E}[\![e_1 \ \texttt{or} \ e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1 \ \overline{\texttt{or}} \ v_1', t_1 \cup t_1')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_1,\varphi_1\rangle = (v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1',t_1')$, and

$$\mathcal{E}[\![e_1 \ \texttt{or} \ e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2 \ \overline{\texttt{or}} \ v_2', t_2 \cup t_2')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_2,\varphi_2\rangle = (v_2,t_2)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2',t_2')$. The analysis gives

$$\frac{\Gamma \vdash e_1 : \delta \qquad \Gamma \vdash e_2 : \delta'}{\Gamma \vdash e_1 \ \texttt{or} \ e_2 : \delta \otimes \delta'}$$

We have $t_1 \in \delta$, $t_2 \in \delta$, $t_1' \in \delta'$ and $t_2' \in \delta'$ from the induction hypothesis. Hence $t_1 \cup t_1' \in \delta \otimes \delta'$ and $t_2 \cup t_2' \in \delta \otimes \delta'$.

Case e_1 `xor` e_2 **:** The semantics evaluates the expression as

$$\mathcal{E}[\![e_1 \ \texttt{xor} \ e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1 \ \overline{\texttt{xor}} \ v_1', t_1 \cup t_1')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_1,\varphi_1\rangle = (v_1,t_1)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_1,\varphi_1\rangle = (v_1',t_1')$, and

$$\mathcal{E}[\![e_1 \ \texttt{xor} \ e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2 \ \overline{\texttt{xor}} \ v_2', t_2 \cup t_2')$$

where $\mathcal{E}[\![e_1]\!]\langle\sigma_2,\varphi_2\rangle = (v_2,t_2)$ and $\mathcal{E}[\![e_2]\!]\langle\sigma_2,\varphi_2\rangle = (v_2',t_2')$. The analysis gives

$$\frac{\Gamma \vdash e_1 : \delta \qquad \Gamma \vdash e_2 : \delta'}{\Gamma \vdash e_1 \ \texttt{xor} \ e_2 : \delta \otimes \delta'}$$

We have $t_1 \in \delta$, $t_2 \in \delta$, $t_1' \in \delta'$ and $t_2' \in \delta'$ from the induction hypothesis. Hence $t_1 \cup t_1' \in \delta \otimes \delta'$ and $t_2 \cup t_2' = \delta \otimes \delta'$. □

The correctness of the analysis for statements is stated in the following subject reduction result.

Lemma 6.5 (Subject Reduction) For a statement ss that is evaluated by the semantics in the states σ and φ as $t \vdash \langle ss,\sigma,\varphi\rangle \Rightarrow \langle ss',\sigma',\varphi'\rangle$ we have

$$\left.\begin{array}{l}\Gamma,\delta \vdash ss : \tau \ \wedge \ t \in \delta \ \wedge \\ \forall x : \sigma \ x|_{Time} \in \Gamma \ x \ \wedge \\ \forall s : \varphi \ s \ 0|_{Time} \in \Gamma \ s\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\exists \tau' : \Gamma,\delta \vdash ss' : \tau' \ \wedge \\ \tau \sqsupseteq \tau'\end{array}\right.$$

Proof. The proof proceeds by induction on the shape of the execution tree.

Case Local Variable Assignment: The semantics evaluates the statement as

$$t \vdash \langle x := e, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi \rangle$$

where $\mathcal{E}[\![e]\!]\langle \sigma, \varphi \rangle = (v, t')$ and $\sigma' = \sigma[x \mapsto (v, t \cup t')]$. The analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad \delta \otimes \delta' \subseteq \Gamma(x)}{\Gamma, \delta \vdash x := e : [x \mapsto \delta \otimes \delta']}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

now $\tau = [x \mapsto \delta \otimes \delta']$ and $\tau' = \bot$, hence $\tau \sqsupseteq \tau'$.

Case Signal Assignment: The semantics evaluates the statement as

$$t \vdash \langle s <= e, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi' \rangle$$

where $\mathcal{E}[\![e]\!]\langle \sigma, \varphi \rangle = (v, t')$ and $\varphi' = \varphi^{[1]}[s \mapsto (v, t \cup t')]$. The analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad \delta \otimes \delta' \subseteq \Gamma(s)}{\Gamma, \delta \vdash s <= e : [s \mapsto \delta \otimes \delta']}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

now $\tau = [s \mapsto \delta \otimes \delta']$ and $\tau' = \bot$, hence $\tau \sqsupseteq \tau'$.

Case Skip: The semantics evaluates the statement as

$$t \vdash \langle \texttt{null}, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi \rangle$$

The analysis gives

$$\Gamma, \delta \vdash \texttt{null} : \bot$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

now $\tau = \bot$ and $\tau' = \bot$, and the case follows as $\tau = \tau'$.

Case Composition: In one rule the semantics evaluates the statement as

$$\frac{t \vdash \langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma', \varphi'\rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi'\rangle}$$

where $ss_1' \in Stmt$. The analysis gives

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

and the analysis of ss_1' gives $\Gamma, \delta \vdash ss_1' : \tau_1'$. Now applying the rule for composition we can derive

$$\frac{\Gamma, \delta \vdash ss_1' : \tau_1' \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1'; ss_2 : \tau_1' \curlyvee \tau_2}$$

now it follows from the induction hypothesis for $t \vdash \langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle ss_1', \sigma', \varphi'\rangle$ and $\Gamma, \delta \vdash ss_1 : \tau_1$ that we can choose τ_1' such that $\Gamma, \delta \vdash ss_1' : \tau_1'$ and $\tau_1 \sqsupseteq \tau_1'$, and clearly $\tau_1 \curlyvee \tau_2 \sqsupseteq \tau_1' \curlyvee \tau_2$ which proves the rule. In the other rule the semantics evaluates the statement as

$$\frac{t \vdash \langle ss_1, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi\rangle \Rightarrow \langle ss_2, \sigma', \varphi'\rangle}$$

The analysis gives

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

and

$$\Gamma, \delta \vdash ss_2 : \tau_2$$

now $\tau_1 \curlyvee \tau_2 \sqsupseteq \tau_2$.

Case Low Conditional: The semantics evaluates the statement as $t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t \cup t^e] ss', \sigma, \varphi\rangle$, where $ss' = ss_1$ when $E[\![e]\!]\langle \sigma, \varphi\rangle = ('1', t^e)$ and $ss' = ss_2$ when $E[\![e]\!]\langle \sigma, \varphi\rangle = ('0', t^e)$. The analysis gives

$$\frac{FV(e) \cup FS(e) \subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta^e \qquad \Gamma, \delta \otimes \delta^e \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \otimes \delta^e \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau_1 \Cup \tau_2}$$

and

$$\frac{\Gamma, \delta \otimes \{t^e\} \vdash ss' : \tau'}{\Gamma, \delta \vdash [t^e]ss' : \tau'}$$

from Lemma 6.4 we get that $t^e \in \delta^e$. Thus $\delta \otimes \{t^e\} \subseteq \delta \otimes \delta^e$ and we get that either $\tau_1 \sqsupseteq \tau'$ or $\tau_2 \sqsupseteq \tau'$. Now it follows that $\tau_1 \Cup \tau_2 \sqsupseteq \tau'$.

Case High Conditional: The semantics evaluates the statement as $t \vdash \langle \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t \cup t^e]ss', \sigma, \varphi\rangle$, where $ss' = ss_1$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('1', t^e)$ and $ss' = ss_2$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('0', t^e)$. The analysis gives

$$\frac{FV(e) \cup FS(e) \not\subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau}{\Gamma, \delta \vdash \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2 : \tau}$$

where $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$. Now from Lemma 6.4 we get that $t^e \in \delta^e$ and

$$\frac{\Gamma, \delta \otimes \{t^e\} \vdash ss' : \tau'}{\Gamma, \delta \vdash [t^e]ss' : \tau'}$$

it follows that $\tau \sqsupseteq \tau'$.

Case Branch: In one rule the semantics evaluates the statement as

$$\frac{t \cup t^e \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle}{t \vdash \langle [t^e]ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle}$$

where $ss' \in Stmt$. The analysis gives

$$\frac{\Gamma, \delta \otimes \{t^e\} \vdash ss : \tau}{\Gamma, \delta \vdash [t^e]ss : \tau}$$

and

$$\frac{\Gamma, \delta \otimes \{t^e\} \vdash ss' : \tau'}{\Gamma, \delta \vdash [t^e]ss' : \tau'}$$

and the case follows from applying the induction hypothesis. In the other rule the semantics evaluates the statement as

$$\frac{t \cup t^e \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{t \vdash \langle [t^e]ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}$$

again the analysis gives

$$\frac{\Gamma, \delta \otimes \{t^e\} \vdash ss : \tau}{\Gamma, \delta \vdash [t^e]ss : \tau}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

and the case follows because $\tau' = \bot$, hence $\tau \sqsupseteq \tau'$. □

Now we stated that the result of the analysis of a program contain all possible modifications of timing behaviours in the associated states. First it is shown for semantical evaluations in one step. Below we generalise for executions of arbitrary length.

Lemma 6.6 (One step semantic correctness of time delays) For a statement ss that is evaluated by the semantics in the states σ and φ as $t \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle$ we have

$$\left.\begin{array}{l} \Gamma, \delta \vdash ss : \tau \wedge \\ \Gamma, \delta \vdash ss' : \tau' \wedge \\ t \in \delta \wedge \\ \sigma\ x|_{Time} \in \Gamma\ x \wedge \\ \varphi\ s\ 0|_{Time} \in \Gamma\ s \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \sigma'\ x|_{Time} \in \Gamma\ x\ \wedge\ \varphi'\ s\ 1|^{--}_{Time} \in \Gamma\ s \wedge \\ \forall x' : x' \notin dom(\tau)\backslash dom(\tau') \Rightarrow \sigma\ x' = \sigma'\ x' \wedge \\ \forall x'' : x'' \in dom(\tau)\backslash dom(\tau') \Rightarrow \sigma'\ x''|_{Time} \in \tau\ x'' \wedge \\ \forall s' : s' \notin dom(\tau)\backslash dom(\tau') \Rightarrow \varphi\ s'\ 1 = \varphi'\ s'\ 1 \wedge \\ \forall s'' : s'' \in dom(\tau)\backslash dom(\tau') \Rightarrow \varphi'\ s''\ 1|^{--}_{Time} \in \tau\ s'' \end{array}\right.$$

Proof. By induction on the shape of the derivation tree.

Case Local Variable Assignment: The semantics evaluates the statement as

$$t \vdash \langle x := e, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma, \varphi\rangle = (v, t')$ and $\sigma' = \sigma[x \mapsto (v, t \cup t')]$, while the analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad \delta \cup \delta' \subseteq \Gamma(x)}{\Gamma, \delta \vdash x := e : \tau}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \tau'$$

where $\tau = [x \mapsto \delta \otimes \delta']$ and $\tau' = \bot$. From Lemma 6.4 it follows that $t' \in \delta'$. Now $\sigma'\ x|_{Time} \in \Gamma\ x = \tau\ x$, $dom(\tau) = \{x\}$, $dom(\tau') = \emptyset$ and $dom(\tau)\backslash dom(\tau') = \{x\}$.

Case Signal Assignment: The semantics evaluates the statement as

$$t \vdash \langle s <= e, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi' \rangle$$

where $\mathcal{E}[\![e]\!]\langle \sigma, \varphi \rangle = (v, t')$ and $\varphi' = \varphi^{[1]}[s \mapsto (v, t \cup t')]$, while the analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad (\delta \otimes \delta')^{--} \subseteq \Gamma(s)}{\Gamma, \delta \vdash s <= e : \tau}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \tau'$$

where $\tau = [s \mapsto (\delta \otimes \delta')^{--}$ and $\tau' = \bot$. From Lemma 6.4 it follows that $t' \in \delta'$. Now $\varphi'\ s\ 1|^{--}_{Time} \delta\Gamma\ s = \tau\ s$, $dom(\tau) = \{s\}$, $dom(\tau') = \emptyset$ and $dom(\tau) \backslash dom(\tau') = \{s\}$.

Case Skip: The semantics evaluates the statement as

$$t \vdash \langle \texttt{null}, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma, \varphi \rangle$$

while the analysis gives

$$\Gamma, \delta \vdash \texttt{null} : \bot$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

so that the case follows.

Case Composition: There are two subcases. One is where the semantics evaluates the statement as

$$\frac{t \vdash \langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle ss_1', \sigma', \varphi' \rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi \rangle \Rightarrow \langle ss_1'; ss_2, \sigma', \varphi' \rangle}$$

where $ss_1' \in Stmt$, while the analysis give

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

Now assuming that $\Gamma, \delta \vdash ss_1' : \tau_1'$ we can apply rule [**Composition**] with the same derivation tree as above for ss_2 and get

$$\frac{\Gamma, \delta \vdash ss_1' : \tau_1' \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1'; ss_2 : \tau_1' \curlyvee \tau_2}$$

Applying the induction hypothesis on $t \vdash \langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle ss_1', \sigma', \varphi' \rangle$ we get

$$\begin{array}{l} \sigma'\ x|_{Time} \in \Gamma\ x\ \wedge\ \varphi'\ s\ 1|^{--}_{Time} \in \Gamma\ s\ \wedge \\ \forall x : x \notin dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \sigma\ x = \sigma'\ x\ \wedge \\ \forall x : x \in dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \sigma'\ x|_{Time} \in \tau_1\ x\ \wedge \\ \forall s : s \notin dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \varphi\ s\ 1 = \varphi'\ s\ 1\ \wedge \\ \forall s : s \in dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \varphi'\ s\ 1|^{--}_{Time} \in \tau_1\ s \end{array}$$

and since $dom(\tau_1 \curlyvee \tau_2) \backslash dom(\tau_1' \curlyvee \tau_2) = dom(\tau_1) \backslash dom(\tau_1')$ the case follows. The other subcase is where the first step in the derivation sequence was evaluated as

$$\frac{t \vdash \langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi' \rangle}{t \vdash \langle ss_1; ss_2, \sigma, \varphi \rangle \Rightarrow \langle ss_2, \sigma', \varphi' \rangle}$$

and again the analysis gives

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

and

$$\Gamma, \delta \vdash ss_2 : \tau_2$$

Applying the induction hypothesis on $t \vdash \langle ss_1, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi' \rangle$ we get

$$\begin{array}{l} \sigma'\ x|_{Time} \in \Gamma\ x\ \wedge\ \varphi'\ s\ 1|^{--}_{Time} \in \Gamma\ s\ \wedge \\ \forall x : x \notin dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \sigma\ x = \sigma'\ x\ \wedge \\ \forall x : x \in dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \sigma'\ x|_{Time} \in \tau_1\ x\ \wedge \\ \forall s : s \notin dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \varphi\ s\ 1 = \varphi'\ s\ 1\ \wedge \\ \forall s : s \in dom(\tau_1) \backslash dom(\tau_1') \Rightarrow \varphi'\ s\ 1|^{--}_{Time} \in \tau_1\ s \end{array}$$

where $\tau_1' = \bot$ because $\Gamma, \delta \vdash \texttt{final} : \bot$. Since $dom(\tau_1 \curlyvee \tau_2) \backslash dom(\tau_2) = dom(\tau_1)$ the case follows.

Case Low Conditional: The semantics evaluates the statement as $t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t \cup t^e]ss', \sigma, \varphi\rangle$, where $ss' = ss_1$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('1', t^e)$ and $ss' = ss_2$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('0', t^e)$. The analysis gives

$$\frac{FV(e) \cup FS(e) \subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau_1 \Cup \tau_2}$$

now as the states are unchanged and the case follows.

Case High Conditional: The semantics evaluates the statement as $t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma, \varphi\rangle \Rightarrow \langle [t \cup t^e]ss', \sigma, \varphi\rangle$, where $ss' = ss_1$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('1', t^e)$ and $ss' = ss_2$ when $E[\![e]\!]\langle\sigma, \varphi\rangle = ('0', t^e)$. The analysis gives

$$\frac{FV(e) \cup FS(e) \not\subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau}$$

where $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$. Now as the states are unchanged and the case follows.

Case Branch: There are two subcases. One is where the semantics evaluates the statement

$$\frac{t \cup t' \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle ss', \sigma', \varphi'\rangle}{t \vdash \langle [t']ss, \sigma, \varphi\rangle \Rightarrow \langle [t']ss', \sigma', \varphi'\rangle}$$

while the analysis gives

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss : \tau}{\Gamma, \delta \vdash [t']ss : \tau}$$

and

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss' : \tau'}{\Gamma, \delta \vdash [t']ss' : \tau'}$$

The desired result then follows from applying the induction hypothesis. In the other subcase the semantics evaluates the statement as

$$\frac{t \cup t' \vdash \langle ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}{t \vdash \langle [t']ss, \sigma, \varphi\rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi'\rangle}$$

and again the analysis

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss : \tau}{\Gamma, \delta \vdash [t']ss : \tau}$$

and

$$\Gamma, \delta \vdash \texttt{final} : \bot$$

The desired result follows from applying the induction hypothesis on $t \cup t' \vdash \langle ss, \sigma, \varphi \rangle \Rightarrow \langle \texttt{final}, \sigma', \varphi' \rangle$. Where the analysis for $t \cup t' \vdash \langle \texttt{final}, \sigma', \varphi' \rangle$ gives $\Gamma, \delta \otimes \{t'\} \vdash \texttt{final} : \bot$. This completes the proof of Lemma 6.6. □

We generalise the result above by showing that the result of the analysis of a program contain all modifications of timing behaviours in executions of arbitrary length.

Lemma 6.7 (Semantic correctness of time delays) For a statement ss that is evaluated by the semantics in the states σ and φ as $t \vdash \langle ss, \sigma, \varphi \rangle \Rightarrow^* \langle \texttt{final}, \sigma', \varphi' \rangle$ we have

$$\left.\begin{array}{l} \Gamma, \delta \vdash ss : \tau \ \wedge \ t \in \delta \ \wedge \\ \sigma \ x|_{Time} \in \Gamma \ x \ \wedge \\ \varphi \ s \ 0|_{Time} \in \Gamma \ s \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \forall x' : x' \notin dom(\tau) \Rightarrow \sigma \ x' = \sigma' \ x' \ \wedge \\ \forall x'' : x'' \in dom(\tau) \Rightarrow \sigma' \ x''|_{Time} \in \tau \ x'' \ \wedge \\ \forall s' : s' \notin dom(\tau) \Rightarrow \varphi \ s' \ 1 = \varphi' \ s' \ 1 \ \wedge \\ \forall s'' : s'' \in dom(\tau) \Rightarrow \varphi' \ s'' \ 1|^{--}_{Time} \in \tau \ s'' \end{array}\right.$$

Proof. The Lemma follows from the transitive closure of Lemma 6.6. □

In the below results of secure statements we need to consider the first coming statement in the evaluation. This is defined in the following definition.

Definition 6.8 (First statement) The function $first$ on a statements is defined as

$$\begin{array}{rcl} first_t(x := e) & = & x := e \\ first_t(s <= e) & = & s <= e \\ first_t(\texttt{null}) & = & \texttt{null} \\ first_t(ss_1; ss_2) & = & first_t(ss_1) \\ first_t(\texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2) & = & [t]\texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 \\ first_t([t']ss) & = & first_{t \cup t'}(ss) \end{array}$$

Now we stated that the successful analysis of a program provides the guarantees stated in the absence of timing leaks property. First it is shown for semantical evaluations in one step. Below we generalise for executions of arbitrary length.

Lemma 6.9 (One step Secure Statements) For every statement ss where $first_t(ss) \neq [t']\texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2$ or $FV(e) \cup FS(e) \subseteq \mathcal{L}$ then if the semantics evaluates the statement in the states σ_1 and φ_1 as $t \vdash \langle ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_1', \sigma_1', \varphi_1'\rangle$ and in states σ_2 and φ_2 as $t \vdash \langle ss, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss_2', \sigma_2', \varphi_2'\rangle$ we have

$$\begin{array}{ll} \Gamma, \delta \vdash ss : \tau \ \wedge \ t \in \delta \ \wedge & \\ \sigma_1\ x|_{Time} = \sigma_2\ x|_{Time} \in \Gamma\ x \ \wedge & (6.9.1) \\ \varphi_1\ s\ 0|_{Time} = \varphi_2\ s\ 0|_{Time} \in \Gamma\ s \ \wedge & (6.9.2) \\ \forall s : s \in \mathcal{L} \Rightarrow \varphi_1\ s\ 0 = \varphi_2\ s\ 0 & (6.9.3) \end{array}$$

implies

$$\begin{array}{ll} ss_1' = ss_2' \ \wedge & (6.9.4) \\ \sigma_1'\ x|_{Time} = \sigma_2'\ x|_{Time} \ \wedge & (6.9.5) \\ \varphi_1'\ s\ 1|_{Time} = \varphi_2'\ s\ 1|_{Time} & (6.9.6) \end{array}$$

otherwise if $first_t(ss) = [t']\texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2$ and $FV(e) \cup FS(e) \not\subseteq \mathcal{L}$ then if the semantics evaluates the statement in the states σ_1 and φ_1 as $t' \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_1, \varphi_1\rangle \Rightarrow^* \langle \texttt{final}, \sigma_1', \varphi_1'\rangle$ and in states σ_2 and φ_2 as $t' \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_2, \varphi_2\rangle \Rightarrow^* \langle \texttt{final}, \sigma_2', \varphi_2'\rangle$ we have

$$\begin{array}{ll} \Gamma, \delta \vdash ss : \tau \ \wedge \ t \in \delta \ \wedge & \\ \sigma_1\ x|_{Time} = \sigma_2\ x|_{Time} \in \Gamma\ x \ \wedge & (6.9.7) \\ \varphi_1\ s\ 0|_{Time} = \varphi_2\ s\ 0|_{Time} \in \Gamma\ s \ \wedge & (6.9.8) \\ \forall s : s \in \mathcal{L} \Rightarrow \varphi_1\ s\ 0 = \varphi_2\ s\ 0 & (6.9.9) \end{array}$$

implies

$$\begin{array}{ll} \sigma_1'\ x|_{Time} = \sigma_2'\ x|_{Time} \ \wedge & (6.9.10) \\ \varphi_1'\ s\ 1|_{Time} = \varphi_2'\ s\ 1|_{Time} & (6.9.11) \end{array}$$

Proof. The proof proceeds by induction on the shape of the derivation sequence.

Case Local Variable Assignment: The semantics evaluates the statement as

$$t \vdash \langle x := e, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1[x \mapsto (v_1, t \cup t_1)], \varphi_1\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma_1, \varphi_1\rangle = (v_1, t_1)$, and

$$t \vdash \langle x := e, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma_2[x \mapsto (v_2, t \cup t_2)], \varphi_2\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma_2, \varphi_2\rangle = (v_2, t_2)$. The analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad \delta \otimes \delta' \subseteq \Gamma(x)}{\Gamma, \delta \vdash x := e : [x \mapsto \delta \otimes \delta']}$$

and from Lemma 6.4 we get $t_1 = t_2 \in \delta'$ and $t \cup t_1 = t \cup t_2 \in \delta \otimes \delta'$. Hence $\sigma_1' \ x|_{Time} = \sigma_2' \ x|_{Time}$ which proves (6.9.5). Finally, (6.9.4) and (6.9.6) follow directly from the evaluation and the analysis.

Case Signal Assignment: The semantics evaluates the statement as

$$t \vdash \langle s <= e, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1, \varphi_1^{[1]}[s \mapsto (v_1, t \cup t_1)]\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma_1, \varphi_1\rangle = (v_1, t_1)$, and

$$t \vdash \langle s <= e, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma_2, \varphi_2^{[1]}[s \mapsto (v_2, t \cup t_2)]\rangle$$

where $\mathcal{E}[\![e]\!]\langle\sigma_2, \varphi_2\rangle = (v_2, t_2)$. The analysis gives

$$\frac{\Gamma \vdash e : \delta' \qquad (\delta \otimes \delta')^{--} \subseteq \Gamma(s)}{\Gamma, \delta \vdash s <= e : [s \mapsto (\delta \otimes \delta')^{--}]}$$

and from Lemma 6.4 we get $t_1 = t_2 \in \delta'$, and $t \cup t_1 = t \cup t_2 \in \delta \otimes \delta'$. Hence $\varphi_1' \ s \ 1|_{Time} = \varphi_2' \ s \ 1|_{Time}$ which proves (6.9.6). Finally, (6.9.4) and (6.9.5) follow directly from the evaluation and the analysis.

Case Skip: The semantics evaluates the statement as

$$t \vdash \langle \texttt{null}, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1, \varphi_1\rangle$$

and

$$t \vdash \langle \texttt{null}, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma_2, \varphi_2\rangle$$

The analysis gives

$$\Gamma, \delta \vdash \texttt{null} : \bot$$

and the case follows.

Case Composition: There are two subcases. One is where the semantics evaluates the statement as

$$\frac{t \vdash \langle ss_1, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_1', \sigma_1', \varphi_1'\rangle}{t \vdash \langle ss_1; ss_2, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_1'; ss_2, \sigma_1', \varphi_1'\rangle}$$

and

$$\frac{t \vdash \langle ss_1, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss_2', \sigma_2', \varphi_2'\rangle}{t \vdash \langle ss_1; ss_2, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss_2'; ss_2, \sigma_2', \varphi_2'\rangle}$$

where $ss_1' \in Stmt$. The analysis gives

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

and the case follows from applying the induction hypothesis. In the other subcase the semantics evaluates the statement as

$$\frac{t \vdash \langle ss_1, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma_1', \varphi_1'\rangle}{t \vdash \langle ss_1; ss_2, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss_2, \sigma_1', \varphi_1'\rangle}$$

and

$$\frac{t \vdash \langle ss_1, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma_2', \varphi_2'\rangle}{t \vdash \langle ss_1; ss_2, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss_2, \sigma_2', \varphi_2'\rangle}$$

again the analysis gives

$$\frac{\Gamma, \delta \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash ss_1; ss_2 : \tau_1 \curlyvee \tau_2}$$

and the case follows from applying the induction hypothesis.

Case Low Conditional: The semantics evaluates the statement as

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_1, \varphi_1\rangle \Rightarrow \langle [t^e]ss', \sigma_1', \varphi_1'\rangle$$

and

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_2, \varphi_2\rangle \Rightarrow \langle [t'^e]ss'', \sigma_2', \varphi_2'\rangle$$

The analysis gives

$$\frac{FV(e) \cup FS(e) \subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau_1 \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau_2}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau_1 \Cup \tau_2}$$

now it follow from $FV(e) \cup FS(e) \subseteq \mathcal{L}$ and (6.9.3) that $\mathcal{E}[\![e]\!]\langle\sigma_1, \varphi_1\rangle = \mathcal{E}[\![e]\!]\langle\sigma_2, \varphi_2\rangle$. Therefore we have $t^e = t'^e$ and $ss' = ss''$. Because the states are unchanged the case follows.

Case High Conditional: The semantics evaluates the statement as

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_1, \varphi_1\rangle \Rightarrow^* \langle \texttt{final}, \sigma_1', \varphi_1'\rangle$$

and

$$t \vdash \langle \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2, \sigma_2, \varphi_2\rangle \Rightarrow^* \langle \texttt{final}, \sigma_2', \varphi_2'\rangle$$

The analysis gives

$$\frac{FV(e) \cup FS(e) \not\subseteq \mathcal{L} \qquad \Gamma \vdash e : \delta' \qquad \Gamma, \delta \otimes \delta' \vdash ss_1 : \tau \qquad \Gamma, \delta \otimes \delta' \vdash ss_2 : \tau}{\Gamma, \delta \vdash \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2 : \tau}$$

where $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$ and from Lemma 6.7 we get

$$\begin{array}{ll} (i) & x' \notin dom(\tau) \Rightarrow \sigma_1\ x' = \sigma_1'\ x' \wedge \\ (ii) & x \in dom(\tau) \Rightarrow \sigma_1'\ x|_{Time} \in \tau\ x \wedge \\ (iii) & s' \notin dom(\tau) \Rightarrow \varphi_1\ s'\ 1 = \varphi_1'\ s'\ 1 \wedge \\ (iv) & s \in dom(\tau) \Rightarrow \varphi_1'\ s\ 1|^{--}_{Time} \in \tau\ s \end{array}$$

and

$$\begin{array}{ll} (v) & x' \notin dom(\tau) \Rightarrow \sigma_2\ x' = \sigma_2'\ x' \wedge \\ (vi) & x \in dom(\tau) \Rightarrow \sigma_2'\ x|_{Time} \in \tau\ x \wedge \\ (vii) & s' \notin dom(\tau) \Rightarrow \varphi_2\ s'\ 1 = \varphi_2'\ s'\ 1 \wedge \\ (viii) & s \in dom(\tau) \Rightarrow \varphi_2'\ s\ 1|^{--}_{Time} \in \tau\ s \end{array}$$

Since the timing behaviour for each branch is identical τ, each timing effect has a unique timing delay for all variables and signals (due to the condition $\forall n : n \in dom(\tau) \wedge |\tau\ n| \leq 1$), and the states projected on to the timing delays are equivalent $\sigma_1\ x|_{Time} = \sigma_2\ x|_{Time}$, we can combine (i) and (v) to obtain

$$\forall x' : x' \notin dom(\tau) \Rightarrow \sigma_1'\ x'|_{Time} = \sigma_2'\ x'|_{Time}$$

and similarly (ii) and (vi) to obtain

$$\forall x : x \in dom(\tau) \Rightarrow \{\sigma_1'\ x|_{Time}\} = \tau\ x = \{\sigma_2'\ x|_{Time}\}$$

combining these we get

$$\forall x : \sigma'_1\ x|_{Time} = \sigma'_2\ x|_{Time}$$

In a similar way one can combine (iii), (iv), (vii) and $(viii)$ to get

$$\forall s : \varphi'_1\ s\ 1|_{Time} = \varphi'_2\ s\ 1|_{Time}$$

Hence the case follows.

Case Branch: There are two subcases. One is where the semantics evaluates the statement as

$$\frac{t \cup t' \vdash \langle ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss'_1, \sigma'_1, \varphi'_1\rangle}{t \vdash \langle [t']ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle ss'_1, \sigma'_1, \varphi'_1\rangle}$$

and

$$\frac{t \cup t' \vdash \langle ss, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss'_2, \sigma'_2, \varphi'_2\rangle}{t \vdash \langle [t']ss, \sigma_2, \varphi_2\rangle \Rightarrow \langle ss'_2, \sigma'_2, \varphi'_2\rangle}$$

The analysis gives

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss : \tau}{\Gamma, \delta \vdash [t']ss : \tau}$$

and the case follows from applying the induction hypothesis. In the other subcase the semantics evaluates the statement as

$$\frac{t \cup t' \vdash \langle ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma'_1, \varphi'_1\rangle}{t \vdash \langle [t']ss, \sigma_1, \varphi_1\rangle \Rightarrow \langle \texttt{final}, \sigma'_1, \varphi'_1\rangle}$$

and

$$\frac{t \cup t' \vdash \langle ss, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma'_2, \varphi'_2\rangle}{t \vdash \langle [t']ss, \sigma_2, \varphi_2\rangle \Rightarrow \langle \texttt{final}, \sigma'_2, \varphi'_2\rangle}$$

and again the analysis gives

$$\frac{\Gamma, \delta \otimes \{t'\} \vdash ss : \tau}{\Gamma, \delta \vdash [t']ss : \tau}$$

now the case follows from applying the induction hypothesis. □

We generalise the static security guarantee of the analysis of statements for executions of arbitrary length.

Lemma 6.10 (Secure Statements) For every statement ss that is evaluated by the semantics in the states σ_1 and φ_1 as $t \vdash \langle ss, \sigma_1, \varphi_1\rangle \Rightarrow^* \langle \texttt{final}, \sigma_1', \varphi_1'\rangle$ and in the states σ_2 and φ_2 as $t \vdash \langle ss, \sigma_2, \varphi_2\rangle \Rightarrow^* \langle \texttt{final}, \sigma_2', \varphi_2'\rangle$ we have

$$\begin{array}{ll} \Gamma, \delta \vdash ss : \tau \ \wedge \ t \in \delta \ \wedge & \\ \sigma_1\ x|_{Time} = \sigma_2\ x|_{Time} \in \Gamma\ x \ \wedge & (6.10.1) \\ \varphi_1\ s\ 0|_{Time} = \varphi_2\ s\ 0|_{Time} \in \Gamma\ s & (6.10.2) \end{array}$$

implies

$$\begin{array}{ll} \sigma_1'\ x|_{Time} = \sigma_2'\ x|_{Time} \ \wedge & (6.10.3) \\ \varphi_1'\ s\ 1|_{Time} = \varphi_2'\ s\ 1|_{Time} & (6.10.4) \end{array}$$

Proof. The proof proceeds by induction on the length of the derivation sequence

Case Local Variable Assignment, Signal Assignment, Skip and Conditional follow from Lemma 6.9.

Case Composition: From Lemma 6.6 we have for the first evaluation step for both semantical rules for composition that $\sigma_1'\ x|_{Time} = \sigma_2'\ x|_{Time} \in \Gamma\ x$. As the semantics for statements does not modify the present value of signals we have $\varphi_1'\ s\ 0|_{Time} = \varphi_2'\ s\ 0|_{Time} \in \Gamma\ s$. Both subcases follows from the induction hypothesis to the remaining part of the derivation sequence, which is obviously shorter.

Case Branch: There are two cases. The case where the semantics evaluated the statement in one step the case follows from Lemma 6.9. The case where the branch statement is not evaluated in one step the we have that $\sigma_1'\ x|_{Time} = \sigma_2'\ x|_{Time} \in \Gamma\ x$ from Lemma 6.6. As the semantics for statements does not modify the present value of signals we have $\varphi_1'\ s\ 0|_{Time} = \varphi_2'\ s\ 0|_{Time} \in \Gamma\ s$. The case follows from the induction hypothesis to the remaining part of the derivation sequence, which is obviously shorter. □

Finally, we show the main result for processes.

Theorem 6.11 (Secure Process) For every concurrent statement css where the semantics evaluates each process i in the states σ_{1i} and φ_{1i} as $\|_{i\in I}\ \langle css_i, \sigma_{1i}, \varphi_{1i}\rangle$

$\Longrightarrow_H^* \Longrightarrow_A \|_{i\in I}\ \langle css'_{1i}, \sigma'_{1i}, \varphi'_{1i}\rangle$ and in the states σ_{2i} and φ_{2i} as $\|_{i\in I}\ \langle css_i, \sigma_{2i}, \varphi_{2i}\rangle$ $\Longrightarrow_H^* \Longrightarrow_A \|_{i\in I}\ \langle css'_{2i}, \sigma'_{2i}, \varphi'_{2i}\rangle$ implies

$$\begin{array}{l}\Gamma \vdash_c css : \tau\ \wedge\\ \sigma_{1i}\ x|_{Time} = \sigma_{2i}\ x|_{Time} \in \Gamma\ x\ \wedge\\ \varphi_{1i}\ s\ 0|_{Time} = \varphi_{2i}\ s\ 0|_{Time} \in \Gamma\ s\ \wedge\\ \varphi'_{1i}\ s\ 1|_{Time} = \varphi'_{2i}\ s\ 1|_{Time}\ \wedge\\ \forall s : s \in \mathcal{L} \Rightarrow \varphi_1\ s\ 0 = \varphi_2\ s\ 0\end{array}$$

implies

$$\begin{array}{l}css'_{1i} = css'_{2i}\ \wedge\\ \sigma'_{1i}\ x|_{Time} = \sigma'_{2i}\ x|_{Time} \in \Gamma\ x\ \wedge\\ \varphi'_{1i}\ s\ 0|_{Time} = \varphi'_{2i}\ s\ 0|_{Time} \in \Gamma\ s\ \wedge\\ \varphi'_{1i}\ s\ 1|_{Time} = \varphi'_{2i}\ s\ 1|_{Time} = undef\end{array}$$

Proof. We introduce a configuration on the evaluation sequences, such that

$$\|_{i\in I}\ \langle css_i, \sigma_{1i}, \varphi_{1i}\rangle \Longrightarrow_H^* \|_{i\in I}\ \langle css''_{1i}, \sigma''_{1i}, \varphi''_{1i}\rangle \Longrightarrow_A \|_{i\in I}\ \langle css'_{1i}, \sigma'_{1i}, \varphi'_{1i}\rangle$$

and

$$\|_{i\in I}\ \langle css_i, \sigma_{2i}, \varphi_{2i}\rangle \Longrightarrow_H^* \|_{i\in I}\ \langle css''_{2i}, \sigma''_{2i}, \varphi''_{2i}\rangle \Longrightarrow_A \|_{i\in I}\ \langle css'_{2i}, \sigma'_{2i}, \varphi'_{2i}\rangle$$

For the evaluation sequences $\|_{i\in I}\ \langle css_i, \sigma_{1i}, \varphi_{1i}\rangle \Longrightarrow_H^* \|_{i\in I}\ \langle css''_{1i}, \sigma''_{1i}, \varphi''_{1i}\rangle$ and $\|_{i\in I}\ \langle css_i, \sigma_{2i}, \varphi_{2i}\rangle \Longrightarrow_H^* \|_{i\in I}\ \langle css''_{2i}, \sigma''_{2i}, \varphi''_{2i}\rangle$ applying Lemma 6.10 give

$$\begin{array}{ll}\sigma''_{1i}\ x|_{Time} = \sigma''_{2i}\ x|_{Time}\ \wedge & (6.10.3)\\ \varphi''_{1i}\ s\ 1|_{Time} = \varphi''_{2i}\ s\ 1|_{Time} & (6.10.4)\end{array}$$

furthermore it follows that $css''_{1i} = css''_{2i}$. Now from Lemma 6.6 we have

$$\sigma''_{1i}\ x|_{Time} = \Gamma\ x\ \wedge\ \varphi''_{1i}\ s\ 1|^{--}_{Time} = \Gamma\ s$$

and because evaluating the configuration with the rule [**Active signals (A)**] give

$$\|_{i\in I}\ \langle i : \texttt{process}\ (clk)\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i, \sigma_i, \varphi_i\rangle \Longrightarrow \|_{i\in I}\ \langle ss'_i, \sigma_i, \varphi'_i\rangle$$

where

$$\begin{array}{lcl}\varphi'_i\ s\ 0 & = & \begin{cases} f_s\{\{(v, t^{--}) \mid \mathcal{U}(\varphi_k, s) = (v, t) \wedge k \in I\}\} & \\ & \text{if } \exists j \in I.\ \varphi_j\ s\ 1 \text{ is defined}\\ (v, t^{--}) & \text{otherwise, } \varphi_i\ s\ 0 = (v, t)\end{cases}\\ \varphi'_i\ s\ 1 & = & undef\\ ss'_i & = & ss_i; i : \texttt{process}\ (clk)\ \texttt{begin}\ ss_i;\ \texttt{end process}\ i\end{array}$$

Now $css'_{1i} = css'_{2i}$, $\sigma'_{1i}\ x|_{Time} = \sigma'_{2i}\ x|_{Time} \in \Gamma\ x$ and $\varphi'_{1i}\ s\ 1|_{Time} = \varphi'_{2i}\ s\ 1|_{Time}$ $= undef$ follow directly from the rule. Finally, $\varphi'_{1i}\ s\ 0|_{Time} = \varphi'_{2i}\ s\ 0|_{Time} \in \Gamma\ s$ follow from the two cases; first when $\exists j \in I.\ \varphi_j\ s\ 1$ is defined we have the result of the analysis $\varphi''_{1i}\ s\ 1|^{--}_{Time} = \varphi''_{2i}\ s\ 1|^{--}_{Time} \in \Gamma\ s$, and since the rule modify the signal stores as $\varphi'_i\ s\ 0 = f_s\{\{(v, t^{--}) \mid \mathcal{U}(\varphi_k, s) = (v, t) \wedge k \in I\}\}$ we get the result for all active signals. The other case is when the signal is not active. From Lemma 6.7 we observe that if a signal is not active in one execution it will never be in any execution that can be typed with the same behaviour, τ. Hence no execution path will modify the signals timing delay, therefore it will be $\emptyset$, and the case follows. □

6.5 Analysing Advanced Encryption Standard

In this Section we briefly summarise the application of our prototype implementation of the analysis to verify the AES standard pipelined implementation [WBRF00]. The 128 bit version of the algorithm works on blocks of 128 bit, that are encrypted through 10 rounds. Each round consists of 4 stages that substitute, shift, mix and add a *round key* to the block, respectively. Decryption is done by reversing the order of the inverse stages in each round. The implementation considered performs either encryption or decryption depending on the value of the incoming signal `ALG_ENC`. The implementation uses the concurrent statement

```
ALG_DATA_LAST <= POST_OUT_INT when ALG_ENC = '0' else FINAL_OUT;
```

to transfer the result of either encryption or decryption from an internal signal to the final register before the outgoing signal is set. Observe that only one of the signals `POST_OUT_INT` and `FINAL_OUT` influences the signal `ALG_DATA_LAST`, depending on the value of the signal `ALG_ENC`. In the case of decryption the algorithm specifies that the key should be added to the final state. For encryption this is done before the first round instead. Figure 6.2 shows the flow of information from the process performing the last round in the AES algorithm to the `ALG_DATA_LAST` signal is set.

The implementation of the analysis starts by pre-processing VHDL programs to CoreVHDL. For the statement considered it is rewritten into a conditional using the rule presented in [Ash02]. It turns out that our analysis does not accept the program because the process `ARRAY_REG128` will spend one clock cycle to stabilise the signal for the adder (`POST_ADD`) that then add the final

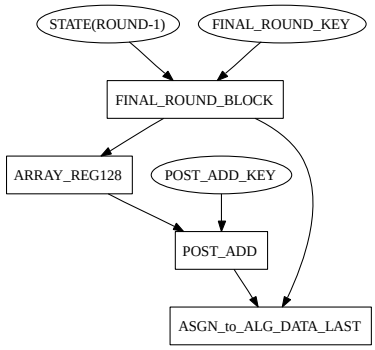


Figure 6.2: Process dependencies of part of AES

key to the block. This difference might cause a timing leak, as it could leak the value of `ALG_ENC`.

Observing the signal `ALG_ENC` tells whether the system is performing encryption or decryption, and hence carry no secret information. Therefore to verify the implementation we added the signal to the threshold of our observer ($\mathcal{L} = \{\texttt{ALG\_ENC}\}$). This allow us to verify that the system is timing secure, as an observer can learn no additional information by observing the timing delays of the outcoming signals.

6.6 Discussion of practical issues

The presented semantics for CoreVHDL extended with timing behaviours capture the delays introduced by executions paths found in the specification. Alternative execution paths have been illustrated to have the capability to result in timing channels, yet a couple of practical issues remain to be investigated. Hence in this section two practical issues that could result in timing channels are discussed. Observe that they have no impact on the analysis of the AES example, and thus our results for the example program remain valid.

6.6.1 Stabilising signals

The VHDL Standard [IEE01] specifies that all signals should carry stable values before the next synchronisation. Therefore for synchronous specifications it is safe to assume that all signals will have a stable value no later than the rising clock edge. But all signals are not assigned a value at exactly the time of the clock, rather all signals are assigned their value in the interval between two clock edges. In our model of systems an observer can measure this interval, and with specific knowledges about the layout at gate level the observer might use the stabilising time to differentiate between secret input values.

Example 6.2 *For a VHDL specification the synthesiser tool will generate the electrical circuit. Consider the program*

$$d <= a \ \texttt{or} \ (b \ \texttt{and} \ c)$$

assume that d is an outgoing signal and that the synthesiser tool would generate a circuit corresponding to the gates illustrated in Figure 6.3(a). For the input

$$[a = 1, b = 0, c = 1]$$

the circuit will stabilise its value at the time t_{or}, since the or *gate stabilises because of the value of a, without waiting for the* and *gate to stabilise. However if the input was*

$$[a = 0, b = 1, c = 1]$$

then the value of d would not stabilise before the time of the or *gate and the* and *gate (i.e. $t_{or} + t_{and}$). The times are illustrated in the graph in Figure 6.3(b), where the time t_{clock} is the clock the system synchronises on.*

To handle this possible difference in stabilising speeds we transform the VHDL specification in such a manner that all secret outgoing signals go through a *buffer*. Technically this is done by rewriting the program so that all assignments to an outgoing signal are altered to assignment to our buffer signal. Then we introduce a process of the form

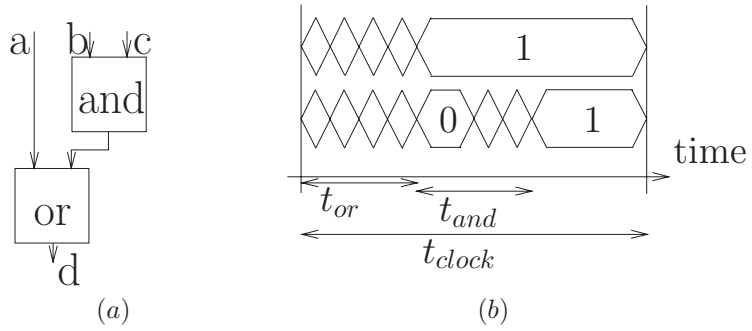


Figure 6.3: Stabilising times for a small circuit; (a) the considered circuit, (b) timing of signals changes

$$buf_s : \texttt{process (}clk\texttt{) begin } s <= s_{buf}\texttt{; end process;}$$

where s is the outgoing signal, s_{buf} is an internal signal specified within the block, and *clk* is the signal carrying the global clock. The *buffer* process will delay the signal one clock cycle, and then uniformly transfer the value from the memory cell to the outgoing wire.

While this buffering method handles the potential differences in time intervals of stabilising signals, it is not sufficient for eliminating timing channels completely. Timing channels caused by different paths through a program are considered in the following Section.

6.6.2 Shortest circuit evaluation

In this Section we introduce an alternative semantics for the logical operators `and` and `or`. This extension is based on the observation that when applying the operators, one side of the expression might be *disregarded* as the value of the other side might dominate the value of evaluating the expression. This observation has previously been exploited in e.g. *short circuit* evaluation of boolean expression in compilers [WM95].

The semantics are extended with a set of specialised rules that apply to the cases

where a shorter circuit is available instead of evaluating the whole expression.

$$\mathcal{E}[\![e_1 \texttt{ or } e_2]\!]\langle\sigma,\varphi\rangle = \begin{cases} ('1', t_1) & \text{where } \mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle = ('1', t_1) \\ ('1', t_2) & \text{where } \mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle = ('1', t_2) \\ (v_1 \mathbin{\overline{\texttt{or}}} v_2, t_1 \cup t_2) & \text{where } \mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle = (v_1, t_1) \\ & \text{and } \mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle = (v_2, t_2) \end{cases}$$

$$\mathcal{E}[\![e_1 \texttt{ and } e_2]\!]\langle\sigma,\varphi\rangle = \begin{cases} ('0', t_1) & \text{where } \mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle = ('0', t_1) \\ ('0', t_2) & \text{where } \mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle = ('0', t_2) \\ (v_1 \mathbin{\overline{\texttt{and}}} v_2, t_1 \cup t_2) & \text{where } \mathcal{E}[\![e_1]\!]\langle\sigma,\varphi\rangle = (v_1, t_1) \\ & \text{and } \mathcal{E}[\![e_2]\!]\langle\sigma,\varphi\rangle = (v_2, t_2) \end{cases}$$

The analysis can be modified to accommodate the alternative semantics. The rules for `and` and `or` can be modified as

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_1 \triangleq t_2}{\Gamma \vdash e_1 \texttt{ and } e_2 : t_1 \cup t_2} \qquad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_1 \triangleq t_2}{\Gamma \vdash e_1 \texttt{ or } e_2 : t_1 \cup t_2}$$

We introduce the binary operator $\triangleq$ for comparing the types of two expression

$$t_1 \triangleq t_2 \Leftrightarrow \{(n, z_1) \mid \exists z_2 : (n, z_1) \in t_1 \wedge (n, z_2) \in t_2\} = \{(n', z_2) \mid \exists z_1 : (n', z_1) \in t_1 \wedge (n', z_2) \in t_2\}$$

which is used to make sure that if the evaluation of an expression might depend on only part of the values manipulated, then the timing delay of each part is the same. As an example for the operators `and` and `or` the evaluation of the first operand might determine the value of the whole expression, thereby rendering the other operand without influence of the result and the timing delay of dependent signals. Notice that the evaluation of `xor` always need to evaluate both operands to determine the resulting value, therefore we don't need to restrict the operands timing delay.

6.7 Summary

A semantics for a synchronous synthesizable subset of VHDL has been presented, with a novel component for describing the timing delay of programs. It was argued that when considering specifications of hardware a new model for the absence of timing leaks is needed. This is needed because for hardware systems an outgoing signal has a value at all times, and an observer therefore might

observe modifications rather than termination or difference in execution time. A model was proposed and formalised with respect to the semantics.

To verify the absence of timing leaks in a specification of a system an automatic type based analysis have been presented. A novel point of the analysis is that it focuses on the problem of timing leaks. This allows us to argue the absence of timing channels in such systems as cryptographic algorithms, even when information flow analyses would not permit it. Furthermore the result of the analysis can be composed with the information flow analyses for CoreVHDL to achieve a timing sensitive bisimulation-based confidentiality property. Finally the analysis was used it to verify the AES standard implementation.

Chapter 7

Conclusion

In the Kamigata area they have a sort of tiered lunchbox
they use for a single day when flower viewing.
Upon returning, they throw them away, trampling them underfoot.
The end is important in all things.
— Yamamoto Tsunetomo

In this thesis the semantical model of VHDL has been investigated and formalised. As a result we have been able to present formal security properties for VHDL programs that guarantee them to preserve the confidentiality of secret information by not allowing channels observable to different principals to interfere. In fact, the investigated security properties generalise noninterference, yet in a controlled fashion, allowing systems, through the execution of specified parts of the program, to dynamically release sensitive information. Our security properties are supported by static analyses, that allow for automatic verification of systems.

Previously the need for static mechanisms has been argued [Vol99], and the analyses proposed here fulfill the requirements stated in Chapter 1:

Automatic: The analyses are specified so that the process of validating a security policy for a given program is fully automated. Furthermore the prototype implementation of the analyses provide automated inference algorithms.

Correctness: The semantic correctness of the analyses, shown in Chapter 4 and 6, demonstrates how the analyses are tied to the semantics of VHDL and the security properties. The extensions presented in Chapter 5 are based on existing analysis approaches and hence the correctness is believed to be adaptable from the similar results presented in [NNH99].

Efficient: The analyses for verifying the flow of information are divided into several components, and the correctness results are stated such that the local dependencies are sufficient for verifying most systems. Hence the global dependency analysis is a means to trade in efficiency for precision.

Exhaustiveness: By their nature the analyses clearly provide an exhaustive approach to verifying programs, as we do not rely on the extraction of a model of the essential part of the program, program transformations, or similar abstractions. Instead the analyses are applied directly to the verification target.

Another main achievement is the verification of the reference implementation of the Advanced Encryption Standard. This allows us to conclude that the primary goal of this thesis, namely to accommodate the verification of hardware specifications, has been achieved. The prototype implementation has also been successful in analysing the systems provided by Maersk Data Defence. Although the full Common Criteria report considers aspects of the system that has not been discussed in this thesis, the objectives regarding the identification of information flow and covert channels stated in the Chapter 13.1 Objective: *Covert Channel Analysis (AVA_CCA)*, have been achieved.

7.1 Final remarks

Throughout the thesis we have investigated the reference implementation of the Advanced Encryption Standard. Verifying a cryptographic algorithm in the information flow setting is not straightforward, and we have assumed that the inreversibility of the cipher text is indeed provided by the implementation. History-based Release has merely allowed us to guarantee that all rounds are applied, and that each round gradually lowers the confidentiality level of texts.

The correctness of the implementation has previously been argued by the developers in [WBRF00] and related publications. Their correctness result relies on mathematical observations and testing through a test-bench, following the *Monte Carlo* approach where the output is fed back into the program for the next round. This approach clearly does not provide an exhaustive correctness result.

Other approaches include the correctness guarantees of the algorithm in the information flow analysis. Laud [Lau03] presents an information flow analysis that considers the computations and by doing this can verify systems that protect the secrecy of the input with a given complexity of the reversibility of the outputs. Recently, Askarov et al. [AS05, AHS06] have investigated the information flow in systems where cryptographic functions are available. However the cryptographic functions are considered as black boxes that correctly implement the cryptographic algorithms and hence provide the necessary guarantees.

The issue of timing channels have been investigated by several researchers [VS99, Aga00, SS00, Sab01, KM05, KB06], yet the security properties proposed are all strong noninterference properties. In this thesis we have specialised the absence of timing leaks to deal with timing channels. Hence the property becomes orthogonal to the generalised noninterference properties that we might choose to use for verifying the flow of information through the system. This flexibility also allows us to specify that information is allowed to flow on a channel, yet not being observable by observations on the timing behaviour of this channel. This is of utmost importance when considering cryptographic systems, as timing channels have often been exploited in order to reduce the search space of secret key or plain text attacks.

The analyses presented throughout this thesis have been implemented using the Succinct Solver v. 1.0 [NNS02, NNS$^+$04], which has made it possible to verify the AES implementation and the Maersk Data Defence program. The prototype implementation has shown very good running times on the discussed programs, taking only a few minutes. However, for the analysis of large-scale, complex systems Mantel [Man01, Man02] have argued in favour of compositional properties and analyses. In this thesis we have taken an approach that allows for a great deal of compositionallity. First, the local dependencies are analysed at process level, without regard for the rest of the system. Second, the global dependencies are derived from the local dependencies and strengthened by the Reaching Definitions analysis. This means that during a development process where the system goes through several stages, the analysis result for each unmodified process can be reused when verifying the complete system.

Future investigations might consider the possibility of deciding the need for applying the Reaching Definitions analysis automatically, by a pre-analysis of the security policy. This would also allow for a fine-grained trade off between precision and efficiency, where necessary.

Bibliography

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *Proc. Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2004.

[AF03] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2003.

[Aga00] Johan Agat. Transforming out timing leaks. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, January 2000.

[AHS06] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. In *Proc. Symposium on Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 353–369. Springer-Verlag, 2006.

[AS05] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 197–221. Springer-Verlag, 2005.

[Ash02] Peter J. Ashenden. *The Designer's Guide To VHDL*. Morgan Kaufmann, 2nd edition, 2002.

[BBN+03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario

Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In *Global Computing*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer-Verlag, 2003.

[BDF05] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *Proc. Foundations of Software Science and Computation Structure*, volume 3441 of *Lecture Notes in Computer Science*, pages 316–332. Springer-Verlag, 2005.

[BFK94] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. Clean formal semantics for vhdl. In *EDAC-ETC-EUROASIC*, pages 641–647, 1994.

[BL73] David E. Bell and Len J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[BLPV94] Jörg Bormann, Jörg Lohse, Michael Payer, and Gerd Venzl. Vhdl translation for bdd-based formal verification. Technical report, Siemens, 1994.

[BN04] Anindya Banerjee and David A. Naumann. History-based access control and secure information flow. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48. Springer-Verlag, March 2004.

[BS06] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag, 2006.

[CC98] International Standards Organisation. *Common Criteria for information technology security (CC), ISO/IS 15408 Final Committee Draft, version 2.0.*, 1998.

[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 *Proc. Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, 1999.

[CHH02] David Clark, Chris Hankin, and Sebastian Hunt. Information flow for algol-like languages. *Comput. Lang.*, 28(1):3–28, 2002.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[CM05] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, pages 241–254. IEEE Computer Society Press, 2005.

[Coh77] Ellis S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[Coh78] Ellis S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[DB93] David Déharbe and Dominique Borrione. A qualitative finite subset of vhdl and semantics. Technical Report RR943, IMAG Institute, 1993.

[DB95] David Déharbe and Dominique Borrione. Semantics of a verification-oriented subset of vhdl. In *Proc. Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 293–310. Springer-Verlag, 1995.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[DKL+98] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A Practical Implementation of the Timing Attack. In *Proc. of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 1998.

[DOD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, dod 5200.28-std (the orange book) edition, December 1985.

[Est05] Esterel Technologies. *The Esterel v7 Reference Manual*, version v7_30 - initial IEEE standardization proposal edition, 2005.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[GM84] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–87. IEEE Computer Society Press, 1984.

[Goo95] Kees G. W. Goossens. Reasoning About VHDL Using Operational and Observational Semantics. In *Proc. Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 311–327. Springer-Verlag, 1995.

[HKMY86] J. Thomas Haigh, Richard A. Kemmerer, John McHugh, and William D. Young. An experience using two covert channel analysis techniques on a real system design. In *Proc. IEEE Symposium on Security and Privacy*, pages 14–24. IEEE Computer Society Press, 1986.

[HL98] Yee-Wing Hsieh and Steven P. Levitan. Control / data-flow analysis for vhdl semantic extraction. *Journal of Information Science and Engineering*, 14(3):547–565, 1998.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition.* Addison-Wesley, 2001.

[HR98] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, 1998.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proc. ACM Symposium on Principles of Programming Languages*. ACM Press, January 2006.

[HY86] J. Thomas Haigh and William D. Young. Extending the Non-Interference Version of MLS for SAT. In *Proc. IEEE Symposium on Security and Privacy*, pages 232–239. IEEE Computer Society Press, 1986.

[Hym02] Charles Hymans. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. In *Proc. Symposium on Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 444–460. Springer-Verlag, 2002.

[IEC05] IEC and IEEE inc. *International Standard VHDL Register Transfer Level (RTL) synthesis, IEC 62050, IEEE Std 1076.6-2004*, 2005. Revision of IEEE Std 1076.6-1999.

[IEE87] IEEE inc. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987*, 1987.

[IEE93a] IEEE inc. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164), IEEE Std 1164-1993*, 1993.

[IEE93b] IEEE inc. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993*, 1993. Revision of IEEE Std 1076-1987 [IEE87].

[IEE95] IEEE inc. *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-1995*, 1995.

[IEE01] IEEE inc. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-2001*, 2001. Revision of IEEE Std 1076-1993 [IEE93b].

[IEE05] IEEE inc. *IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2005*, 2005.

[ITS91] Commission of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC): Preliminary Harmonised Criteria*, document COM(90) 314, Version 1.2 edition, June 1991.

[KB06] Boris Köpf and David Basin. Timing-sensitive information flow analysis for synchronous systems. In *Proc. European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 243–262. Springer-Verlag, 2006.

[Kem82] Richard A. Kemmerer. A practical approach to identifying storage and timing channels. In *Proc. IEEE Symposium on Security and Privacy*, pages 66–73. IEEE Computer Society Press, 1982.

[Kem02] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proc. IEEE Annual Computer Security Applications Conference*, pages 109–118. IEEE Computer Society Press, 2002.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proc. CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[KM05] Boris Köpf and Heiko Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Proc. Workshop on Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, July 2005.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[KT96] Richard A. Kemmerer and Tad Taylor. A modular covert channel analysis methodology for trusted dg/ux/sup tm/. In *Proc. IEEE Annual Computer Security Applications Conference*, pages 224–235. IEEE Computer Society Press, 1996.

[KWMK01] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Fault-based side-channel cryptanalysis tolerant rijndael symmetric block cipher architecture. In *Proc. Defect and Fault-Tolerance in VLSI Systems*, pages 427–435. IEEE Computer Society Press, 2001.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, 1973.

[Lau03] Peeter Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, 2003.

[Man01] Heiko Mantel. Information flow control and applications - bridging a gap. In *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 153–172. Springer-Verlag, March 2001.

[Man02] Heiko Mantel. On the composition of secure systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 88–101. IEEE Computer Society Press, May 2002.

[Mat04] Dan Matthews. Hardware Bus Security in Embedded Systems. The Fifth HOPE. 2600, 2004.

[MB05] Ana Almeida Matos and Gérard Boudol. On declasification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240. IEEE Computer Society Press, June 2005.

[McH83] John McHugh. *Towards Efficient Code from Verified Programs.* PhD thesis, The University of Texas at Austin, 1983.

[McH95] John McHugh. Covert Channel Analysis. *Handbook for the Computer Security Certification of Trusted Systems*, 1995.

[MED02] MEDEA+. *Design Automation Solutions for Europe*, 2002.

[MED04] MEDEA+. *The Future of the European Microelectronics Industry*, 2004.

[Men03] Mentor Graphics. *ModelSim SE 5.7d VHDL simulator*, 2003. http://www.model.com/.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. ACM Symposium on Operating System Principles*, pages 129–142. ACM Press, 1997.

[MS04] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proc. Asian Symposium on Programming Languages and Systems*, volume 2244 of *Lecture Notes in Computer Science*, pages 129–145. Springer-Verlag, 2004.

[Mye99a] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 228–241. ACM Press, 1999.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1999.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley & Sons, 1992.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NNS02] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

[NNS+04] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Michael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The Succinct Solver Suite. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 251–265. Springer-Verlag, 2004.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60–61:17–139, 2004.

[QS01] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards. In *Proc. of E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2001.

[RBG00] Vanderlei Moraes Rodrigues, Dominique Borrione, and Philippe Georgelin. Using the acl2 theorem prover to reason about vhdl components. *RITA*, 7(1):129–148, 2000.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238. IEEE Computer Society Press, 1999.

[RMMG01] Peter Ryan, John D. McLean, Jonathan K. Millen, and Virgil D. Gligor. Non-interference: Who needs it? In *Proc. IEEE Computer Security Foundations Workshop*, pages 237–238. IEEE Computer Society Press, 2001.

[Rus92] John M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, December 1992.

[Rus95] David M. Russinoff. A formalization of a subset of vhdl in the boyer-moore logic. *Formal Methods in System Design*, 7(1/2):7–25, 1995.

[Sab01] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, 2001.

[SEM03] International SEMATECH. *International Technology Roadmap for Semiconductors*. 2003.

[SH89] A. Salz and Mark Horowitz. IRSIM: An Incremental MOS Switch-Level Simulator. In *Proc. 26th Design Automation Conference*, June 1989.

[SM03a] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM03b] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *ISSS*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, 2003.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[SS01] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, 2001.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005.

[Sut86] D. Sutherland. A model of information. In *Proc. National Computer Security Conference*, pages 175–183, September 1986.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 355–364. ACM Press, 1998.

[SV00] Daryl Stewart and Myra VanInwegen. Three notes on the interpretation of verilog. Technical Report 485, University of Cambridge, 2000.

[TE01] Krishnaprasad Thirunarayan and Robert L. Ewing. Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93. *Formal Methods in System Design*, 18(1):69–88, 2001.

[Thi03] Personal communication with Krishnaprasad Thirunarayan, August 2003.

[TN06] Terkel K. Tolstrup and Flemming Nielson. Analyzing for absence of timing leaks in VHDL. In *Proc. Sixth International Workshop on Issues in the Theory of Security*, March 2006.

[TNH06] Terkel K. Tolstrup, Flemming Nielson, and René Rydhof Hansen. Locality-based security policies. In *Proc. Workshop on Formal Aspects in Security and Trust*, Lecture Notes in Computer Science. Springer-Verlag, August 2006.

[TNN05] Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson. Information Flow Analysis for VHDL. In *Proc. Eighth International Conference on Parallel Computing Technologies*, volume 3606 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2005.

[Tui92] Paul W. Tuinenga. *SPICE: A Guide to Circuit Simulation and Analysis Using PSpice*. Prentice Hall, 1992.

[Vol99] Dennis M. Volpano. Safety versus secrecy. In *Proc. Symposium on Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer-Verlag, 1999.

[VS99] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.

[VSI93] The Keystone Research Group. *VCOMP & VSIM*, 1993.

[VSI96] Dennis M. Volpano, Geoffrey Smith, and Cynthia E. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[vT93] John P. van Tassel. *FEMTO-VHDL: The semantics of a subset of VHDL and its embedding in the hol proof assistant.* PhD thesis, University of Cambridge, 1993.

[WBRF00] Bryan Weeks, Mark Bean, Tom Rozylowicz, and Chris Ficke. Hardware performance simulations of round 2 advanced encryption standard algorithms. Technical report, National Security Agency, 2000.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison Wesley Higher Education, 1995.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.