FORMAL SEMANTICS AND PROOF TECHNIQUES FOR OPTIMIZING VHDL MODELS

Kothanda Umamageswaran
Sheetanshu L. Pandey
Philip A. Wilsey

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

FORMAL SEMANTICS AND PROOF TECHNIQUES FOR OPTIMIZING VHDL MODELS

FORMAL SEMANTICS AND PROOF TECHNIQUES FOR OPTIMIZING VHDL MODELS

Kothanda Umamageswaran

Sheetanshu L. Pandey

Philip A. Wilsey

University of Cincinnati

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-7331-5 ISBN 978-1-4615-5123-2 (eBook)
DOI 10.1007/978-1-4615-5123-2

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Printed on acid-free paper.

Contents

List of Figures

List of Tables

Preface

VHDL is a hardware description language that has been formally standardized by the IEEE as IEEE–Std–1076. By the rules of the IEEE, the VHDL language standard must be reviewed and re-balloted every 5 years. It was originally balloted in 1987, again in 1993, and is currently going through re-balloting this year (1998). The VHDL standard is monitored, developed, and interpreted by the VASG (VHDL Analysis and Standards Group).

VHDL is widely used in the design of electronic systems and a wide variety of commercial tools exist to manipulate designs captured in VHDL. These tools range from discrete event simulators and waveform viewing environments to synthesis programs and equivalence checkers. However, the semantics of VHDL is provided informally using English prose and examples. Consequently, there remain several aspects of the language that are incompletely specified or have been subject to multiple interpretations. To address these issues, the VASG has setup a subcommittee called ISAC (Issues Screening and Analysis Committee). ISAC then addresses questions of language interpretation and reports its findings to the VSAG for review and approval. ISAC receives IRs (Issue Reports) from the community (and itself) that highlight an issue of language interpretation. The IR identifies the problem (frequently including a proposed resolution) and requests that ISAC (through the VSAG) provide a ruling on the correct interpretation. As these issues are resolved, they generally are incorporated into the formal language standard in the next round of balloting. Despite considerable effort with the 1987 standard, the 1993 language still had approximately 30 IRs issued for resolution.

This book presents a formal model of VHDL that clearly specifies both the static and dynamic semantics of VHDL. It provides a mathematical framework for representing VHDL constructs and shows how those constructs can be formally manipulated to reason about VHDL. The dynamic semantics is presented as a description of what the simulation of VHDL means. In particular it specifies what values the signals of a VHDL description will take if the description were to be executed. An advantage of the approach is that the semantic model

can be used to validate different simulation algorithms. The book also presents an embedding of the dynamic semantics in a proof checker which is then used to prove equivalences of classes of VHDL descriptions.

We recommend this book for readers interested in formal semantics of VHDL. The book assumes that the reader has a fair knowledge of VHDL. We would be pleased to receive comments and corrections from the users of this book.

AUTHORS

To my parents....

— K. Umamageswaran

To my parents and beloved
sisters whose love and affection
I cannot do without.

— S. L. Pandey

To Marilyn....Thank you for
joining me in the battle of life.

— P. A. Wilsey

Acknowledgments

This research has been conducted with the participation of many investigators. In particular, we would like to acknowledge the efforts of the following individuals (in alphabetical order): David Benz, Xianzhi Fan, Tim McBrayer, Dale Martin, Magesh Narayanan, and David Sims. Furthermore, we have benefited greatly from our regular interactions with colleagues locally at UC and "out there" on the internet. While not a complete list, the following individuals have made notable direct and/or indirect contributions to this effort (in alphabetical order): Perry Alexander, Peter Ashenden, David Barton, Harold Carter, David Goldschlag, Kees Goossens, Dr. John Hines, J.P. Letellier, Capt. Brad Mallare, Paul Menchini, Capt. Greg Peterson, and Mark Richards.

We would also like to thank Carl Harris for carefully reviewing this work and providing useful suggestions.

This research was supported in part by the Defense Advanced Research Projects Agency and monitored by the Air Force Wright Laboratory under contract number F33615–93–C–1315. In addition, we benefited greatly from the technical support and guidance by the Wright Laboratory program officers, notably Capt. Brad Mallare, Dr. Robert Ewing, and Dr. John Hines.

Foreword

It has been quite a while since the emergence of the original IEEE standard for VHDL. In that time, many efforts have been made to apply formal methods to the language. They have all, in general, been associated with one subset or another. Many of those subsets were chosen in order explore different aspects of VHDL's semantics. Some have proved commercially successful (i.e. formal tools associated with the RTL synthesis semantic currently employed in mainstream digital design). Others, have been useful vehicles for furthering the community's general understanding of the language. All represent *tour de force* work in the application of formal semantic techniques to a modern, mainstream design language.

The work on the pages which follow expands upon what has come before to provide a more complete formalization of the semantics of VHDL. At the same time, it makes use of the mathematics in a different context than previous work. In earlier activities, the objective was to prove properties of VHDL designs from the program artefacts *in situ*. The material in this book employs the semantics of the language to transform those same program texts into more simple forms in order to enable efficient simulation. Along the way, proof obligations are defined for discharge by an automated theorem-prover to ensure that the transformations are being correctly performed. All this, coupled with the fact that the semantics was also used to prove the correctness of the underlying simulation algorithm, represents a set of truly important results.

As the design of digital systems becomes increasingly complex (it is already beginning to take on significant aspects of traditional system design), it is imperative that formal techiques are used to augment familiar, simulation-based analysis approaches. VHDL will play a significant role in those design practices. To enable that future, a semantic which transcends our current, RTL-based one to encompass more behavioral and algorithmic abstractions is required. The work presented here is a huge step in the right direction, and one is deeply flattered to have been asked to introduce it.

John Van Tassel
Dallas, TX

1 INTRODUCTION

Hardware Description Languages (HDLs) are used extensively to specify, verify, and document hardware designs. Typically, an HDL specification is compiled, elaborated, and then simulated. The hardware description language VHDL is one of the more widely supported language for specifying hardware designs. Academic and industrial researchers, along with CAD vendors have provided extensive tool support for VHDL. The official specification of the syntax and the semantics of VHDL is the Language Reference Manual (LRM) [27] which describes the language informally in English prose form. Unfortunately, it is impossible to formally validate simulation results or optimizations to the simulation process using this informal specification. Furthermore, a formal specification on paper is not of much use, since it is not possible to determine the accuracy of the results proven using the semantics. A formal semantics of the language must be developed and embedded in an automated theorem proving environment in order to validate optimizations.

Most current investigations into the formal semantics of VHDL rely on a denotational, axiomatic, or operational definition of a simulation cycle as defined in the LRM [27] to prove correctness of VHDL programs. The goal of these proofs is typically to show the equivalence of a behavioral level specification, and implementation level specification of a circuit. One disadvantage of these approaches is that they bind the meaning of a VHDL description to one particular method of executing its simulation. Alternative mechanisms of simulating VHDL cannot be validated using the above approaches. Further,

the verification methods are applied to specific design instances and cannot be used to prove properties about a set of VHDL descriptions that have a syntactic pattern.

The formal techniques presented in this book to define and reason about VHDL descriptions attempt to address the above concerns. In particular, the book presents a formalization of the static and dynamic semantics of VHDL and a framework for proving equivalences of *classes* of VHDL descriptions. The definition of the static semantics of VHDL is based on a mathematical representation of the static constructs of VHDL such as signals, variables, and concurrent statements. This mathematical representation is called the Static Model. The static semantics appears as a set of well-formedness rules stated in first order predicate logic. A reduction algebra is defined that reduces the Static Model constructs to a minimal, *normal* form. This normal form is used to define a model (called the Dynamic Model) of the dynamic semantics of VHDL. The semantics is presented in a declarative style using Interval Temporal Logic [3, 4, 5] and is independent of the LRM simulation cycle.

Further, the dynamic semantics of VHDL are embedded in Prototype Verification System (PVS) [44, 33] and a framework has been developed for proving equivalences of *classes* of VHDL descriptions. More precisely, this framework accepts specifications of VHDL patterns and proves their equivalence; equivalence proofs of specific instances of the patterns naturally follow. Accordingly, optimizations that transform descriptions from one form to another can be formally verified. The framework consists of the following:

1. A translator that accepts two VHDL descriptions and a set of user-specified signals, and generates a theorem in PVS [39, 45, 40, 15].

2. A static semantics embedded in PVS that facilitates the representation of VHDL constructs.

3. A dynamic semantics embedded in PVS that denotes *what* happens when a VHDL program is simulated.

The translator (SCRAM) [41, 52] consists of a front end parser that checks for syntactic well-formedness of VHDL descriptions. The result of successful parsing is a representation for the input VHDL program in the AIRE/CE Intermediate Format (IF) [41, 52]. From the AIRE/CE representation for the two descriptions and an indication of the set of signals to watch, the PVS code generator outputs a theorem in PVS. When completed, the proof of the theorem is sufficient to show that the two input descriptions are equivalent with respect to the signals that are being watched. The proof is carried out in PVS using (i) the embedding of the static and the dynamic semantics and (ii) all the results that have been previously proven correct. The advantage of this approach is that a theorem when proven can be re-used if/when necessary; this facilitates modular/hierarchical proofs — a capability that is useful for proving equivalences of VHDL descriptions.

1.1 GOALS OF THE WORK

The work presented in this book is intended to be a comprehensive characterization of the semantics of VHDL to validate optimizations of CAD tools for VHDL. The main motivations behind the work are:

- to define a formal semantic model that characterizes a considerable subset of VHDL including key aspects such as inertial and transport delays, sequential and concurrent behavior, pulse rejection limits, zero delayed assignments, and resolution functions.
- to demonstrate the utility of the model in CAD tool optimization by validating transformations on VHDL descriptions and also by deriving conditions necessary to eliminate constructs such as signals and drivers from the simulation of a circuit.
- to define a semantic model that can serve as a basis for validating various methods of simulation other than the *simulation cycle approach* specified by the VHDL LRM.
- to validate semantics preserving transformations on classes of VHDL descriptions, and to obtain the necessary and sufficient preconditions during the process of validation.
- to check the equivalence of a behavioral level specification and an implementation level specification.
- to make the application of the formal semantics transparent; this transparency allows the user to concentrate on writing VHDL descriptions and proving their equivalence, thus saving the drudgery of translating VHDL specifications into the formal framework.

1.2 SCOPE OF THE WORK

The semantics presented in this work is a characterization of the elaborated form of VHDL '93 as defined in the language standard, IEEE Std 1076-1993 [27]. Although the semantics does not cover all the features of VHDL, it is self-contained and characterizes most of the important features.

The Static Model characterizes important VHDL constructs such as concurrent statements, sequential statements, data types, expressions, subprograms, signals, variables, and ports.

The Dynamic Model defines the dynamic semantics of VHDL and uses Interval Temporal Logic to capture the timing information in a VHDL description. Most of the dynamic concepts of VHDL are captured in the Dynamic Model. In particular,

1. *Signal assignment* statements with inertial and transport delay mechanisms are characterized. Transport delayed assignments are defined in terms of inertially delayed assignments and only the latter are then characterized. Signal assignments with zero delay are modeled.

2. The model includes semantics for signals declared with kind `bus` or `register` and signals declared without any kind (referred to as `discrete` signals in this work). Issues related to guarded assignments are also addressed.

3. New features introduced in VHDL '93 such as postponed processes and pulse rejection limits are characterized.

4. Propagation of signal values up and down hierarchies and issues related to driving values and effective values of resolved and unresolved signals are defined.

5. Ports of mode `in, inout, out, buffer,` and `linkage` are characterized.

6. Semantics of all sequential statements with the exception of *procedure calls* and *return* statements are defined.

Some of the features of VHDL are not characterized in this work. In particular,

1. Issues related to configurations, *generate* statements and libraries are not addressed (these are pre-elaboration constructs)

2. Issues related to *procedure* and *function* calls such as passing parameters by value and reference, dynamic stack allocation, scope and visibility of variables are not addressed.

3. Arrays, record types, and pointers are not characterized.

4. File I/O is not considered.

1.3 NOTATION

This document uses the following notation for presenting elements of the static and dynamic models.

1. Terminal items are set in a `typewriter` font and non-terminals are set in a Sans-Serif font.

2. Sets are set off in curly brackets. For example the set of positive integers would be denoted as shown below:

$$\text{positive-integers} \quad = \quad \{ x : x \in \mathsf{N} \wedge x \geq 1 \}.$$

3. Tuples are set off in angled brackets. For example, a 3-tuple for the xyz-plane is written as shown below:

$$\text{xyz-plane} = \langle\, x, y, z \,\rangle.$$

4. Sequences are placed in parenthesis. For example, a sequence of adjacent time intervals is defined by the equation

$$\text{time-keeper} = (\, TK(i) : i \geq 0 \wedge \text{MEETS}(TK(i), TK(i+1)) \,).$$

5. Selection of one element from a list is denoted by enclosing the list in square brackets and separating choices with a vertical bar. For example

$$\text{postponement} = [\, \texttt{postponed} \mid \texttt{not-postponed} \,].$$

6. An axiom of the form $\forall\, x\colon Q(x) \bullet P(x)$ must be read as: For all x such that Q(x) is satisfied, P(x) holds. Sometimes, for the sake of brevity, an axiom of the form $\forall\, x\colon x \in A,\ P(x)$ will be written as $\forall\, x \in A \bullet P(x)$. If no pre-conditions exist, then the axiom $\forall\, x\colon P(x)$ must be read as: For all x, P(x) holds.

7. References to the IEEE Standard VHDL Language Reference Manual (Std 1076-1993) will be of the form (§s, ¶p, Ll) where, s in the section number, p the page number and l the line number. Sometimes, where appropriate, the line number will be omitted.

1.4 OVERVIEW OF BOOK

The remainder of this book is organized as follows. Chapter 2 presents some background information on the area of formal semantics in general and the specific applications of formal semantics to characterize VHDL. Techniques used for specification such as Boyer-Moore Logic, Higher Order Logic, Axiomatic Semantics, and Denotational Semantics are discussed. Chapter 3 presents the mathematical representation of the static constructs of VHDL such as signals, ports, and port associations. Chapter 4 discusses several conditions that need to be satisfied for a VHDL description to be well-formed. Chapter 5 details the algebra that translates the Static Model into a compact reduced form. Chapter 6 proves that the reduced form is complete. Chapter 7 presents the rudiments of interval temporal logic which is the central theory on which the Dynamic Model is based. Chapter 8 develops the notion of a time keeper against which the dynamic semantics of VHDL is defined in later chapters. It also develops the formal semantics of VHDL sequential statements and the process statement. Further, it presents the formalization of transaction list maintenance for updating signal values and characterizes the driving and effective values of signals over a set of time intervals. Chapter 9 presents discussions on the utility of the model developed in Chapter 8. Chapter 10 provides a framework for transforming VHDL descriptions into PVS, and proving properties about classes of VHDL descriptions. Finally, Chapter 11 contains some concluding remarks.

2 RELATED WORK

The semantics of a programming language describes a relationship between the syntax and the model of computation. It is concerned with the understanding or interpretation of programs written in the language and prediction of the output of such programs. Early efforts in formalizing the syntax date back to 1963 when Naur [34] defined the syntax of Algol 60 using the Backus Naur Form (BNF). Since then, context-free grammars that are similar to BNF have been used extensively to specify the syntax of programming languages. The formal specification of the semantics of programming languages has also been keenly pursued. While academic researchers have shown considerable interest and significant contributions have been made, no well-accepted widely-used technique for semantic specification has emerged. There are several widely used techniques for specifying the formal semantics of programming languages. Some of them are listed below.

Denotational Semantics : describing what is computed by a program in terms of mathematical functions for the program or the sub-elements of the program [17, 6, 32].

Algebraic Semantics : describing the meaning of a program using an algebra. The algebraic relationships are expressed using axioms and equations [42].

Axiomatic Semantics : defining the meaning of the program using a set of axioms [7]. It provides a set of assertions about the pre-conditions and the post-conditions that need to hold true. Properties of programs may be inferred from the set of axioms provided.

Operational Semantics : specifying how the state of an abstract computer would change while executing the program [49, 51].

Most of the above techniques have been used in the specification of the semantics of VHDL. Other techniques such as *Evolving Algebras* [8], *Petri Nets* [16, 35, 38, 37], and *Higher Order Logic* [22] have also been used and are discussed in detail in the following sections. In general, the formal semantics of a language has been used to clarify the informal semantics. In the case of VHDL, it is also used for providing a basis for verifying the correctness of hardware designs and to directly reason about the descriptions.

2.1 HIGHER ORDER LOGIC

Higher order logic is an extension of first order logic in which variables can range over functions and predicates [12]. A function is first order if none of its arguments is a function, *i.e.,* all of its arguments are data. In higher order logic, functions may be arguments to other functions, and functions may be returned as a result of computing the value of expressions. For example, consider the function *map*:

```
map(f: [item -> item], l: list(item)): list(item) =
      if l == null
        null
      else
        f(head(l)) . map(f, tail(l))
      fi
```

The function *map* is a higher order function that takes as arguments, a function and a list, and returns a list by applying the function to every element in the list. Applications that could use the function *map* include, but are not limited to, the conversion of all the strings in a list into uppercase (or lowercase) and to double every element present in a list. In the former case, the function *f*, converts a given string into the corresponding string in uppercase (or lowercase) and the list `l` contains the list of strings that need to be converted. The function *map* provides an example of variables ranging over functions. A classic example of the case when a variable ranges over a predicate is the definition of induction

for most types. In the case of natural numbers, induction is defined as

$$\forall P : pred[nat] \quad (P(0) \wedge (\forall n : nat \quad P(n) \Rightarrow P(n+1))) \Rightarrow (\forall k : nat \quad P(k)).$$

In the above example P is a variable that ranges over the predicate for natural numbers. A popular proof assistant that supports Higher Order Logic is HOL [23].

Specifications of digital circuits directly in HOL has been pursued by Gordon [22]. He formally describes circuits using pure logic. The inference mechanisms present in the logic form the necessary tools to prove properties about the circuits. Gordon models circuits as a conjunction of predicates where each predicate represents a component in the circuit. Proofs of equivalence of implementations of generic n-bit adders, sequential multipliers, and flip-flops have been demonstrated. Although, many useful results can be obtained, this method is not widely adopted because errors frequently arise in formally specifying the behavior of the circuit in pure logic.

Van Tassel [49, 50, 51] provides an operational semantics for a subset of VHDL '87 (Femto-VHDL) and embeds the semantics in the HOL proof assistant. The semantics covers delta delays, inertial and transport delays, zero delay assignments, `if-then-else` statements, and signal assignment statements. However, the semantics assumes that all the signals are boolean valued, and that there are no resolved signals. In contrast to other efforts which provide a *shallow embedding* of VHDL [14, 8, 36], Van Tassel's effort provides a *deep embedding* of the language structure [12]. More precisely, the semantics includes the necessary and sufficient conditions for a Femto-VHDL program to be syntactically valid. This syntactic well-formedness is captured as a set of rules that check for multiple drivers on a signal, valid declarations for all signals appearing in the program text and valid directionality for all signal assignments (*i.e.,* signals of direction `in` must not be driven, and signals of direction `out` must not be read). The dynamic semantics is presented as a formalization of the *simulation cycle* presented in the VHDL LRM [26]. Using the static and dynamic semantics, and the embedding in the HOL proof assistant, proofs of equivalence of behavior and implementation of simple circuits such as NAND gates, flip-flops, and parity-checkers are demonstrated. The primary limitation of this approach is the close relationship of the semantic model with the simulation cycle which complicates investigations into different methods of simulating VHDL (*e.g.,* parallel simulation).

LAMBDA is a synthesis tool from Abstract Hardware Limited that automatically applies formal verification during the design process. It uses higher order logic to formally represent and manipulate behavioral descriptions. The tool set consists of (i) DIALOG, a design environment, (ii) ANIMATOR, a behavioral simulator, and (iii) BROWSER, an interface to the *theorem prover core*, which can be used effectively for proving properties of specifications and

for the development of new proofs, extending the design automation capability. The advantage of using LAMBDA is that errors are detected and corrected at an early stage in the design cycle and the circuits generated are *correct by construction.*

2.2 DENOTATIONAL SEMANTICS

Denotational semantics has traditionally been described as the theory of true meanings for programs, or, as the theory of *what* programs denote [19]. Denotations tell what is computed by giving a mathematical object (typically a function) which is the meaning of the program. A denotational definition of the semantics of a language typically consists of three parts: the abstract syntax of the language, a semantic algebra defining a computational model, and valuation functions. The valuation functions map the syntactic constructs of the language to the semantic algebra. In the case of VHDL, most of the work done in this area has been towards describing the semantics of the simulation by representing the simulation kernel as a function.

Davis [17] provides a denotational specification of the simulation cycle for a limited subset of VHDL '87. A characterization of the VHDL simulation kernel is provided in terms of semantic algebras and valuation functions that map states to consecutive states. Transport delays, inertial delays, and propagation of signal values are characterized, but guarded signal assignments and communication between architectures are not.

Barton [6] also provides a functional characterization of several of the language features but does not attempt to give a complete definition. Characterizations of drivers, delayed signal assignments, resolution and type conversion functions are provided. No proof methods are investigated.

Müeller [32] defines a static semantics of VHDL using a denotational approach and a dynamic semantics using an operational approach. The abstract syntax of the language is a mapping of the static constructs of VHDL onto the formal framework. The dynamic semantics is based on an operational model using Interval Event Structures. The subset handles global and local variable assignments, transport and inertial delays, delta delays, and postponed processes. The semantics, however, restricts itself to a single architectural specification and a set of explicit process statements. More precisely, communication between architectures is ignored.

2.3 FUNCTIONAL SEMANTICS

Functional Semantics as defined by Breuer *et al* [14] is one that is declarative in style and concept but can also be executed. More precisely, it is a semantics which (i) when executed provides results that resemble the execution of a program in that language (when compiled and simulated/executed using other compilers), and (ii) when read, resembles a high-level non-operational semantics (declarative semantics).

Breuer *et al* [14] define a functional semantics for VHDL descriptions that contain unit delays only. The semantics covers signal assignment statements and wait statements. Delta-delayed signal signal assignments, variables, resolution functions, pulse-rejections limits, postponed processes, and other features included in VHDL '93 have not been characterized. A VHDL sequential statement is modeled as a function that takes as argument the current set of drivers and provides a pair that describes the new set of drivers and the sequence of states that existed during the time that the statement was blocked (called an *Episode*). Stated formally,

$$Sequential\_Statement\_Semantics : \\ DriverSet \rightarrow (Episode \times DriverSet).$$

A VHDL process statement is modeled as a function that maps from an initial driver set to an *Episode*. In general, the resulting *Episode* is infinite. It is the result of the execution from a view point that is in the far-future. Stated formally,

$$Process\_Statement\_Semantics : \quad DriverSet \rightarrow Episode.$$

2.4 AXIOMATIC SEMANTICS

Axiomatic semantics defines the meaning of the program implicitly by making assertions about relationships that hold at each point in the execution of the program [7]. Axioms define the properties of the control structures and state the properties that may be inferred. A property about a program is deduced by using these axioms. Each program has a pre-condition which describes the initial conditions required by the program prior to execution and a post-condition which describes, upon termination of the program, the desired program property.

Bickford and Jamsek [7] define an axiomatic semantics of VHDL using the Larch Prover [21]. The mathematical conditions to be proven are specified in the Larch Shared Language [24] and the connection between these conditions and the VHDL entities is provided by special interface specifications. Using the specification of the architecture corresponding to the entity in VHDL and the interface specifications, they generate a set of conditions that need to be verified. These conditions are then proven using the Larch Prover. Properties of some simple circuits consisting of both combinational and state-holding devices have been proven.

2.5 PETRI NETS

A Petri Net is a graphical and mathematical modeling tool introduced by Petri [2]. Petri Nets are a promising tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri Nets can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

Damm *et al* [16] present a formal semantics of VHDL using a class of Petri Nets called AADL Nets. AADL Nets are condition/event nets, the transitions of which are labeled with transformations on a state space generated by variable and signal assignment statements and whose places can be marked with predicates on a state-space. In an AADL net, every sequential statement constitutes a subnet and the subnets are connected in a cyclic fashion to reflect the cyclic nature of the process statement. The movement of the tokens in the nets reflects the control flow of the VHDL processes. The authors claim that this approach coupled with results from finite state verification methods, such as BDD (Binary Decision Diagrams) based tools and model checkers, proofs of implementation relations between VHDL designs can be obtained. However, no proofs are demonstrated. The characterization restricts itself to variables and signals of type `bit`, single entity-architecture pairs, and non-zero delayed signal assignments.

Olcoz and Colom [35, 38, 37] use colored Petri Nets (CPNs) to model elaborated VHDL. The formalization using CPNs consists of the following basic elements: (i) a collection of user-defined processes, (ii) a kernel process, and (iii) a reliable network communication subsystem between the processes. The user-defined processes are the sets of sequential statements obtained by elaborating the design hierarchy. They execute asynchronously with respect to each other, and communicate due to signal and wait statements present in these sets. The CPN for a process is obtained by translating each sequential statement into the corresponding CPN and then composing the resulting CPNs. The kernel process controls the advancement of simulation time and the communication among the user-defined processes. Tools have been developed to elaborate a VHDL description and to generate CPNs from the elaborated VHDL descriptions. An advantage of using this approach is that the CPNs can be applied independent of the description style and the abstraction at which the VHDL program is presented. One of the limitations of the approach is that the features introduced in VHDL '93 such as postponed processes and pulse rejection limits are not described and no investigations into equivalence proofs are performed.

2.6 EVOLVING ALGEBRAS

Evolving Algebras, as defined by Börger *et al* [8], is *pseudocode over abstract data* without any particular theoretical pre-requisites. The abstract data ele-

ments are members of sets and represent the current state of a system. State transformations are reflected as transformations on these algebras. A sequential EA-machine is represented by a finite set of transition rules of the form

If *Condition* then *Updates*,

where *Condition* is a first order expression, that triggers the execution of all the update instructions in the finite set *Updates*, when it evaluates to true.

Börger *et al* [8] provide a rigorous, but transparent semantic model for VHDL '93 using EA-machines. The semantics defines an abstract VHDL simulation engine in terms of a set of transition rules for an EA-machine. The definition covers the interaction of the simulation kernel with user-defined processes, their suspension and resumption. The semantics is essentially presented as a collection of variables, declared signals, and ports (the abstract data), and functions modeling the simulation engine acting on these abstract data. Issues relating to VHDL '93 constructs such as postponed processes, pulse rejection limits, and shared variables have been discussed. The semantics, however, does not deal with guarded signals, generics, and component instantiations. No proof methodologies have been investigated.

2.7 BOYER-MOORE LOGIC

Boyer-Moore logic [13] is a quantifier free first order logic with equality. The basic theory has axioms defining:

- the boolean constants `t` and `f`, which define the true and false truth values,
- equality,
- an `if-then-else` function, and
- the boolean operations: and, or, not, implies, and iff.

The *Shell Principle* and the *Definitional Principle* allow the user to introduce new ideas into the logic with the guarantee of consistency. The Shell Principle allows the user to add axioms introducing new inductively defined abstract data types. For example, natural numbers are defined inductively by defining a constant `zero` and a constructor function `ADD1`. The definitional principle allows the user to define new functions. For example, the operations of addition, subtraction, and so on are defined using the definitional principle. The rules of inference in the logic are propositional logic, equality, and mathematical induction. The logic is fully supported by the Nqthm theorem prover [13].

Read and Edwards [43] present a mathematical definition of a semantics preserving translation to a subset of VHDL. Essentially, the system operates by transforming VHDL descriptions into equivalent *programs* in Boyer-Moore Logic in two stages. The first stage defines a mapping of the syntax of the

language into expressions in the Boyer-Moore Logic; the mapping is denotational and it creates an abstract syntax of the language. The second stage operationally defines the meaning of the functions used in the first stage. More precisely, it is an operational specification of a simulation kernel for VHDL. All constructs, except expressions, are translated into functions which map the state of the signals and variables before execution into the corresponding state after execution. Entity-Architecture pairs and process statements are translated into nested functions. Although, a significant subset is considered, wait statements are not handled by the semantics. No formal proofs are demonstrated but the authors identify the problems associated with formalizing the semantics in Boyer-Moore logic and suggest strategies to overcome them.

A more successful attempt at proofs is made by Borrione *et al* [9, 11, 10]. They define the semantics of VHDL in terms of a stream functional model wherein an architecture is viewed as a set of functions that express the output ports in terms of the past and current values of the input ports and the initial values of local variables and signals. The Prevail proof system [10] used by them, recognizes a synthesizable subset of VHDL extended with generic parameters. The proof system provides capabilities for proving equivalences of two architectures, diagnosing incorrect architectures, and proving certain temporal properties of an architecture of the circuit. Tools have been created to convert VHDL programs into a VHDL Intermediate form (VIF), and to convert VIF into Proof oriented Intermediate Form (PIF). The advantage of using the PIF is that, specifications in PIF can be converted into input sources for several provers. One of the common methods for verification is to convert the specification in PIF into a format recognized by the Nqthm theorem prover. Nqthm has been integrated into Prevail to verify parameterized functions, library modules and to use induction capabilities (necessary to prove the correctness of the implementation of an n bit comparator for any n). Several other proof systems such as LOVERT [1] and SMOCK [10] are also embedded in the Prevail environment.

2.8 SUMMARY

Several formal techniques have been used to define the semantics of VHDL. In some cases, these formalizations have been embedded in theorem proving environments for automatic formal verification. In most cases, however, the semantics is defined only on paper and is restricted to a small subset of VHDL (the work of Börger *et al* [8] being one of the few exceptions). Further, existing semantic definitions of VHDL restrict themselves to specific types in VHDL (an axiomatic definition by Bickford and Jamsek [7] being an exception). Although Van Tassel [50] provides a framework for proving symbolic equivalence of specific descriptions, there does not exist a semantic model that handles symbolic verification of general VHDL descriptions.

While the current semantic models are very useful in understanding the *simulation cycle* defined by the VHDL LRM, they cannot be used to validate

other methods of simulating VHDL such as parallel simulation. Any simulation technique that deviates from that prescribed by the VHDL LRM cannot fit into these semantic models, and hence cannot be validated. In this book, these limitations are overcome by defining *what* happens when a given VHDL description is evaluated rather than describing *how* it is evaluated.

3 THE STATIC MODEL

This chapter presents a mathematical representation of the static constructs of VHDL. This representation is referred to as the Static Model. The Static Model ignores the elaboration process and assumes that the given VHDL description has been fully elaborated and has correct syntax. It is not one-to-one with VHDL. In fact, many elements of VHDL do not appear in the model. For example, design entities are not presented in the model. They are not necessary. The model need only maintain the leaf level components (*e.g.,* concurrent statements and their constituent subparts) and the netlist specifying their interconnections (*i.e.,* the port hierarchy). In some instances not all of the leaf components that might be expected are present. For example, one might reasonably expect to see a tuple definition for *concurrent signal assignment.* However, such a tuple definition does not exist; instead, the concurrent signal assignment statement is represented using two distinct tuples, namely the selected concurrent signal assignment and the conditional concurrent signal assignment. This decomposition is easier to formally manipulate and the respective forms have distinct translations to process statements.

The following sections assume that the reader is well acquainted with VHDL. At first, an overall picture of the Static Model is presented and then each of the elements of the model are discussed in detail.

vhdl-world	=	< Environment, Concurrent-Statements >
Environment	=	Signals ∪ Variables ∪ Ports ∪ Port-Associations ∪ Types
Concurrent-Statements	=	Block-Stmts ∪ Process-Stmts ∪ Conc-Call-Stmts ∪ Conc-Assert-Stmts ∪ Cond-Signal-Assign ∪ Sel-Signal-Assign ∪ Guarded-Cond-Signal-Assign ∪ Guarded-Sel-Signal-Assign

Figure 3.1. The VHDL world

3.1 THE VHDL WORLD

An elaborated VHDL description may be viewed as a series of design units, a design unit consisting of concurrent statements. A concurrent statement is executed in a given *environment* where *environment* consists of the data declarations and subprograms which are in the scope of the concurrent statement as determined by the elaboration process. Consequently, the Static Model represents the VHDL world as the tuple ⟨ Environment, Concurrent-Statements ⟩ (defined in Figure 3.1). The following sections present the elements of the Static Model.

3.2 SIGNALS

The set Signals in VHDL denote entities that hold values of the machine state. In this model, a distinction between signals declared with a signal declaration and port declarations is made. Port declarations are discussed below and they are not considered to be in the set Signals. Formally, a signal, signal ∈ Signals, can be represented as an 6-tuple:

signal = ⟨ name, res-fn, data-type, signal-kind, disconnection-delay, initial-value ⟩,

signal-kind = [`register` | `bus` | `discrete`].

The following points regarding the elements of signal should be noted:

res-fn: This is the resolution function of the signal. For a non-resolved signal, res-fn = ∅.

signal-kind: Three choices for the signal-kind are allowed. This is in contrast with the general syntax of VHDL where only `register` and `bus` are recognized. However, the functional behavior of a signal changes based on the presence *or absence* of these classifications. Thus, we use the third choice `discrete` to denote the third possibility found in VHDL — a signal declaration without either the `register` or `bus` signal kind.

disconnection-delay: When a disconnection specification is elaborated, a disconnection delay is added to each signal. For guarded signals, disconnection delay describes the delay involved in a null transaction (§5.3 ¶81-83). For non-guarded signals, disconnection-delay = 0.

initial-value: This is the initial value of the signal. If no initial value is provided in the VHDL description, a default value of data-type'LEFT is used.

3.3 VARIABLES

The elements of the set Variables collectively constitute the entire set of variables in a VHDL description. Formally, a variable declaration, variable ∈ Variables, is represented as a 4-tuple:

variable = ⟨ name, shared-status, data-type, initial-value ⟩,

shared-status = [`shared` | `not-shared`].

If initial value for a variable is not declared, then a default initial value is assumed. The default initial value for a variable of a scalar type is data-type'LEFT.

3.4 THE PORT HIERARCHY

3.4.1 Ports

The elements of the set Ports represent all elements of the formal port clauses found in a VHDL description. Formally, a port declaration, port ∈ Ports, is represented as a 6-tuple:

port = ⟨ name, res-fn, data-type, mode, connection, default-expr ⟩,

mode = [`in` | `out` | `inout` | `buffer` | `linkage`].

connection = [*connected* | *unconnected*]

A port is *unconnected* if it is associated with the word open. On the other hand, if the port is associated with an actual port, declared signal or expression, then it is said to be *connected.* A default expression is necessary if the port is of mode `in` and is *unconnected.*

3.4.2 Port Associations

In conjunction with VHDL component instantiations there is a set whose elements link the formal port names given in the component definition with the actual signal (or port) names of the instantiating environment. This set, Port-Associations, contains all of the port associations of an elaborated VHDL description. Formally, a port association, port-association $\in$ Port-Associations, is represented as a 4-tuple:

$$\text{port-association} = \langle \text{formal, fn-f2a, actual, fn-a2f} \rangle$$

where formal (formal $\in$ Ports) is the formal in the association, fn-f2a is a type conversion function that maps the values of the formal's type to the values of the actual's type, actual (actual $\in$ Signals $\cup$ Ports $\cup$ Expressions $\cup$ { open }) is the actual in the association, and fn-a2f is a type conversion function mapping the values of the actual's type to values of the formal's type. In the case where no type conversion function(s) is specified in the VHDL description, fn-a2f and/or fn-f2a will be a function that simply returns the input value.

3.5 DATA TYPES

The set Types contains any and all type definitions used in a VHDL description. Formally, a type $\in$ Types is represented as a 3-tuple:

$$\text{type} = \langle \text{name, parent-type, sequence-of-values} \rangle,$$

$$\text{sequence-of-values} = (\text{values}),$$

where parent-type $\in$ Types denotes the parent type for a given subtype (parent-type $= \emptyset$ if not a subtype), and set-of-values is a set of possible values for an object of the given type. The following points should be noted:

- All types and subtypes are denoted above, including floating point types and physical types.
- The VHDL data type `real` consists of at least the enumeration of a 6 digit mantissa (in the range $\pm$ 1.0) matched with exponents ranging from 0 to $\pm$ 38.
- Secondary units in physical types are ignored and assumed to be expanded as part of the elaboration process.

3.6 EXPRESSIONS

An expression is either an individual data element or a combination of various elements. A data element is, informally, something with which a value is associated, whether it be a variable or a function. More formally, an expression is described as

expression = [unary-expression | binary-expression | port | signal | variable | constant | function-call].

Unary and binary expressions associate an operator with various data elements providing a tree-like structure for describing expressions. It is assumed that precedence and associativity are dealt with in the elaboration process. Formally, an unary expression is represented as a 3-tuple:

unary-expression = ⟨ operator, rhs, data-type ⟩,

and a binary expression as a 4-tuple:

binary-expression = ⟨ operator, lhs, rhs, data-type ⟩,

where

operator = ⟨ operator-symbol, lhs-type, rhs-type ⟩,

lhs, rhs = [expression],

constant = ⟨ value, data-type ⟩,

function-call = ⟨ name, function-name, argument-list, data-type ⟩.

3.7 SUBPROGRAMS

The set Subprograms contains all procedure and function declarations in a VHDL description. Formally, a subprogram ∈ Subprograms is represented as a 5 tuple:

subprogram = ⟨ name, designator, parameter-list, ordered-statements, return-type ⟩,

designator = [`function` | `procedure`],

parameter-list = (parameter),

parameter = ⟨ direction, data-type ⟩,

direction = [`in` | `out` | `inout`].

All functions in the Static Model are assumed to be `pure`, *i.e.*, they do not have any side effects.

3.8 SEQUENTIAL STATEMENTS

The set Statements contains all statements in a VHDL description, both sequential and concurrent. Concurrent statements are dealt with in Section 3.9. Sequential statements appear within a concurrent process statement or a subprogram and are represented as members of the set Sequential-Statements $\subseteq$ Statements. Formally, a sequential statement, seq-statement, is represented by

$$\begin{aligned}\text{seq-statement} \;=\; [\;&\text{wait-stmt} \mid \text{assert-stmt} \mid \text{report-stmt} \mid \\ &\text{signal-assignment} \mid \text{variable-assign} \mid \\ &\text{procedure-call} \mid \text{return-stmt} \mid \text{if-stmt} \mid \text{case-stmt} \mid \\ &\text{loop-stmt} \mid \text{next-stmt} \mid \text{exit-stmt}\;].\end{aligned}$$

Along with a formal representation of the above sequential statements, we will also provide a definition of a function Ψ that, given a statement s, returns a set consisting of all possible statements that may be executed as a result of the execution of s.

Further, the function Ψ on a sequence of statements seq is defined as

$$\Psi(\text{seq}) = \bigcup_1^{|seq|} \Psi(\text{seq}_i).$$

3.8.1 Wait Statement

The set Wait-Stmts (Wait-Stmts $\subseteq$ Statements) defines all *wait statements* in a VHDL description. Formally, a *wait statement*, wait-stmt $\in$ Wait-Stmts, is represented as a 4-tuple:

$$\text{wait-stmt} \;=\; \langle\ \text{name, sensitivity-list, condition, timeout}\ \rangle,$$

$$\text{sensitivity-list} \;=\; \{\ \text{signal} : \text{signal} \in \text{Signals} \cup \text{Ports}\ \},$$

and

$$\Psi(\text{wait-stmt}) = \{\ \text{wait-stmt}\ \}.$$

If no timeout in the original VHDL wait statement, then an equivalent value of infinity is used. A default condition of `true` is assumed if none is specified.

3.8.2 *Assertion Statement*

The set Assertion-Stmts (Assertion-Stmts $\subseteq$ Statements) defines all *assertion statements* in a VHDL description. Formally, an *assertion statement*, assert $\in$ Assertion-Stmts, is represented as a 4-tuple:

$$\text{assert-stmt} = \langle \text{ name, condition, message, severity } \rangle$$

and

$$\Psi(\text{assert-stmt}) = \{ \text{ assert-stmt } \}.$$

The assert message and severity are optionally specified fields, which if undeclared by the user take on default values. The default value for an assertion message is "Assertion violation" and the assert severity defaults to a value of `error`.

3.8.3 *Report Statement*

The set Report-Stmts (Report-Stmts $\subseteq$ Statements) defines all *report statements* in a VHDL description. Formally, a *report statement*, report-stmt $\in$ Report-Stmts, is represented as a 3-tuple:

$$\text{report-stmt} = \langle \text{ name, message, severity } \rangle$$

and

$$\Psi(\text{report-stmt}) = \{ \text{ report-stmt } \}.$$

Report severity assumes a default value of `note` if unspecified by the user.

3.8.4 *Signal Assignment Statement*

The set Signal-Assignments (Signal-Assignments $\subseteq$ Statements) defines all *signal assignment statements* in a VHDL description. Formally, a *signal assignment statement*, signal-assignment $\in$ Signal-Assignments, is represented as a 4-tuple:

$$\text{signal-assignment} = \langle \text{ name, destination, pulse-rejection, waveform } \rangle,$$

$$\text{waveform} = (\text{ wave-element }),$$

$$\text{wave-element}_i = \langle \text{ expr}_i, \text{ delay}_i \rangle,$$

and

$$\Psi(\text{signal-assignment}) = \{ \text{ signal-assignment } \}.$$

All signal assignment statements are assumed to have inertial delay. VHDL signal assignments with the transport delay are represented as inertial delay with a pulse rejection limit of 0 units. The tuple element waveform is a sequence of waveform elements, each element corresponding to the appropriate member from the original VHDL statement.

3.8.5 *Variable Assignment Statement*

The set Variable-Assignments (Variable-Assignments $\subseteq$ Statements) defines all *variable assignment statements* in a VHDL description. Formally, a *variable assignment statement*, variable-assignment $\in$ Variable-Assignments, is represented as a 3-tuple:

$$\text{variable-assignment} = \langle\ \text{name, destination, expression}\ \rangle$$

and

$$\Psi(\text{variable-assignment}) = \{\ \text{variable-assignment}\ \}.$$

3.8.6 *Procedure Call Statement*

The set Procedure-Calls (Procedure-Calls $\subseteq$ Statements) defines all *procedure call statements* in a VHDL description. Formally, a *procedure call statement*, procedure-call $\in$ Procedure-Calls, is represented as a 3-tuple:

$$\text{procedure-call} = \langle\ \text{name, procedure-name, argument-list}\ \rangle,$$

$$\text{argument-list} = (\ \text{expression}\ ),$$

and

$$\Psi(\text{procedure-call}) = \Psi(\text{proc.ordered-stmt}),$$

where proc $\in$ Subprograms $\wedge$ proc.name = procedure-call.name and the expressions in the argument-list are ordered and correspond to the given and/or default arguments in the source VHDL program.

3.8.7 *Return Statement*

The set Return-Stmts, Return-Stmts $\subseteq$ Statements, defines all *return statements* in a VHDL description. Formally, a *return statement*, return-stmt, is represented as a 2-tuple:

$$\text{return-stmt} = \langle\ \text{name, expression}\ \rangle$$

and

$$\Psi(\text{return-stmt}) = \{\ \text{return-stmt}\ \}.$$

3.8.8 *If Statement*

The set If-Stmts (If-Stmts $\subseteq$ Statements) defines all *if statements* in a VHDL description. Formally, an *if statement*, if-stmt $\in$ If-Stmts, is represented as a 2-tuple:

$$\text{if-stmt} = \langle \text{ name, if-choice-list } \rangle,$$

$$\text{if-choice-list} = (\text{ if-choice }),$$

$$\text{if-choice} = \langle \text{ condition, ordered-statements } \rangle,$$

and

$$\Psi(\text{if-stmt}) = \bigcup_{i=1}^{|if-choice-list|} \Psi(\text{if-choice}_i.\text{ordered-statements}).$$

The if statement else clause is represented as an if-choice with the condition of `true`.

3.8.9 *Case Statement*

The set Case-Stmts (Case-Stmts $\subseteq$ Statements) defines all *case statements* in a VHDL description. Formally, a *case statement*, case-stmt $\in$ Case-Stmts, is represented as a 3-tuple:

$$\text{case-stmt} = \langle \text{ name, expression, alternative-list } \rangle,$$

$$\text{alternative-list} = (\text{ alternative }),$$

$$\text{alternative} = \langle \text{ choices, ordered-statements } \rangle,$$

$$\text{choices} = \{ \text{ expression } \},$$

and

$$\Psi(\text{case-stmt}) = \bigcup_{i=1}^{|alternative-list|} \Psi(\text{alternative}_i.\text{ordered-statements}).$$

The model assumes that *VHDL others* clause in choices is properly enumerated into the correct values.

3.8.10 Loop Statement

The set Loop-Stmts (Loop-Stmts $\subseteq$ Statements) defines all *loop statements* in a VHDL description. Formally, a *loop statement*, loop-stmt $\in$ Loop-Stmts, is represented as a 3-tuple:

$$\text{loop-stmt} = \langle \text{ name, condition, ordered-statements } \rangle$$

and

$$\Psi(\text{loop-stmt}) = \Psi(\text{loop-stmt.ordered-statements}).$$

The `for` iteration scheme in VHDL can be rewritten with variable initialization preceding loop-stmt, variable increment appended to ordered-statements, and with terminating condition denoted in condition.

3.8.11 Next Statement

The set Next-Stmts (Next-Stmts $\subseteq$ Statements) defines all *next statements* in a VHDL description. Formally, a *next statement*, next-stmt $\in$ Next-Stmts, is represented as a 3-tuple:

$$\text{next-stmt} = \langle \text{ name, loop-name, condition } \rangle$$

and

$$\Psi(\text{next-stmt}) = \{ \text{ next-stmt } \}.$$

3.8.12 Exit Statement

The set Exit-Stmts (Exit-Stmts $\subseteq$ Statements) defines all *exit statements* in a VHDL description. Formally, an *exit statement*, exit-stmt $\in$ Exit-Stmt, is represented as a 3-tuple:

$$\text{exit-stmt} = \langle \text{ name, loop-name, condition } \rangle$$

and

$$\Psi(\text{exit-stmt}) = \{ \text{ exit-stmt } \}.$$

3.9 CONCURRENT STATEMENTS

Concurrent statements have no order and are represented as members of the set Concurrent-Statements $\subseteq$ Statements. Formally, a concurrent statement, conc-statement $\in$ Concurrent-Statements, is represented by:

$$\begin{aligned}\text{conc-statement} = [\ &\text{block-stmt} \mid \text{process-stmt} \mid \text{conc-call-stmt} \mid \\ &\text{conc-assert-stmt} \mid \text{cond-signal-assign} \mid \\ &\text{sel-signal-assign} \mid \text{guarded-cond-signal-assign} \\ &\text{guarded-sel-signal-assign}\ ].\end{aligned}$$

Since we assume a fully elaborated model, component instantiations and generate statements are not needed.

3.9.1 Block Statement

The set Block-Stmts (Block-Stmts $\subseteq$ Statements) defines all block statements in a VHDL description. Formally, a block statement, block-stmt $\in$ Block-Stmts, is represented as a 3-tuple:

$$\text{block-stmt} = \langle\ \text{name, guard-expression, conc-stmts}\ \rangle,$$

$$\text{conc-stmts} \subseteq \text{Concurrent-Statements}.$$

3.9.2 Process Statement

The set Process-Stmts (Process-Stmts $\subseteq$ Statements) defines all *process statements* in a VHDL description. Formally, a *process statement*, process-stmt $\in$ Process-Stmts, is represented as a 5-tuple:

$$\begin{aligned}\text{process-stmt} = \langle\ &\text{name, postponement, sensitivity-list,} \\ &\text{ordered-statements, set-of-drivers}\ \rangle,\end{aligned}$$

$$\text{postponement} = [\ \texttt{postponed} \mid \texttt{not-postponed}\ ],$$

$$\text{set-of-drivers} = \{\ \text{set-of-drivers} : \text{set-of-drivers} = \langle\ \text{destination, tr-list}\ \rangle\ \},$$

$$\text{tr-list} = \{\ \text{tr} : \text{tr} = \langle\ \text{value, time}_r\text{, time}_d\ \rangle\ \},$$

and

$$\Psi(\text{process-stmt}) = \Psi(\text{process-stmt.ordered-statements}).$$

The elements set-of-drivers and tr-list will be dealt with when defining the dynamic semantics of the process statement.

3.9.3 Concurrent Procedure Call Statement

The set Conc-Call-Stmts (Conc-Call-Stmts $\subseteq$ Statements) defines all *concurrent procedure statements* in a VHDL description. Formally, a *concurrent procedure call statement*, conc-call-stmt $\in$ Conc-Call-Stmts, is represented as a 4-tuple:

conc-call-stmt = ⟨ name, postponement, procedure-name, argument-list ⟩.

3.9.4 Concurrent Assertion Statement

The set Conc-Assert-Stmts (Conc-Assert-Stmts $\subseteq$ Statements) defines all *concurrent assertion statements* in a VHDL description. Formally, a *concurrent assertion statement*, conc-assert-stmt $\in$ Conc-Assert-Stmts, is represented as a 5-tuple:

conc-assert-stmt = ⟨ name, postponement, condition, message, severity ⟩.

3.9.5 Concurrent Signal Assignment Statement

Concurrent signal assignments are represented using two different tuples. One corresponds to conditional signal assignments and the other corresponds to selected signal assignments.

Conditional Signal Assignment. In particular, the set Cond-Signal-Assigns (Cond-Signal-Assigns $\subseteq$ Statements) defines all unguarded *conditional signal assignment statements* in a VHDL description. Formally, a *conditional signal assignment statement*, cond-signal-assign $\in$ Cond-Signal-Assigns, is represented as a 5-tuple:

cond-signal-assign = ⟨ name, postponement, destination, pulse-rejection, cond-waveform-selections ⟩.

For the case of a guarded signal assignment statement, a separate tuple is defined with an added implicit guard signal. The value of the guard signal is calculated by evaluating the *guard-expression* associated with it. Thus we can define guarded-cond-signal-assign $\in$ Guarded-Cond-Signal-Assigns as

guarded-cond-signal-assign = ⟨ name, postponement, guard, destination, pulse-rejection, cond-waveform-selections ⟩,

cond-waveform-selections = (cond-waveform),

cond-waveform = ⟨ condition, waveform ⟩.

If a simple waveform is specified in the original VHDL program, then it is represented in the model as a cond-sig-assign with one cond-waveform having the condition specified as `true` and the waveform as given in the VHDL source.

Selected Signal Assignment. The set Sel-Signal-Assigns defines all unguarded *selected signal assignment statements* in a VHDL description. Formally, a *selected signal assignment statement*, sel-signal-assign $\in$ Sel-Signal-Assigns, is represented as a 6-tuple:

$$\begin{aligned}\text{sel-signal-assign} \;=\; \langle\ &\text{name, postponement, destination,}\\ &\text{pulse-rejection, expression,}\\ &\text{sel-waveform-selections}\ \rangle.\end{aligned}$$

In the guarded case we define guarded-sel-signal-assign $\in$ Guarded-Sel-Signal-Assigns as

$$\begin{aligned}\text{guarded-sel-signal-assign} \;=\; \langle\ &\text{name, postponement, guard, destination,}\\ &\text{pulse-rejection, expression,}\\ &\text{sel-waveform-selections}\ \rangle,\end{aligned}$$

$$\text{sel-waveform-selections} \;=\; (\ \text{sel-waveform}\ ),$$

$$\text{sel-waveform} \;=\; \langle\ \text{choices, waveform}\ \rangle,$$

where guard is as described in Section 3.9.5. In sel-waveform, we assume that any VHDL `others` clause in choices is properly enumerated into the correct values.

3.10 SUMMARY

The above sections provided a mathematical framework in which the static structures of VHDL can be represented. The framework can now be used to reason about VHDL semantics. The next chapter formalizes the well-formedness rules of VHDL using this framework.

4 A WELL-FORMED VHDL MODEL

This chapter presents a methodology for defining formally what it means for a Static Model representation of a VHDL description to be well-formed. The well-formedness rules are written as a series of axioms describing conditions which must hold true for the Static Model to be considered correct. Each well-formedness condition is written as an informal requirement with a reference to the appropriate section in the LRM followed by the formal axiom.

Note that a distinction is made between a well-formed VHDL program and the well-formed Static Model. A well-formed VHDL program is described by the syntax and semantics given in the LRM. The axioms for describing Static Model well-formedness are a subset of the well-formed VHDL program axioms. The subset may be characterized as those axioms which still apply after the elaboration process has been completed. For instance, scoping rules are assumed dealt with during elaboration. Once the source program is elaborated into it's Static Model form, the information necessary to do scope checking has been lost. The well-formedness rules are elaborated in the following sections.

4.1 SIGNALS

The type of the resolution function provided must match the type of the signal (§2.4, ¶27, Ł340).

$$\forall \text{ signal} : \text{signal} \in \text{Signals} \bullet$$

(signal.res-fn $\neq \emptyset \wedge$
signal.data-type $\in$ Types $\wedge$
signal.data-type = signal.res-fn.data-type) $\vee$
signal.res-fn $= \emptyset$.

Note that validity of a given type is checked by set membership in the set Types.

A *non-resolved* signal may not have multiple sources. This corresponds to the signal being in at most one transaction list or having at most one formal port associated with it (§4.3.1.2, ¶55, L185).

$\forall$ signal : signal $\in$ Signals •
signal.res-fn $= \emptyset \Rightarrow$
(($\forall$ p, q, tr_i, tr_j : p, q $\in$ Process-Statements •
($\langle$ signal, tr_i $\rangle \in$ p.set-of-drivers $\wedge$
$\langle$ signal, tr_j $\rangle \in$ q.set-of-drivers) $\Rightarrow$ p = q) $\wedge$
$\neg \exists$ pa: pa $\in$ Port-Associations • pa.actual = signal) $\vee$
(($\forall$ pa_i, pa_j: pa_i, $pa_j \in$ Port-Associations •
(pa_i.actual = signal $\wedge$ pa_j.actual = signal) $\Rightarrow$ $pa_i = pa_j$) $\wedge$
$\neg \exists$ p, tr_i: p $\in$ Process-Statements •
$\langle$ signal, tr_i $\rangle \in$ p.set-of-drivers) .

4.2 VARIABLES

The type of the initial value expression must be the same type as the variable (§4.3.1.3, ¶56, L230).

$\forall$ variable : variable $\in$ Variables •
variable.data-type $\in$ Types $\wedge$
variable.data-type = variable.initial-value.data-type .

4.3 THE PORT HIERARCHY

4.3.1 Ports

A `buffer` port may have at most one source (§1.1.1.2, ¶8, L120).

$\forall$ prt : prt $\in$ Ports •
prt.mode = `buffer` $\Rightarrow$
(($\forall$ p, q, tr_i, tr_j : p, q $\in$ Process-Statements •
($\langle$ prt, tr_i $\rangle \in$ p.set-of-drivers $\wedge$
$\langle$ prt, tr_j $\rangle \in$ q.set-of-drivers) $\Rightarrow$ p = q) $\wedge$
$\neg \exists$ pa: pa $\in$ Port-Associations • pa.actual = prt) $\vee$
(($\forall$ pa_i, pa_j: pa_i, $pa_j \in$ Port-Associations •
(pa_i.actual = prt $\wedge$ pa_j.actual = prt) $\Rightarrow$ $pa_i = pa_j$) $\wedge$
$\neg \exists$ p, tr_i: p $\in$ Process-Statements •
$\langle$ prt, tr_i $\rangle \in$ p.set-of-drivers) .

4.3.2 Port Associations

After elaboration, if a formal port is associated with an actual that is itself a port, then port association is subject to certain restrictions (§1.1.1.2, ¶7, Ł95). For example, a formal of mode `in` many not be associated with an actual of mode `out`. The axiom which formally describes these limits is:

$$
\begin{aligned}
&\forall\ \mathsf{pa} : \mathsf{pa} \in \text{Port-Associations} \wedge \mathsf{pa.formal} \in \text{Ports} \wedge \\
&\quad \mathsf{pa.actual} \in \text{Ports} \bullet \\
&\quad (\mathsf{pa.formal.mode} = \texttt{in} \wedge \\
&\quad\ \mathsf{pa.actual.mode} = [\texttt{in} \mid \texttt{inout} \mid \texttt{buffer}\ ]) \vee \\
&\quad (\mathsf{pa.formal.mode} = \texttt{out} \wedge \\
&\quad\ \mathsf{pa.actual.mode} = [\ \texttt{out} \mid \texttt{inout}\ ]) \vee \\
&\quad (\mathsf{pa.formal.mode} = \texttt{inout} \wedge \mathsf{pa.actual.mode} = \texttt{inout}) \vee \\
&\quad (\mathsf{pa.formal.mode} = \texttt{buffer} \wedge \mathsf{pa.actual.mode} = \texttt{buffer}) \vee \\
&\quad (\mathsf{pa.formal.mode} = \texttt{linkage} \wedge \\
&\quad\ \mathsf{pa.actual.mode} = [\texttt{in} \mid \texttt{out} \mid \texttt{inout} \mid \texttt{buffer} \mid \texttt{linkage}\ ])\ .
\end{aligned}
$$

The subtype of the formal (§12.2.4, ¶156, Ł130) must equal the subtype of the actual.

$$
\begin{aligned}
&\forall\ \mathsf{pa} : \mathsf{pa} \in \text{Port-Associations} \bullet \\
&\quad \mathsf{pa.formal.data\text{-}type} \in \text{Types} \wedge \mathsf{pa.actual.data\text{-}type} \in \text{Types} \wedge \\
&\quad \mathsf{pa.formal.data\text{-}type} = \mathsf{pa.actual.data\text{-}type}\ .
\end{aligned}
$$

Any actual associated with a formal `buffer` port may have at most one source (§1.1.1.2, ¶7, Ł110). This source is the formal itself and hence no other formal or transaction list can be associated with the actual.

$$
\begin{aligned}
&\forall\ \mathsf{pa} : \mathsf{pa} \in \text{Port-Associations} \wedge \mathsf{pa.formal.mode} = \texttt{buffer} \bullet \\
&\quad (\neg\ \exists\ \mathsf{pa'}\text{: } \mathsf{pa'} \in \text{Port-Associations} \bullet \\
&\quad\quad \mathsf{pa'} \neq \mathsf{pa},\ \mathsf{pa'.actual} = \mathsf{pa.actual}) \wedge \\
&\quad (\neg\ \exists\ \mathsf{p},\ \mathsf{tr}_i\text{: } \mathsf{p} \in \text{Process-Statements} \bullet \\
&\quad\quad \langle\ \mathsf{pa.actual},\ \mathsf{tr}_i\ \rangle \in \mathsf{p.set\text{-}of\text{-}drivers}\ )\ .
\end{aligned}
$$

4.4 DATA TYPES

A well-formed VHDL program must contain at least certain types. The minimal set of types is defined in the STANDARD package section of the Language Reference Manual (§14.2, ¶195-206). The types are described here as tuples of the form type:

$$
\begin{aligned}
\mathsf{boolean} \quad = \quad &\langle\ \mathsf{boolean},\ \emptyset, \\
&\quad (\ \texttt{false},\ \texttt{true}\ )\ \rangle,
\end{aligned}
$$

bit = ⟨ bit, ∅,
('0', '1') ⟩,

character = ⟨ character, ∅,
('0' ... '9' ... 'A' ... 'Z', 'a' ... 'z', ...) ⟩,

integer = ⟨ integer, ∅,
(-2147483647 ... -2, -1, 0, 1, 2 ... 2147483647) ⟩,

positive = ⟨ positive, integer,
(1, 2, ..., integer'high) ⟩,

natural = ⟨ natural, integer,
(0, 1, ..., integer'high) ⟩,

real = ⟨ real, ∅,
(-1.000000E38 ... -0.999999E-38, 0,
0.999999E-38 ... 1.000000E38) ⟩,

severity-level = ⟨ severity-level, ∅,
(note, warning, error, failure) ⟩.

The definition of type time must at least include the range -2147483647 to 2147483647. Formally,

time = ⟨ time, ∅,
(-2147483647 ... -2, -1, 0, 1, 2 ... 2147483647) ⟩.

Thus, at a minimum the set Minimal-Types (Minimal-Types $\subseteq$ Types) would consist of the union over all of the previous tuple definitions. Formally:

Minimal-Types = {boolean, bit, character, integer, real,
severity-level, time, positive, natural, } .

4.5 EXPRESSIONS

4.5.1 Unary and Binary Expressions

This section is organized according to the various valid operators in a VHDL expression. The first few sections demonstrate how the axioms may be written for given operators. Subsequent sections merely list the conditions which must hold for the various operators.

Logical Operators. A logical operator is defined as

logical-operator = [and | or | nand | nor | xor | xnor].

Logical operators are defined for the types bit (§7.2.1, ¶93, Ł70) and boolean.

∀ expr : expr ∈ Binary-Expressions •
expr.operator.operator-symbol = logical-operator ∧
(expr.operator.lhs-type = `boolean` ∧
expr.operator.rhs-type = `boolean` ∧
expr.data-type = `boolean`) ∨
(expr.operator.lhs-type = `bit` ∧
expr.operator.rhs-type = `bit` ∧
expr.data-type = `bit`) .

Relational Operators. A relational operator is defined as

relational-operator = [= | /= | < | <= | > | >=].

Relational operators are defined for all types with the restriction that the types of the operands must be the same (§7.2.2, ¶93, Ł100).

∀ expr : expr ∈ Binary-Expressions •
expr.operator.operator-symbol = relational-operator ∧
expr.operator.lhs-type = expr.operator.rhs-type ∧
expr.data-type = `boolean` .

NOTE: When compound types are added this will not be true. Less than and greater than operators are defined for only scalar types and discrete array types.

Shift Operators. All the shift operators are defined for any one dimensional array type whose element type is either of the predefined types bit or boolean (§7.2.3, ¶94, Ł135).

A shift operator is defined as

shift-operator = [sll | srl | sla | sra | rol | ror].

Adding / Sign Operators. The LRM makes a distinction between adding and sign operators. A concatenation operator is also viewed as an adding operator. For semantic purposes we will treat the sign operators similar to the adding operators. The concatenation operator will be treated separately. An adding operator is defined as

adding-operator = [+ | -].

The adding operators are defined on any numeric type. In the case where an adding operator is used as a sign the left hand side of the expression will be null.

A concatenation operator is defined as

concatenation-operator = [&].

Multiplying Operators. A multiplying operator is defined as

multiplying-operator = [* | / | mod | rem].

The multiplying operators are defined on any numeric type.

Miscellaneous Operators. The unary operator abs is defined on any numeric type. The exponentiation operator ** is defined on integer types and floating point types. In either case the right operand, or exponent, must be of type integer.

miscellaneous-operators = [abs | **].

4.5.2 Constants

The value of the constant must belong to the data type of the declared constant (§4.3.1.1, ¶54, Ł140).

$$\forall \text{ constant} : \text{constant} \in \text{Constants} \bullet \\ \text{constant.data-type} \in \text{Types} \wedge \\ \text{constant.value} \in \text{constant.data-type.sequence-of-values} .$$

A constant may not be modified after elaboration. This is mentioned in this section for emphasis, but is formally taken care of in other axioms such as the signal assignment axioms (§4.3.1.1, ¶54, Ł140).

4.5.3 Function Call

For a function call, the size of the argument list must match the size of the defined parameter list (§8.6, ¶120, Ł380).

$$\forall \text{ func} : \text{func} \in \text{Function-Calls} \bullet$$
$$(\exists \text{ subprog-def} : \text{subprog-def} \in \text{Subprograms} \bullet$$
$$\text{subprog-def.designator} = \texttt{function} \wedge$$
$$\text{subprog-def.name} = \text{func.function-name} \wedge$$
$$|\text{ subprog-def.parameter-list }| = |\text{ func.argument-list }|) .$$

4.6 SEQUENTIAL STATEMENTS

4.6.1 Wait Statement

For a wait statement, *read* access must be permitted on all signals in the sensitivity list (§8.1, ¶112, Ł30). According to the LRM, ports of mode `in` or `inout` have *read* access. Formally,

$$\forall \text{ wa-stmt} : \text{wa-stmt} \in \text{Wait-Stmts} \bullet$$
$$\forall \text{ prt} : \text{prt} \in \text{wa-stmt.sensitivity-list} \wedge \text{prt} \in \text{Ports} \bullet$$
$$(\text{port.mode} = \texttt{in} \vee \text{port.mode} = \texttt{inout}) .$$

A wait statement may not appear in a process statement which contains a sensitivity list (§8.1, ¶113, Ł75). Formally:

$$\neg\exists \text{ pr, wait-stmt} : \text{pr} \in \text{Process-Statements, wait-stmt} \in \text{Wait-Stmts} \bullet$$
$$\text{pr.sensitivity-list} \neq \emptyset \wedge$$
$$\text{wait-stmt} \in \Psi(\text{pr.ordered-statements}) .$$

The condition clause must be of type boolean (§8.1, ¶111, Ł25).

$$\forall \text{ wa-stmt} : \text{wa-stmt} \in \text{Wait-Stmts} \bullet$$
$$\text{wa-stmt.condition.data-type} = \texttt{boolean} .$$

The timeout clause must be of type time (§8.1, ¶111, Ł25).

$$\forall \text{ wa-stmt} : \text{wa-stmt} \in \text{Wait-Stmts} \bullet$$
$$\text{wa-stmt.timeout.data-type} = \texttt{time} .$$

4.6.2 Assertion Statement

An assertion statement condition (§8.2, ¶118, Ł125) must be a boolean expression:

$$\forall \text{ assert} : \text{assert} \in \text{Assertion-Stmts} \bullet \\ \text{assert.condition.data-type} = \texttt{boolean} .$$

Similarly, the assertion message expression should be of type string:

$$\forall \text{ assert} : \text{assert} \in \text{Assertion-Stmts} \bullet \\ \text{assert.message.data-type} = \texttt{string}$$

and the severity expression must be of type severity-level:

$$\forall \text{ assert} : \text{assert} \in \text{Assertion-Stmts} \bullet \\ \text{assert.severity.data-type} = \texttt{severity-level} .$$

4.6.3 Report Statement

For a report statement (§8.3, ¶114, Ł135), the message reported must be of type string:

$$\forall \text{ report} : \text{report} \in \text{Report-Stmts} \bullet \\ \text{report.message.data-type} = \texttt{string}$$

and the severity expression must be of type severity-level:

$$\forall \text{ report} : \text{report} \in \text{Report-Stmts} \bullet \\ \text{report.severity.data-type} = \texttt{severity-level} .$$

4.6.4 Signal Assignment Statement

A signal assignment statement must assign to a declared signal or port (of any mode other than `in` (§8.4, ¶115, Ł175).

$$\forall \text{ sa-stmt} : \text{sa-stmt} \in \text{Signal-Assignments} \bullet \\ \text{sa-stmt.destination} \in \text{Signals} \vee \\ (\text{sa-stmt.destination} \in \text{Ports} \wedge \text{sa-stmt.destination.mode} \neq \texttt{in}) .$$

The base type of the waveform expressions must equal the base type of the declared signal type (§8.4, ¶115, Ł165). To define this formally, a function BASE_TYPE is first introduced to find the base type of a given data type.

$$\text{BASE\_TYPE}(x) =$$

if x.parent-type = ∅ then x
else BASE_TYPE(x.parent-type).

Then the axiom for comparing waveform data type is defined as:

∀ sa-stmt : sa-stmt ∈ Signal-Assignments •
(∀ $expr_i$ ∈ sa-stmt.waveform •
$expr_i$.data-type ∈ Types ∧
sa-stmt.destination.data-type ∈ Types ∧
BASE_TYPE($expr_i$.data-type) =
BASE_TYPE(sa-stmt.destination.data-type.sequence-of-values)) .

The base type of the time expression in each waveform element must be of the predefined type `time` (§8.4.1, ¶117, Ł235).

∀ sa-stmt : sa-stmt ∈ Signal-Assignments •
(∀ $delay\text{-}elem_i$ ∈ sa-stmt.waveform •
$delay\text{-}elem_i$.data-type.parent-type = `time`) .

4.6.5 *Variable Assignment Statement*

A variable assignment statement must assign to a declared variable and the base type of the assigned expression must match the base type of the declared variable (§8.5, ¶119, Ł335). More precisely:

∀ va-stmt : va-stmt ∈ Variable-Assignment •
va-stmt.destination ∈ Variables ∧
variable.data-type ∈ Types ∧
va-stmt.expression.data-type ∈ Types ∧
BASE_TYPE(variable.data-type) =
BASE_TYPE(va-stmt.expression.data-type)) .

The above axiom also ensures that a variable assignment statement does not assign to a constant (§4.3.1.1, ¶54, Ł140).

4.6.6 *Procedure Call Statement*

For a procedure call statement, the size of the argument list must match the size of the defined parameter list (§8.6, ¶120, Ł380).

∀ proc-stmt : proc-stmt ∈ Procedure-Stmts •
(∃ subprog-def : subprog-def ∈ Subprograms •
subprog-def.designator = `procedure` ∧
subprog-def.name = proc-stmt.procedure-name ∧
| subprog-def.parameter-list | = | proc-stmt.argument-list |) .

4.6.7 Return Statement

The type of the return expression must match the base type of the appropriate specified subprogram (§8.12, ¶124, Ł520). In the case of a procedure, the return type will be the empty set.

$$
\begin{aligned}
&\forall \text{ proc-sub} : \text{proc-sub} \in \text{Subprograms} \bullet \\
&\quad (\forall \text{ stmt}_i : \text{stmt}_i \in \\
&\qquad \Psi\ (\text{proc-sub.ordered-statements}_i) \cap \text{Return-Stmts} \bullet \\
&\quad \text{stmt.expression.data-type} \in \text{Types} \wedge \\
&\quad \text{proc-sub.return-type} \in \text{Types} \wedge \\
&\quad \text{stmt.expression.data-type} = \text{BASE\_TYPE}(\text{proc-sub.return-type}))\ .
\end{aligned}
$$

Return statements (§8.12, ¶124, Ł515) are only allowed within a subprogram. This is formulated as a series of axioms stating which tuples may not have *return statements* in their ordered statement lists.

$$
\begin{aligned}
&\neg\exists \text{ if-stmt, return-stmt} : \text{if-stmt} \in \text{If-Stmts}, \\
&\qquad \text{return-stmt} \in \text{Return-Stmts} \bullet \\
&\qquad (\forall\ i{:}\ 1 \leq i \leq |\ \text{if-stmt.if-choice-list}\ | \bullet \\
&\qquad \text{return-stmt} \in \Psi(\text{if-stmt.if-choice}_i\text{.ordered-statements}))\ .
\end{aligned}
$$

$$
\begin{aligned}
&\neg\exists \text{ case-stmt, return-stmt} : \text{case-stmt} \in \text{Case-Stmts}, \\
&\qquad \text{return-stmt} \in \text{Return-Stmts} \bullet \\
&\qquad (\forall\ i{:}\ 1 \leq i \leq |\ \text{case-stmt.if-choice-list}\ | \bullet \\
&\qquad \text{return-stmt} \in \Psi(\text{case-stmt.alternative}_i\text{.ordered-statements}))\ .
\end{aligned}
$$

$$
\begin{aligned}
&\neg\exists \text{ stmt, return-stmt} : \text{stmt} \in \text{Loop-Stmts} \cup \text{Process-Stmts} \bullet \\
&\qquad \text{return-stmt} \in \Psi(\text{stmt.ordered-stmts})\ .
\end{aligned}
$$

4.6.8 If Statement

The if condition must be a boolean expression (§8.7, ¶121).

$$
\begin{aligned}
&\forall \text{ if-stmt} : \text{if-stmt} \in \text{If-Stmts} \bullet \\
&\quad (\forall \text{ cond}_i : \text{cond}_i \in \text{if-stmt.if-choice-list} \bullet \\
&\quad \text{cond}_i\text{.data-type} = \texttt{boolean})\ .
\end{aligned}
$$

4.6.9 Case Statement

Each choice in the case statement choice list must be of the same type as the determining expression (§8.8, ¶121, Ł420).

$$
\begin{aligned}
&\forall \text{ case} : \text{case} \in \text{Case-Stmts} \bullet \\
&\quad (\forall \text{ choice} : \text{choice} \in \text{case.alternative-list}_i\text{.choices} \bullet \\
&\quad \text{choice.expression.data-type} \in \text{Types} \wedge \\
&\quad \text{case.expression.data-type} \in \text{Types} \wedge \\
&\quad \text{choice.expression.data-type} = \text{case.expression.data-type})\ .
\end{aligned}
$$

4.6.10 Loop Statement

The loop condition must be a boolean expression (§8.9, ¶123).

∀ loop : loop ∈ Loop-Stmts •
    loop.condition.data-type = `boolean` .

4.6.11 Next Statement

The next condition must be a boolean expression (§8.10, ¶123).

∀ next : next ∈ Next-Stmts •
    next.condition.data-type = `boolean` .

4.6.12 Exit Statement

The exit condition must be a boolean expression (§8.11, ¶124).

∀ exit : exit ∈ Exit-Stmts •
    exit.condition.data-type = `boolean` .

4.7 CONCURRENT STATEMENTS

4.7.1 Conditional Signal Assignment

A conditional signal assignment without a guard expression may not have a destination which is a guarded target. Formally this is expressed as

∀ ccsa : ccsa ∈ Cond-Signal-Assigns •
    ccsa.destination.signal-kind ≠ `register` | `bus` .

In the guarded form of the conditional signal assignment the guard expression must be of type boolean (§9.1, ¶126, Ł45).

∀ guarded-ccsa : guarded-ccsa ∈ Guarded-Cond-Signal-Assigns •
    guarded-ccsa.guard-expression.data-type = `boolean` .

4.7.2 Selected Signal Assignment

Likewise, a selected signal assignment without a guard expression may not have a destination which is a guarded target. Formally this is expressed as

∀ scsa : scsa ∈ Sel-Signal-Assigns •
    scsa.destination.signal-kind ≠ `register` | `bus` .

In the guarded form of the selected signal assignment the guard expression must be of type boolean (§9.1, ¶132, Ł45).

∀ guarded-scsa : guarded-scsa ∈ Guarded-Sel-Signal-Assigns •
    guarded-ccsa.guard-expression.data-type = `boolean` .

4.8 SUMMARY

This chapter defined the axioms that must hold for the Static Model representation of a VHDL description to be considered well-formed. More importantly, it demonstrated a technique for formalizing the statements made in the LRM about the static semantics of VHDL programs. The next chapter defines a minimal form of the Static Model. The dynamic semantics of VHDL presented in Chapter 8 builds on top of this minimal form.

5 THE REDUCTION ALGEBRA

The Static Model presented in Chapter 3 has some redundant constructs. For example, the LRM states that every concurrent statement can be transformed into an equivalent process statement. It is desirable to define a form of the Static Model that is less cumbersome and devoid of redundancies. This chapter defines the minimal form of the Static Model. It particular, it provides an algebra (called the reduction algebra) that reduces a given Static Model representation of a VHDL description into its minimal (or *normal*) form. It is important to note that the reductions are based on equivalences that are defined in the LRM. The Dynamic Model presented in Chapter 8 is based on this *normal* form. The following sections define the reduction algebra by defining a reduction operator for each VHDL statement (if a reduction for that statement is defined in the LRM).

5.1 SIGNAL ASSIGNMENT STATEMENTS

Recall that a signal assignment statement is a 4-tuple:

signal-assignment = ⟨ name, destination, pulse-rejection, waveform ⟩

and waveform is a sequence of tuples

waveform = (wave-element),

$$\text{wave-element}_i \;=\; \langle\, \text{expr}_i,\ \text{delay}_i \,\rangle.$$

For the dynamic model, we would like to use a slightly reduced form of the signal assignment statement. Fortunately, a sequential signal assignment statement with multiple waveforms can be reduced to a sequence of sequential assignments with single waveforms as shown.

```
s <= reject 5 ns inertial a after 10 ns,
     b after 20 ns, c after 30 ns;
                                ⇓

             s <= reject 5 ns inertial a after 10 ns;
             s <= reject 0 ns inertial b after 20 ns;
             s <= reject 0 ns inertial c after 30 ns;
```

Formally, for any signal assignment statement, sa, this reduction can be defined in terms of the function SA-REDUCE(sa) as

$$\text{SA-REDUCE(sa)} \;\equiv\; (\text{rsa}_i)$$

where

$$\begin{aligned}\text{rsa}_1 \;=\; \langle\ & \text{sa.name},\\ & \text{sa.destination},\\ & \text{sa.pulse-rejection},\\ & \text{sa.waveform.expr}_i,\\ & \text{sa.waveform.delay}_i\ \rangle\end{aligned}$$

and $\forall i : 2 \leq i \leq |\text{sa.waveform}|$,

$$\begin{aligned}\text{rsa}_i \;=\; \langle\ & \text{sa.name},\\ & \text{sa.destination},\\ & 0,\\ & \text{sa.waveform.expr}_i,\\ & \text{sa.waveform.delay}_i\ \rangle.\end{aligned}$$

Put simply, the original signal assignment statement is rewritten as a sequence of reduced signal assignment statements, each with only one element of the original waveform. The first reduced signal assignment statement contains the original pulse rejection limit and the remaining have a pulse rejection limit of 0 units.

5.2 CONCURRENT STATEMENTS

Except for the process statement, all concurrent statements in VHDL have an equivalent process statement implementation. In this section, we describe their translation into the equivalent process statement(s).

5.2.1 *Process Statements*

A process statement with a sensitivity list can be rewritten into a process statement without one. More precisely, a wait statement with the same sensitivity list as the original process is appended to the list of ordered statements in the new process (§9.2, ¶127) as shown below.

```
proc:  process(clock)                          proc:  process
begin                                          begin
    s <= a and b after 10 ns;          ⇒           s <= a and b after 10 ns;
end process proc;                                  wait on clock;
                                               end process proc;
```

Formally, for a process statement defined as the 5-tuple

process-stmt = ⟨ name, postponement, sensitivity-list, ordered-statements, set-of-drivers ⟩,

an equivalent reduced form is the 4-tuple

reduced-process-stmt = ⟨ name, postponement, ordered-statements, set-of-drivers ⟩

where the transform from a general process statement ps to a reduced form process statement rps is defined by the function PS-REDUCE(ps):

PS-REDUCE(ps) ≡ ⟨ ps.name,
ps.postponement,
(ps.ordered-statements, ps-wait-sensitivity),
ps.set-of-drivers ⟩

where ps-wait-sensitivity is a WAIT statement of the form

ps-wait-sensitivity = ⟨ ps.name,
ps.sensitivity-list,
true,
infinity ⟩.

5.2.2 *Concurrent Procedure Calls*

Every concurrent procedure call statement cpc represented by the 4-tuple

$$\begin{aligned}\text{conc-call-stmt} \;=\; &\langle\ \text{name, postponement, procedure-name,}\\ &\quad\text{argument-list}\ \rangle\end{aligned}$$

can be reduced to an equivalent passive process statement ps, where the ordered statements in ps consist of the sequential form of the procedure call and a wait statement that is sensitive to the signals in cpc.argument-list which have a mode `in` or `inout` (§9.3, ¶128). This reduction is formalized in terms of the function CPC-REDUCE(cpc):

$$\begin{aligned}\text{CPC-REDUCE(cpc)} \;\equiv\; &\langle\ \text{cpc.name,}\\ &\quad\text{cpc.postponement,}\\ &\quad\text{(cpc-call, cpc-wait),}\\ &\quad\text{tr}\ \rangle\end{aligned}$$

where cpc-call is a procedure call statement of the form

$$\begin{aligned}\text{cpc-call} \;=\; &\langle\ \text{cpc.name,}\\ &\quad\text{cpc.procedure-name,}\\ &\quad\text{cpc.argument-list}\ \rangle,\end{aligned}$$

cpc-wait is a wait statement of the form

$$\begin{aligned}\text{cpc-wait} \;=\; &\langle\ \text{cpc.name,}\\ &\quad\text{cpc-sensitivity-list,}\\ &\quad\texttt{true,}\\ &\quad\texttt{infinity}\ \rangle,\end{aligned}$$

cpc-sensitivity-list is defined as

$$\begin{aligned}\text{cpc-sensitivity-list} \;=\; &\{\ \langle\ s_i : s_i \in \text{cpc.argument-list} \wedge\\ &\quad\text{proc-def} \in \text{Subprograms}\\ &\quad\wedge\ \text{proc-def.name} = \text{cpc.procedure-name} \wedge\\ &\quad p_i \in \text{proc-def.parameter-list} \wedge\\ &\quad((p_i\text{.mode} = \texttt{in}) \vee (p_i\text{.mode} = \texttt{inout}))\ \},\end{aligned}$$

and tr is defined as

$$\begin{aligned}\text{tr} \;=\; &\{\ \langle\ \text{dest} : \text{dest} \in \text{cpc.argument-list} \wedge ((\text{dest.mode} = \texttt{out}) \vee\\ &\quad(\text{dest.mode} = \texttt{inout}))\ \}.\end{aligned}$$

5.2.3 *Concurrent Assertion Statements*

Every concurrent assertion statement ca can be reduced to an equivalent passive process statement ps. The ordered statements in ps consist of the sequential form of the assertion statement followed by a wait statement that is sensitive to the signals in ca.condition (§9.4, ¶129) as shown below.

```
postponed assert (not a) report ''violation'' severity warning;
                                    ⇓

proc:  postponed process
begin
    assert (not a) report ''violation'' severity warning;
    wait on a;
end process proc;
```

Formally, a concurrent assertion statement represented by

$$\text{conc-assert-stmt} = \langle\ \text{name, postponement, expression, message, severity}\ \rangle$$

is translated into a passive process statement by the function CAS-REDUCE(cas) defined as

$$\text{CAS-REDUCE(csa)} \equiv \langle\ \text{cas.name, cas.postponement, (assertion, cas-wait), } \emptyset\ \rangle$$

where assertion is an assertion statement of the form

$$\text{assertion} = \langle\ \text{cas.name, cas.expression, cas.message, cas.severity}\ \rangle$$

and cas-wait is defined as

$$\text{cas-wait} = \langle\ \text{cas.name}, \emptyset, \texttt{true}, \texttt{infinity}\ \rangle.$$

5.2.4 *Conditional Concurrent Signal Assignment Statements*

A conditional concurrent assignment assigns a waveform to the destination signal if some conditions are met, and can be rewritten as a process statement (§9.5.1, ¶132) with an `if_then_elseif` statement as shown below.

```
s <= a after 10 ns, b after 20 ns
     when (a=b) else not a after 5 ns when
     (not a) else c after 10 ns when (not c);

                              ⇓

      proc:  process
      begin
          if (a=b) then
              s <= a after 10 ns, b after 20 ns;
          elseif (not a) then
              s <= not a after 5 ns;
          elseif (not c) then
              s <= c after 10 ns;
          endif;
          wait on a, b, c;
      end process proc;
```

The reduction algebra for conditional concurrent signal assignments differs for the tuples with a guard-expression and without. However, due to the similarities in the transforms, they will be presented together in this section. First the case in which no guard expression is present will be dealt with by defining a *signal transform*. Then, it will be explained how this *signal transform* is used in the various guarded assignment transforms.

Recall that the static model representation of the conditional concurrent signal assignment statement is as given below (the guard signal is optional):

$$\begin{aligned}\text{cond-signal-assign} \;=\; \langle\, &\text{name, postponement, destination, (guard),}\\ &\text{pulse-rejection, cond-waveform-selections}\,\rangle.\end{aligned}$$

For each conditional concurrent signal assignment, ccsa, there is an equivalent process statement ps representation. Formally, this reduction is defined by the function CCSA-REDUCE(ccsa):

$$\begin{aligned}\text{CCSA-REDUCE(ccsa)} \;\equiv\; \langle\, &\text{ccsa.name,}\\ &\text{ccsa.postponement,}\\ &\text{(ccsa-transform, ccsa-wait-stmt),}\\ &\text{tr}\,\rangle\end{aligned}$$

where, in the simplest case (no guard expression), ccsa-transform is an if statement equivalent to signal-transform

$$\text{ccsa-transform} \;\equiv\; \text{signal-transform}$$

and ccsa-wait-stmt is equivalent to ccsa-unguarded-wait-stmt (defined later).

For the guarded assignment case, ccsa-transform is an if statement equivalent to ccsa-guarded-transform.

$$\text{ccsa-transform} \equiv \text{ccsa-guarded-transform}$$

and ccsa-wait-stmt is equivalent to ccsa-guarded-wait-stmt (defined later).

Signal Transform. The `when` clauses of the ccsa statement must be rewritten as an if statement. More formally, the signal-transform is realized as an if statement, if-stmt, of the form

$$\text{signal-transform} = \langle\ \text{ccsa.name, ccsa-if-choices}\ \rangle$$

where each member ccsa-if-choices_i of the sequence is of the form

$$\begin{aligned}\text{ccsa-if-choices}_i = \langle\ & \text{ccsa.cond-waveform}_i\text{.condition,} \\ & \text{ccsa-choice-body}_i\ \rangle\end{aligned}$$

and $\text{ccsa-choice-body}_i$ is a signal assignment statement of the form

$$\begin{aligned}\text{ccsa-choice-body}_i = \langle\ & \text{ccsa.name,} \\ & \text{ccsa.destination,} \\ & \text{ccsa.pulse-rejection,} \\ & \text{ccsa.cond-waveform}_i\text{.waveform}\ \rangle.\end{aligned}$$

Finally, the wait statement at the end of the process ordered statements, ccsa-wait-stmt, is defined as equivalent to ccsa-unguarded-wait-stmt where

$$\begin{aligned}\text{ccsa-unguarded-wait-stmt} = \langle\ & \text{ccsa.name,} \\ & \text{ccsa-sensitivities,} \\ & \texttt{true,} \\ & \texttt{infinity}\ \rangle\end{aligned}$$

with ccsa-sensitivities defined as the union of all signals listed in the waveforms.

Guarded Signal Transform. A guarded signal assignment transform is similar to that of the conditional concurrent signal assignment transform up to the point where ccsa-transform is defined. The definition of ccsa-transform may take two different forms of an if statement with the guard expression as the condition.

The guarded transform is applied to conditional signal assignment tuples with guard expressions and when the target of the concurrent signal assignment is a guarded target. A signal is a guarded target if

ccsa.destination.signal-kind = `register` | `bus`.

ccsa-guarded-transform is an if statement of the following form

ccsa-guarded-transform = ⟨ ccsa.name,
(⟨ ccsa.guard-expression, signal-transform ⟩) ⟩.

If a guard expression is present and the target of the concurrent signal assignment is not a guarded target expressed by the pre-condition

ccsa.destination.signal-kind = `discrete`,

then ccsa-guarded-transform is an if statement with an else clause consisting of a disconnection statement as shown below:

ccsa-guarded-transform = ⟨ ccsa.name,
(⟨ ccsa.guard-expression,
signal-transform ⟩,
⟨ `true`, disconnection-statement ⟩) ⟩.

To define the implicit time delay used in the implicit disconnection of drivers of a guarded signal, disconnection-delay was added to the signal tuple. This delay is then used in disconnection-statement (a signal assignment statement which assigns a value of null to it's target):

disconnection-statement = ⟨ ccsa.name, ccsa.destination,
ccsa.pulse-rejection,
⟨ `null`,
ccsa.destination.disconnection-delay ⟩ ⟩.

In this case, the wait statement ccsa-wait-stmt is defined to be equivalent to ccsa-guarded-wait-stmt where

ccsa-guarded-wait-stmt = ⟨ ccsa.name,
ccsa-sensitivities ∪ { ccsa.guard },
`true`,
`infinity` ⟩.

5.2.5 Selected Concurrent Signal Assignment Statements

The reduction of a selected concurrent assignment statement is similar to the conditional case, except that the selected statement is rewritten as a case statement.

The static model representation of the selected concurrent signal assignment statement is stated below (the guard signal is optional):

sel-signal-assign = ⟨ name, postponement, destination, (guard),
pulse-rejection, expression,
sel-waveform-selections ⟩.

The reduction to a reduced process statement of an unguarded selected concurrent signal assignment statement is similar to that of the conditional version. Formally, this reduction is defined by the function SCSA-REDUCE(ccsa):

SCSA-REDUCE(ccsa) ≡ ⟨ scsa.name,
scsa.postponement,
(scsa-guarded-transform, scsa-wait-stmt),
tr ⟩

where, in the simplest case (no guard expression) scsa-transform is an *if statement* equivalent to signal-transform. Formally,

scsa-transform ≡ signal-transform,

and scsa-wait-stmt is equivalent to scsa-unguarded-wait-stmt (defined later).

For the guarded assignment case, scsa-transform is an if statement equivalent to scsa-guarded-transform. Formally,

scsa-transform ≡ scsa-guarded-transform

and scsa-wait-stmt is equivalent to scsa-guarded-wait-stmt (defined later).

Signal Transform. A signal-transform in the selected concurrent assignment is realized as a case-stmt of the form

signal-transform = ⟨ scsa.name,
scsa.expression,
scsa-case-choices ⟩

where each member scsa-case-choices$_i$ of the sequence is of the form

scsa-if-choices$_i$ = ⟨ scsa.sel-waveform$_i$.choices,

scsa-choice-body$_i$ ⟩

and scsa-choice-body$_i$ is a signal assignment statement of the form

scsa-choice-body$_i$ = ⟨ scsa.name,
scsa.destination,
scsa.pulse-rejection,
scsa.sel-waveform$_i$.waveform ⟩.

Finally, the wait statement at the end of the process ordered statements, scsa-wait-stmt is defined as equivalent to scsa-unguarded-wait-stmt where

scsa-unguarded-wait-stmt = ⟨ scsa.name,
scsa-sensitivities,
`true`,
`infinity` ⟩

with scsa-sensitivities defined as the union of all signals listed in the waveforms.

Guarded Signal Transform. A guarded signal assignment transform is similar to that of the selected concurrent signal assignment up to the point where scsa-transform is defined. The definition of scsa-transform may take two different forms of an if statement with the guard expression as the condition.

The guarded transform is applied to conditional signal assignment tuples with guard expressions and when the target of the concurrent signal assignment is a guarded target. A signal is a guarded target if

scsa.destination.signal-kind = `register` | `bus`.

scsa-guarded-transform is an if statement of the form

scsa-guarded-transform = ⟨ scsa.name,
(⟨ scsa.guard-expression,
signal-transform ⟩) ⟩.

If a guard expression is present and the target of the concurrent signal assignment is not a guarded target, expressed by the pre-condition

scsa.destination.signal-kind = `discrete`,

then scsa-guarded-transform is an if statement with an else clause consisting of a disconnection statement

scsa-guarded-transform = ⟨ scsa.name,

(⟨ scsa.guard-expression,
signal-transform ⟩,
⟨ `true`, disconnection-statement ⟩) ⟩.

To define the implicit time delay used in the implicit disconnection of drivers of a guarded signal, `disconnection-delay` was added to the signal tuple. This delay is then used in disconnection-statement (a signal assignment statement which assigns a value of null to it's target):

disconnection-statement = ⟨ scsa.name, scsa.destination,
scsa.pulse-rejection,
⟨ `null`,
scsa.destination.disconnection-delay ⟩ ⟩.

In this case, the wait statement scsa-wait-stmt is defined to be equivalent to scsa-guarded-wait-stmt where

scsa-guarded-wait-stmt = ⟨ scsa.name,
scsa-sensitivities ∪ { scsa.guard },
`true`,
`infinity` ⟩.

5.3 THE REDUCED FORM

The previous sections dealt with defining reduction functions for some VHDL statements. This section steps up a level of abstraction and looks at how the reduction functions can be applied to a Static Model representation of a VHDL description to its minimal form.

5.3.1 Application of the Reduction Algebra

Let vhdl-world be the Static Model representation of a VHDL description. As stated in Chapter 3, vhdl-world is defined as

vhdl-world = ⟨ Environment, Concurrent-Statements ⟩.

Refer to Chapter 3 for the definition of Concurrent-Statements and Environment. We then define reduced-world which is the reduced form of vhdl-world as

reduced-world = ⟨ Reduced-Environment, Reduced-Process-Stmts ⟩.

Further, let $\mathcal{F}$ denote the function that reduces vhdl-world to reduced-world. Formally, $\mathcal{F}$ is defined as

$\mathcal{F}$: Environment $\times$ Concurrent-Statements $\rightarrow$
Reduced-Environment $\times$ Reduced-Process-Stmts

$\mathcal{F}$(vhdl-world) = reduced-world.

Informally, $\mathcal{F}$ applies the reduction function SA_REDUCE to all the signal assignment statements in the subprograms of the VHDL description and then applies the appropriate reductions to all the concurrent statements converting them to process statements. Finally, it applies SA_REDUCE to the signal assignment statements in the resulting process statements. A formal definition of $\mathcal{F}$ is too cumbersome and lengthy to be presented here. The next chapter, however, presents a technique to embed this reduction function in an automated proof checker and shows how this embedding can be used to prove certain properties of $\mathcal{F}$.

6 COMPLETENESS OF THE REDUCED FORM

The reduction algebra defined in the previous chapter can be thought of as taking in a stream of characters and spitting out a transformed stream of characters. It is important to establish that this algebra can successfully reduce the input stream to a reduced *normal* form that cannot be further reduced by the algebra (*irreducibility* property). It is also useful to establish that if the reduction algebra can take two different paths in reducing the input stream, both paths lead to the same final reduced form (*completeness* property). Intuitively, our reduction algebra seems to have both of the above properties since all reduction functions except SA-REDUCE output a process statement on which only SA-REDUCE can be applied. Once SA-REDUCE is applied, there are no further reductions. The following sections prove the above properties formally through an embedding of the reduction algebra in the automated proof checker PVS. The following sections presents a brief introduction to PVS and the embedding of the algebra in PVS.

6.1 A BRIEF OVERVIEW OF PVS

The **P**rototype **V**erification **S**ystem (PVS) is a higher ordered logical system [44] supported by an automatic theorem proving environment [33]. PVS supports construction and verification of parameterized, algebraic specifications for systems. PVS was selected for this activity over systems such as Larch [24],

Z [47], and CSP [25] due to its rich type-system and quality verification support. PVS provides sophisticated type checking and verification using a sequent calculus based proof environment. It is impossible to completely describe PVS in this work. The interested reader is encouraged to read the language and prover reference manuals [44, 33].

6.1.1 The PVS Language

To follow the specifications used in this work, only a few PVS language features need be understood. Specifically, the definition of types, the definition of axioms, and the definition of lemmas and theorems. For the purposes of this book, a PVS specification is minimally a collection of type definitions, axioms, lemmas, and theorems.

PVS types are used to define types and function signatures. A PVS type is declared with the TYPE or a TYPE+ keyword. If only the type name is specified, PVS assumes the type to be uninterpreted. This implies that all characteristics of the type will be defined using axioms instead of identifying specific elements of the type. Types can also be specified as subtypes of other types using a form of set comprehension.

```
% Declares type queue with at least on element.
queue: TYPE+

% Declares a subtype of integer containing values greater than 0.
posint: TYPE = {x: integer | x > 0}

% Declares a subtype of integer containing values satisfying R(x).
% it is a shorthand for rint: TYPE = {x: integer | R(x)}
rint: (R)
```

Function and predicate signatures are defined using a standard mapping notation. Functions and predicates may accept arbitrarily many input parameters, but must produce a single output parameter.

```
% Declares a function taking two arguments of type integer and
% returning an argument of type posint
example: [integer, integer -> posint]
```

Definitions of formulas such as axioms, lemmas, and theorems have the form <name>: <type><expression> where (i) <name> is the name of the item; (ii) <type> is its type (AXIOM, LEMMA, THEOREM, etc); and (iii) <expression> is a boolean valued logical statement. Semantically, lemmas and theorems are identical. Both must be proven before use or completion of the verification activity. Axioms differ in that they are assumed true and their verification is not required.

A **LEMMA** states a obligation that must be proven before the verification is complete. The following lemma expresses the commutativity of the function **add**:

```
% Declares a LEMMA
lemma1: LEMMA (forall (x:integer,y:integer): add(x,y) = add(y,x)
```

Note that the function **add** must have its signature, domain types and range type defined prior to stating this lemma. Replacing the **LEMMA** keyword with **AXIOM** results in a statement that does not need to be proven. Replacing the **LEMMA** keyword with **THEOREM** does nothing semantically, but does indicate to the user that this is a higher level proof goal.

6.1.2 The PVS Prover

The correctness of the specification can be evaluated using the proof checker that accompanies the PVS specification language. The theorem prover uses **sequent calculus** to manipulate and prove lemmas and theorems. The sequent calculus separates logical expressions into antecedents and consequents:

$$\Gamma_0 \wedge \Gamma_1 \ldots \Gamma_n \vdash \Delta_1 \vee \Delta_2 \ldots \Delta_m$$

As is expected, antecedents are assumed true during the proof process and consequents represent proof goals. The prover's objective is to rewrite some Δ_k to *true* or some Γ_j to *false*. Traditional boolean simplification, case splitting, and rewriting are among the most common techniques employed. Axioms, lemmas and theorems can be added as antecedents during the proof process. Additionally, PVS provides a number of existing theories for common mathematical properties and structures. Most important for this work are PVS theories for strong, weak induction, and ordering relationships.

The prover begins verification by type-checking a specification. PVS uses a powerful theorem proving based type checking methodology. This looks at more than simple type compatibility by examining the semantics of a type in the context of its use. *Type Correctness Conditions* (TCCs) are generated and must be proven before the primary verification activity begins. This type checking process saves vast amounts of time during the proof process and is a principle reason for our selection of PVS.

Following type-checking, the user selects theorems to be proven and guides PVS through the proof process. Although PVS is not totally automated, it provides a number of abstract proof strategies. In addition, users may write their own proof strategies using the PVS proof scripting language.

6.2 THE SPECIFICATION OF THE REDUCTION ALGEBRA IN PVS

The separate elements of the reduction algebra appear as axioms in PVS. For example, the reduction function in CA-REDUCE can be embedded in PVS as shown below.

```
ca_reduce_ax:  AXIOM
  ca_reduce(ca(name, postponement, condition, message,
               severity)) =
  ps(name, postponement, NULL,
    append(wa(name, signals_in_expr(condition), MYTRUE,
              INFINITY),
    append(as(name, condition, message, severity), empty_seq)),
    TRNULL)
```

Other reduction functions are embedded in a similar manner. For the conditional signal assignment case and the selected signal assignment case, we represent the reduction with and without the guard expression separately. For example, the reduction function for the unguarded selected signal assignment (USCSA-REDUCE) is embedded as shown below.

```
uscsa_reduce_ax:  AXIOM   uscsa_reduce(uscsa(name, postponement, destination,
   pulse_rejection, expression, sel_waveform_selections))
    =
   ps(name, postponement, NULL,
   append(wa(name, signals_in_sel(sel_waveform_selections), MYTRUE, INFINITY),
   ssa_reduce(uscsa_transform((uscsa(name, postponement, destination,
    pulse_rejection, expression, sel_waveform_selections))))),
    tr_csa(sel_waveform_selections))
```

6.3 SIGNAL ASSIGNMENT REDUCTION

The reduction function SA_REDUCE is different from the other reductions in that it acts on a sequential signal assignment statement while the other reduction functions act on concurrent statements. To simplify our proofs we assume that the reductions on the concurrent statements also apply a reduction SSA_REDUCE to the sequential statements of the resulting process statement which, in turn, merely applies SA_REDUCE to the signal assignments in the process. Further, we assume that there are no subprograms in the VHDL description and hence ignore the application of SA_REDUCE to a signal assignment statement in a subprogram. The following sections present proofs of completeness and irreducibility of the reduction algebra with the above stated assumptions.

6.4 COMPLETENESS

We are interested in knowing whether the repeated application of the algebra reduces a VHDL description v to an irreducible form v' after which, no further reductions are possible. We also wish to determine if the irreducible form v' (if any) is unique for a given v (the *completeness* property). Bearing in mind, the modifications made to the reduction algebra in the previous sections, let $\mathcal{R}$ name the set of all reductions in the reduction algebra. Formally,

R = { CAS_REDUCE, CPC_REDUCE, PS_REDUCE, UCCSA_REDUCE,
GCCSA_REDUCE, USCSA_REDUCE, GSCSA_REDUCE }.

Knuth and Bendix [29] define the completeness property for a set of reductions R. Let R be represented as the set $\{\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n\}$. If a word v has a substring α_k for some k, $1 \leq k \leq n$, then the application of R to v results in the replacement of α_k by β_n. If $v = \theta\alpha_k\gamma$ and $v' = \theta\beta_k\gamma$, the relation between v and v' can be represented as

$$v \longrightarrow v' (R).$$

A string v' is said to be irreducible with respect to R if there exists no v'' such that

$$v' \longrightarrow v'' (R).$$

If R is such that the length of β_k is less than the length of α_k for all k, $1 \leq k \leq n$, then we are assured that R reduces v to an irreducible form v'. Unfortunately, $\mathcal{R}$ does not have this property. In fact, the application of $\mathcal{R}$ to a VHDL code may increase the size of the code. However, we can show that the application of $\mathcal{R}$ does indeed reduce a given VHDL description to a form which cannot be further reduced.

Theorem 6.1 *$\mathcal{R}$ always reduces a VHDL description v to an irreducible form v'.*

We delay the proof of the above theorem to the next section. Now the completeness property can be defined. A set of reductions R is complete if for any v' and v'' which are two irreducible forms of v with respect to R, $v' = v''$. Let $v \rightarrow *v'$ denote that there exist $v_0, v_1, \ldots, v_n$ such that $v_0 = v, v_j \rightarrow v_{j+1}$ for $0 \leq j < n$ and $v_n = v'$. Knuth and Bendix show that R is complete if it satisfies the following lattice condition.

Theorem 6.2 *A set of reductions R is complete iff the following condition is satisfied:*
if $v \rightarrow v'$ and $v \rightarrow v''$, there exists a word γ such that $v' \rightarrow *\gamma$ and $v'' \rightarrow *\gamma$.

Consider a VHDL code $v = s_1 s_2 \cdots s_n$ where $s_1, s_2, \ldots, s_n$ are VHDL concurrent statements. Concurrent statements can be executed in any order. Hence different permutations of $s_1 s_2 \cdots s_n$ are equivalent. However, without loss of generality, we assume that the reductions in $\mathcal{R}$ maintain the ordering of $s_1, s_2, \ldots, s_n$. Let $v = \theta_1 s_k \gamma_1$ where $\theta_1 = s_1 s_2 \cdots s_{k-1}$ and $\gamma_1 = s_{k+1} s_{k+2} \cdots s_n$ where s_k is some concurrent statement. Also let $v' = \theta_1 p_k \gamma_1$ where p_k is the process statement equivalent to s_k. Then we can state

$$v \longrightarrow v' (\mathcal{R}).$$

Alternately, let $v = \theta_2 s_m \gamma_2$ where $\theta_2 = s_1 s_2 \cdots s_{m-1}$ and $\gamma_2 = s_{m+1} s_{m+2} \cdots s_n$ and s_m is some concurrent statement and $m \neq n$. Let $v'' = \theta_2 p_m \gamma_2$ where p_m is the process statement equivalent to s_m. Then we can state

$$v \longrightarrow v'' (\mathcal{R}).$$

We need to show that there exists a λ such that $v' \longrightarrow *\lambda$ and $v'' \longrightarrow *\lambda$. Note that each reduction function in $\mathcal{R}$ is defined such that *it applies all possible reductions to a given concurrent statement in one step.* Consequently, if a concurrent statement is reduced once, no further reductions on it are possible. Therefore, irrespective of the order in which the concurrent statements are reduced, once all of them are reduced, the same irreducible form $\lambda = p_1 p_2 \cdots p_n$ is obtained. Thus, the lattice condition is satisfied. We can now state the following theorem.

Theorem 6.3 *$\mathcal{R}$ is complete.*

6.5 IRREDUCIBILITY

In this section we prove Theorem 6.1 which states that a repeated application of $\mathcal{R}$ to a given VHDL description v ultimately reduces v to a form v' where v' is irreducible.

Automating the proof of the theorem in its current form is difficult. However, the definition of $\mathcal{R}$ can be modified to simplify the proof. Consider a reduction function F that applies all the reductions stated in Section 6.2 to *all* the concurrent statements in the given description in one step. Furthermore, define F such that, if applied on a process statement with no sensitivity list and no signal assignment statements with multiple waveforms, returns the same process statement (unity reduction). We can now redefine $\mathcal{R}$ as $\mathcal{R}' = \{F\}$. Since unity reductions are allowed, the proof goal can be modified. Instead of proving that F reduces v to a form v' after which no reductions are possible, we can show that F reduces v to v' such that only the unity reduction can be applied to v'. More precisely, we show that $F(F(v)) = F(v)$. Functions having this property are said to be *idempotent.*

Lemma 6.5.1 *For a given VHDL description v, $F(F(v)) = F(v)$.*

Table 6.1. The observers for the concurrent statements

Concurrent Statement	Predicate
Concurrent Assertion Statement	ca?
Process Statement	ps?
Concurrent Procedure Call	cpc?
Unguarded Conditional Concurrent Signal Assignment	uccsa?
Guarded Conditional Concurrent Signal Assignment	gccsa?
Unguarded Selected Concurrent Signal Assignment	uscsa?
Guarded Selected Concurrent Signal Assignment	gscsa?

Proof:. This proof is carried out in the PVS theorem prover. The above goal is encoded in the PVS specification language as the theorem

```
goal:  THEOREM F(F(v)) = F(v)
```

where v is a set of concurrent statements (the VHDL description). The proof can be carried out by induction on the size of the VHDL description v. The base and induction steps are presented below.

Base case:. Given that $v = \emptyset$, the theorem appears as `goal.1` as shown below. Using the axiom `goal_empty`, which states that `F(∅) = ∅`, we can clearly see that `F(F(∅)) = F(∅)`. The same is achieved in PVS by rewriting the step `goal.1` using the axiom `goal_empty`.

```
goal_empty:  AXIOM empty?(v) IMPLIES F(v) = v

goal.1 :
{-1}    empty?(x!1)
  |-------
{1}     F(F(x!1)) = F(x!1)
```

Inductive case:. Assuming that `F(F(x!1)) = F(x!1)` for an arbitrary `x!1` and by showing that `F(F(add(csx!1, x!1))) = F(add(csx!1, x!1))` holds for an arbitrary concurrent statement `csx!1`, we can prove the inductive case. This appears as `goal.2` in the PVS theorem prover.

```
goal.2 :
{-1}    F(F(x!1)) = F(x!1)
  |-------
{1}     F(F(add(csx!1, x!1))) = F(add(csx!1, x!1))
```

Table 6.1 lists all the possible concurrent statements and the predicates that identify them. Exactly one of the predicates is true for each concurrent statement. The axiom `Vhdl_inclusive` states that *at least* one of these predicates is true for each concurrent statement. There is a set of axioms that states the fact that *at most* one of these predicates is true. For example, the axiom `Vhdl_disjoint_ca` states that a concurrent statement is a concurrent assertion statement if and only if it is none of the others.

```
Vhdl_inclusive:  AXIOM
      (ca?(csx) OR ps?(csx) OR cpc?(csx) OR uccsa?(csx)
           OR gccsa?(csx) OR uscsa?(csx) OR gscsa?(csx))
Vhdl_disjoint_ca:  AXIOM
      (ca?(csx) IFF NOT (ps?(csx) OR cpc?(csx) OR uccsa?(csx)
           OR gccsa?(csx) OR uscsa?(csx) OR gscsa?(csx)))
```

Using this information, `goal.2` can be proved by cases. Using the axiom `Vhdl_inclusive` on `goal.2` and splitting conjunctions, we derive seven subgoals. The first of these subgoals (for the case where `csx!1` is a concurrent assertion statement) is shown below.

```
goal.2.1 :
{-1}  ca?(csx!1)
[-2]  F(F(x!1)) = F(x!1)
  |-------
[1]   F(F(add(csx!1, x!1))) = F(add(csx!1, x!1))
```

Informally, the above sequent states that we need to show that the inductive case is true for a concurrent assertion statement. Using the axiom stated above (`Vhdl_disjoint_ca`), applying propositional simplification and re-writing using the definition of `F`, the sequent `goal 2.1` simplifies to

```
goal.2.1 :
[-1]  ca?(csx!1)
[-2]  F(F(x!1)) = F(x!1)
  |-------
[1]   F(add(ca_reduce(csx!1), F(x!1))) = add(ca_reduce(csx!1), F(x!1))
```

In order to reduce the above sequent, we use the fact that `ca_reduce` returns a process statement. Therefore, the sequent must be converted to a form to which `ca_reduce` can be applied. To achieve this, we use the fact that `ca` is the only constructor for concurrent assertion statements. Applying the stated conversion and rewriting using the axiom `ca_reduce` (which reduces the `csx!1` to a process statement), the sequent reduces to

```
goal.2.1 :
[-1]    F(F(x!1)) = F(x!1)
  |------
[1]     F(add(ps(name!1, postponement!1, NULL,
              append(wa(name!1, signals_in_expr(condition!1),
                                MYTRUE, INFINITY),
                     append(as(name!1, condition!1,
                               message!1, severity!1),
                              empty_seq)), TRNULL), F(x!1)))
              =
          add(ps(name!1, postponement!1, NULL, append(wa(...)),
              TRNULL), F(x!1))
```

We now need to know how to apply the function F on a set containing a process statement. To simplify the application of the function F we first state that if a concurrent statement is a process statement it is not any of the six other types of concurrent statement. To achieve this, we use the fact that ps is the only constructor for process statements. Applying the stated transformations, propositional simplification and rewriting using the definition of F, the sequent reduces to

```
goal.2.1 :
[-1] F(F(x!1)) = F(x!1)
  |------
[1]  add(ps_reduce(ps(name!1, postponement!1, NULL, append(wa(...)),
         TRNULL)), F(F(x!1)))
     =
     add(ps(name!1, postponement!1, NULL, append(wa(...)), TRNULL),
         F(x!1))
```

Replacing F(F(x!1)) with F(x!1) (using the induction hypothesis) and rewriting using definition of ps_reduce (which in turn applies ssa_reduce to the ordered statements of the process statement), the sequent simplifies to

```
goal.2.1 :
  |------
{1}   add(ps(name!1, postponement!1, NULL, ssa_reduce(append(wa(...))),
           TRNULL), F(x!1))
      =
      add(ps(name!1, postponement!1, NULL, append(wa(...)), TRNULL),
          F(x!1))
```

The function ssa_reduce, does not have any effect on a wait statement or a sequential assert statement, since it only reduces multiple waveforms of a signal assignment statement. Using this, the sequent reduces to

```
goal.2.1 :
  |-------
{1}   add(ps(name!1, postponement!1, NULL,
              append(wa(...), append(as(...), ssa_reduce(empty_seq))),
              TRNULL), F(x!1))
      =
      add(ps(name!1, postponement!1, NULL,
           append(wa(...), append(as(...), empty_seq)), TRNULL), F(x!1))
```

Rewriting using the axiom `ssa_reduce(empty_seq) = empty_seq`, the sequent `goal.2.1` becomes trivially true. This completes the inductive step for the case of the assertion statement. Similar proofs are present for the other six cases. We have not discussed the other six cases here for brevity.

6.6 CONCLUSION

This chapter demonstrated a useful application of an automated proof checking environment (PVS) to reason about a formal representation of VHDL. PVS will be used again in later chapters to reason about equivalence of VHDL programs. So far, we have defined a static representation of VHDL and a reduced form of that representation. We also provided the algebra for obtaining the reduced form. The reduced form is important for optimizations to VHDL CAD tools. In particular, a VHDL compiler may apply the reduction algebra to the input VHDL program so that the redundant static structures may be eliminated and simulations may run more efficiently.

We now turn our attention from the static aspects of VHDL to the dynamic aspects. In particular, we will define the dynamic semantics of a large subset of VHDL based on the reduced form of the Static Model. For example, the only concurrent statement for which the dynamic semnatics is defined is the process statement since other concurrent statements are not present in the reduced form.

7 INTERVAL TEMPORAL LOGIC

The dynamic semantics of VHDL presented in this book is based on Allen's Interval Temporal Logic [4]. This chapter presents some of the key concepts of this logic. Interval Logic is useful in capturing the timing information contained in a VHDL description. The logic organizes a universe by time intervals, relations between time intervals, and by binding actions (or assertions) to time intervals. VHDL behavior can be defined in terms of time intervals and a set of actions that are performed in these time intervals.

Interval Temporal Logic is axiomatized in terms of the single relation meets. A time interval t_1 is said to meet t_2 if t_1 is before t_2 such that there is no time interval separating them and if they do not overlap in any way (Figure 7.1). Furthermore, the symbol "+" is used to denote concatenation of time intervals. For example, in Figure 7.1, $t_1 + t_2 = t_3$.

There are six basic relations other than meets that are available in the logic.[1] Informally, these relations can be interpreted as follows:

equals(t_1, t_2) : t_1 and t_2 represent the same time interval,

before(t_1, t_2) : t_1 finishes before t_2 begins and there is an interval t_3 that is between t_1 and t_2,

overlaps(t_1, t_2) : t_1 begins before t_2 begins, t_2 begins before t_1 finishes, and t_1 finishes before t_2 finishes,

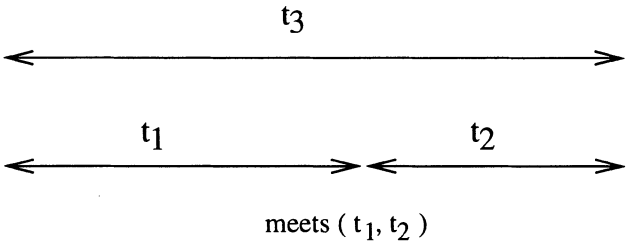


Figure 7.1. The relation meets

during(t_1, t_2) : t_1 is fully contained within t_2,

starts(t_1, t_2) : t_1 and t_2 begin together, but t_1 finishes before t_2 finishes, and

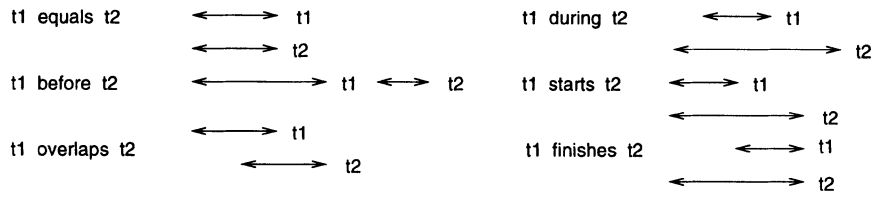
finishes(t_1, t_2) : t_1 starts after t_2 starts, but they finish together.

These relations are formally defined in terms of the relation meets in Figure 7.2. In the figure, the notation a $\diamond$ b has been used to represents meets(a, b). Although the notions of time points and time moments are available in the logic, they are not required in this work. Intervals are not closed and they cannot be empty. The critical issue is that for any given interval t, there exists another interval for each predicate of the logic such that the predicate holds. That is, given an interval t, there is an interval that meets t, that equals t, that starts with t, and so on. Additionally, four other predicates that will be used in this work are defined below:

$$
\begin{array}{lcl}
\mathsf{in}(t_1, t_2) & \Leftrightarrow & \mathsf{during}(t_1,t_2) \vee \mathsf{starts}(t_1, t_2) \vee \mathsf{finishes}(t_1, t_2), \\
\mathsf{begins}(t_1, t_2) & \Leftrightarrow & \mathsf{starts}(t_1, t_2) \vee \mathsf{starts}(t_2, t_1) \vee \mathsf{equals}(t_2, t_1), \\
\mathsf{ends}(t_1, t_2) & \Leftrightarrow & \mathsf{finishes}(t_1, t_2) \vee \mathsf{finishes}(t_2, t_1) \vee \mathsf{equals}(t_2, t_1), \\
\mathsf{endsafter}(t_1, t_2) & \Leftrightarrow & \mathsf{before}(t_2, t_1) \vee \mathsf{overlaps}(t_2, t_1) \vee \\
 & & \mathsf{meets}(t_2, t_1) \vee \mathsf{starts}(t_2, t_1) \vee \mathsf{during}(t_2, t_1).
\end{array}
$$

Informally, the predicate in(t_1, t_2) is `true` if t_1 lies within the boundaries of t_2. The predicate begins(t_1, t_2) is `true` if t_1 and t_2 start at the same point. The predicate ends(t_1, t_2) is true if t_1 and t_2 end at the same point. The predicate endsafter(t_1, t_2) is true if t_1 ends after t_2 ends.

Two other components of Allen's logic relate to *properties* and *events*. A *property* is a time dependent predicate defined on some aspect of the universe under consideration, or a particular state of that universe (*e.g.*, a static value of a program variable). Temporal logic maintains the law of excluded middle, *i.e.*, a property p is either true or false and there is no interval which is between the interval when p is true and the interval when p is false. An *event* is something that may cause a state change or the value of a particular predicate to change



$$
\begin{aligned}
equals(b,a) &= \forall m \bullet m \diamond a \Rightarrow m \diamond b \wedge \forall n \bullet a \diamond n \Rightarrow b \diamond n \\
before(b,a) &= \exists k \bullet b \diamond k \wedge k \diamond a \\
overlaps(b,a) &= \forall m \bullet m \diamond b \Rightarrow (\exists k \bullet m \diamond k \wedge k \diamond a) \wedge \\
&\quad \forall n \bullet a \diamond n \Rightarrow (\exists k \bullet b \diamond k \wedge k \diamond n) \\
during(b,a) &= \forall m \bullet m \diamond a \Rightarrow (\exists k \bullet m \diamond k \wedge k \diamond b) \wedge \\
&\quad \forall n \bullet a \diamond n \Rightarrow (\exists l \bullet b \diamond l \wedge l \diamond n) \\
starts(b,a) &= \forall m \bullet m \diamond a \Rightarrow m \diamond b \wedge \forall n \bullet a \diamond n \Rightarrow (\exists k \bullet b \diamond k \wedge k \diamond n) \\
finishes(b,a) &= \forall m \bullet m \diamond a \Rightarrow (\exists k \bullet m \diamond k \wedge k \diamond b) \wedge \forall n \bullet a \diamond n \Rightarrow b \diamond n
\end{aligned}
$$

Figure 7.2. Relationships of time intervals

(*e.g.*, incrementing the value of a program variable). Corresponding to these two notions, Allen defines two predicates: HOLDS and OCCUR.

The predicate HOLDS takes a predicate p, and a time interval t_1, and is true if the predicate p is true throughout t_1. One of the most important properties of the HOLDS predicate is that if it is true for some predicate p and time interval t_1 then it is also true for every time interval t_2 contained in t_1. Formally, this can be stated as

$$\mathsf{HOLDS}(p, t_1) \Leftrightarrow \forall\, t_2 : \mathsf{in}(t_1, t_2) \rightarrow \mathsf{HOLDS}(p, t_2).$$

The predicate OCCUR takes an event e, and a time interval, t_1, and is true if the event e occurs in t_1 and there is no subinterval t_2 of t_1 in which e occurs. Thus, the predicate OCCUR has the property that if it is true for an event e in time t_1, then it is not true for any subinterval of t_1. Formally, this property of the OCCUR predicate is defined by the axiom

$$(\mathsf{OCCUR}(e, t_1) \wedge \mathsf{in}(t_1, t_2)) \rightarrow \neg\, \mathsf{OCCUR}(e, t_2).$$

The notion of time intervals is closely related to the notion of occurrences of events. In the context of this work, an event could be an execution of a signal assignment statement which occurs over a finite interval of time. Events can occur before, overlap or occur during another event just as these relations can

hold between time intervals. Further, events can be decomposed into smaller events just as time intervals are decomposable. For example, the execution of a signal assignment statement can be decomposed into sub-events such as evaluating the expression on the right hand side of the assignment, storing the result in a register, accessing the memory location of the destination and finally copying the value of the register into the memory location.

Notes

1. These relations can be defined in terms of **meets** alone.

8 THE DYNAMIC MODEL

This chapter presents the dynamic semantics of a VHDL description as a declarative definition of the *state space* evaluated by the description. The state space consists of the values of declared signals and ports in the description.[1] Since the net result of evaluating a VHDL description is reflected in the changes in values of its signals, the 'waveforms' of signals represent a semantics of the description. Thus, the ultimate goal in the definition of the semantics is to specify a set of time intervals spanning the entire simulation of the given VHDL description and to define the values of all signals in the description in those time intervals. The model presented herein is called the Dynamic Model and is based on the normal form of the Static Model as defined in Chapter 5.

Since time plays an important role, a set of reference time intervals is defined against which other intervals can be constructed. This temporal reference frame is called the *time keeper* (or simply TK) and is represented as a sequence of adjacent time intervals. Formally, TK is defined by the equation

$$\mathsf{TK} = (\mathsf{TK(i)} \mid \mathsf{i} \in N \wedge \mathsf{i} \geq 0 \wedge \mathsf{meets(TK(i),\ TK(i+1))}),$$

which states that TK is a sequence of time intervals that meet each other and are hence adjacent to each other. In the remainder of this book, the notation TK(i) will be used to denote the i^{th} element of TK (refer to Figure 8.1). It is important to note that the time keeper plays a dual role. It serves as a reference for defining the time periods of evaluating statements (the real time) as well

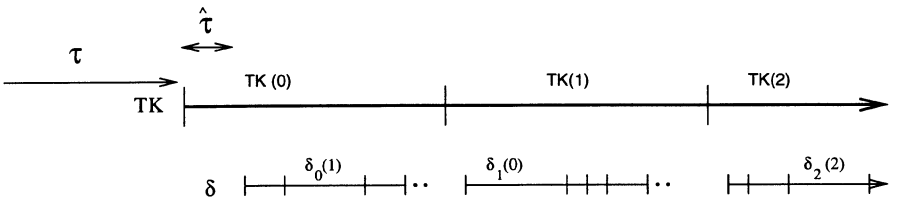


Figure 8.1. Delta intervals

as provides the notion of passage of *virtual*[2] time (or simulation time) against which values of signals are defined. Crossing the boundaries of the time keeper intervals corresponds to moving ahead in simulation time.

In a VHDL simulation, process statements are evaluated in a *simulation cycle.* There may be several simulation cycles before simulation time advances. These cycles, in which simulation time does not advance, are called *delta cycles* in the LRM. In the Dynamic Model, a time interval in which a simulation cycle (or delta cycle) may occur is referred to as a *delta interval.* Delta intervals are wholly contained within TK(i) intervals since simulation time cannot advance within these intervals. In general, there is no upper bound on the number of delta cycles that can occur before simulation time advances. Consequently, a sequence of an infinite number of delta intervals are associated with every TK(i) (Figure 8.1). This is formally stated as

$$\forall\, i,j \in N \bullet \exists\, \delta \bullet \delta = (\delta_i(j) |\ i,j \geq 0 \wedge \mathsf{during}(\delta_i(j), \mathsf{TK}(i)) \wedge \mathsf{meets}(\delta_i(j), \delta_i(j+1)))$$

where δ_i is the j^{th} interval in TK(i). The Dynamic Model does not differentiate between a simulation cycle and a delta cycle. It only provides a set of time intervals (the delta intervals) in which VHDL statements may be evaluated. The notion of passage of time is captured by TK(i) intervals. There is no simulation time difference between two delta intervals that lie in the same TK(i) interval. However, the simulation time difference between the delta intervals $\delta_i(m)$ and $\delta_j(n)$ is n - m. It is important to note that delta cycles and delta intervals are not interchangeable concepts. There are always an infinite number of delta intervals that lie within the boundaries of a single TK(i) but only some of them may contain the evaluation intervals of VHDL statements (corresponding to the execution of a simulation cycle or delta cycle in VHDL).

The construction of delta intervals does not mean that an operational definition of the VHDL simulation cycle is being provided. The Dynamic Model uses delta intervals as *reference* intervals to define the waveforms of signals in a given VHDL description. No assertion is made that simulation of a VHDL description *must* involve a step-by-step execution of simulation cycles and delta cycles. The Dynamic Model only asserts the *existence* of delta intervals (and some other intervals that will be introduced in Section 8.3) based on which, the

waveforms of signals can be defined. These waveforms represent a behavior of a circuit and can be used to establish equivalences. Indeed, we will see in the proof of process folding (Section 9.2) that an equivalence between two VHDL descriptions can be established by merely constructing waveforms of signals without resorting to a step-by-step simulation of the VHDL descriptions.

8.1 METHODOLOGY

As mentioned earlier, the waveforms of signals in a VHDL description represent a semantics of the description. The goal of the Dynamic Model is to define these waveforms. A waveform $\mathcal{W}(\mathsf{s})$ for a signal s is represented as

$$\mathcal{W}(\mathsf{s}) = \{ \langle \mathsf{v}_1, \mathsf{t}_1 \rangle, \langle \mathsf{v}_2, \mathsf{t}_2 \rangle, \langle \mathsf{v}_3, \mathsf{t}_3 \rangle, \ldots \},$$

where v_1 is the value of s in time interval t_1, v_2 is the value of s in time interval t_2, and so on. Dealing with waveforms in this manner allows us to reason about the equivalence of VHDL programs based on the *observable* behavior of these programs. Two VHDL programs are equivalent if the waveforms of the signals of the two programs are identical.

A significant advantage of this approach is that equivalence can be established with respect to a selected set of signals. For example, it is often useful to view the behavior of a VHDL description only from the external ports of the description. Two descriptions that have the same set of external ports can be thought of as being equivalent if the waveforms of their external ports are identical. The waveforms of signals that represent internal wires of a circuit need not be considered.

In order to construct the waveforms of signals of a given VHDL description, we first need to construct some intermediate sets of time intervals. These sets are:

1. Set of delta intervals.
2. Set of time intervals in which statements of the given description may be evaluated (Section 8.2).
3. Sets that store the value and time information related to the assignments on the signals in the description (Section 8.3).

The time intervals in which VHDL statements may be evaluated are determined by establishing *constraints* that these intervals must satisfy. The constraint on the evaluation interval of a VHDL statement s in a time interval t is defined in terms of a predicate $\mathcal{E}(\mathsf{s}, \mathsf{t})$ and a set $\sigma(\mathsf{s}, \mathsf{t})$ as explained below.

The predicate $\mathcal{E}(\mathsf{s}, \mathsf{t})$ is defined to be `true` if t is a time interval in which the evaluation of s is *feasible*. For example, if t lies wholly within a delta interval, it is feasible for a *signal assignment* to be evaluated in t. A *signal assignment*

```
p1: Process
    begin
s1:     A <= '0';
s2:     if ( COND ) then
          s22: C <= '1';
        end if;
s3:     wait on A;
    end process;
```

Figure 8.2. A process statement

cannot be evaluated in an interval that crosses delta interval boundaries. Therefore, the predicate $\mathcal{E}$(s, t) is `true` for a *signal assignment* s if during(t, δ_i(j)) holds for some i and j. For a wait statement wa, $\mathcal{E}$(wa, t) is defined to be `true` if t begins in some delta interval and ends in the delta interval in which the conditions on which wa is waiting are satisfied (see Section 8.2.4). Finally, for an *if* statement or a *loop* statement, the predicate $\mathcal{E}$ is defined recursively in terms of the definition of $\mathcal{E}$ of the statements it contains. Consider the example in Figure 8.2. For simplicity, let the condition COND of statement s2 be `true`. Then the evaluation of s2 in some time interval t_2 involves the evaluation of COND and the statement s22 in time intervals t_{21} and t_{22} respectively where $t_2 = t_{21}+t_{22}$. We say that it it is feasible for s2 to be evaluated in t_2 if it is feasible for COND to be evaluated in t_{21} and s22 to be evaluated in t_{22}. Formally, this is stated as

$$\mathcal{E}(\mathsf{s2}, t_2) \Leftrightarrow \mathcal{E}(\mathsf{COND}, t_{21}) \wedge \mathcal{E}(\mathsf{s22}, t_{22}).$$

Further, let t, t_1 and t_3 be intervals such that $t = t_1+t_2+t_3$. A single evaluation of the *process* p1 is feasible in t if the evaluation of s1 is feasible in t_1, s2 is feasible in t_2 and s3 is feasible in t_3. Formally,

$$\mathcal{E}(\mathsf{p1}, t) \Leftrightarrow \mathcal{E}(\mathsf{s1}, t_1) \wedge \mathcal{E}(\mathsf{s2}, t_2) \wedge \mathcal{E}(\mathsf{s3}, t_3).$$

A set σ(s, t) is also defined which stores the statements that are evaluated as a result of s and the time intervals in which they are evaluated. More precisely, σ(s, t) is a set of 2-tuples $\langle s_i, t_i \rangle$ where s_i is a statement or expression evaluated as the result of the evaluation of s and t_i is the time interval in which s_i is evaluated. In the above example, the evaluation of s2 in t_2 involved the evaluation of COND in t_{21} and s22 in t_{22}. Hence $\sigma(\mathsf{s2}, t_2) = \{ \langle \mathsf{COND}, t_{21} \rangle, \langle \mathsf{s22}, t_{22} \rangle \}$. Finally, the evaluation set for a single evaluation of p1 is equal to $\{ \langle \mathsf{s1}, t_1 \rangle \} \cup \sigma(\mathsf{s2}, t_2) \cup \{ \langle \mathsf{s3}, t_3 \rangle \}$.

To summarize, if $\mathcal{S}$ is a sequence of statements where $\mathcal{S} = s_0\ s_1 \cdots s_n$, the definition of $\mathcal{E}$ and σ for $\mathcal{S}$ in a time interval t is given as

$$\begin{aligned}\mathcal{E}(\mathcal{S}, t) \Leftrightarrow \exists\ & t_0, t_1, \ldots, t_n \bullet \\ & t_0 + t_1 \cdots + t_n = t \wedge \\ & \mathcal{E}(s_0, t_0) \wedge \mathcal{E}(s_1, t_1) \ldots \wedge \mathcal{E}(s_n, t_n) \wedge \\ & \sigma(\mathcal{S}, t) = \sigma(s_0, t_0) \cup \sigma(s_1, t_1) \cdots \cup \sigma(s_n, t_n).\end{aligned}$$

Using the definitions of $\mathcal{E}$ and σ for the statements of a VHDL description, evaluation intervals of *signal assignment* statements can be determined. These intervals are further used to construct transaction and driver sets for signals in Section 8.3. These sets are then used to determine waveforms of signals.

It is important to understand that the construction of transaction and driver sets is not necessary. They are being constructed for ease of presentation. The Dynamic Model does not assert that a VHDL simulator *must* maintain transactions and drivers for signals. The simulator is free to follow any approach as long as it can generate the same waveforms of the signals. In Section 8.7.2, it will be shown that the definition of these intermediate sets is not required for implicit signals since they exhibit simple behavior. It is more difficult to characterize behavior of explicit signals and hence we have resorted to the definition of these intermediate sets.

8.2 EVALUATION OF VHDL STATEMENTS

In this section, we define the constraints on the evaluation intervals of VHDL statements in terms of $\mathcal{E}$ and σ as discussed above. Informally, the notion of constraints (that must be satisfied by the evaluation intervals of VHDL statements) can be explained as follows. Consider the *process* p2 in Figure 8.3. The statements s1, s2, s3, and s4 in p2 are evaluated sequentially in adjacent time intervals. Since s4 is the last statement in the *process*, its evaluation is followed by the evaluation of s1 and the cycle repeats. The constraint on the evaluation interval of the *signal assignment* statement s1 in p2 is that it must be contained within a delta interval. The constraint on the evaluation interval of the *wait* statement s2 in p2 is that it begins just after that of s1 (and p2 is said to be *suspended*) and ends when the condition for the termination of the *wait* statement is satisfied (and p2 is said to *fire*). These constraints, informally stated here, are formalized in the following sections.

8.2.1 The Process Statement

All VHDL concurrent statements have an equivalent *process* statement and hence the normal form of the Static Model only contains the representation of a *process* statement. The Dynamic Model, therefore, only deals with the dynamic semantics of the *process* statement.

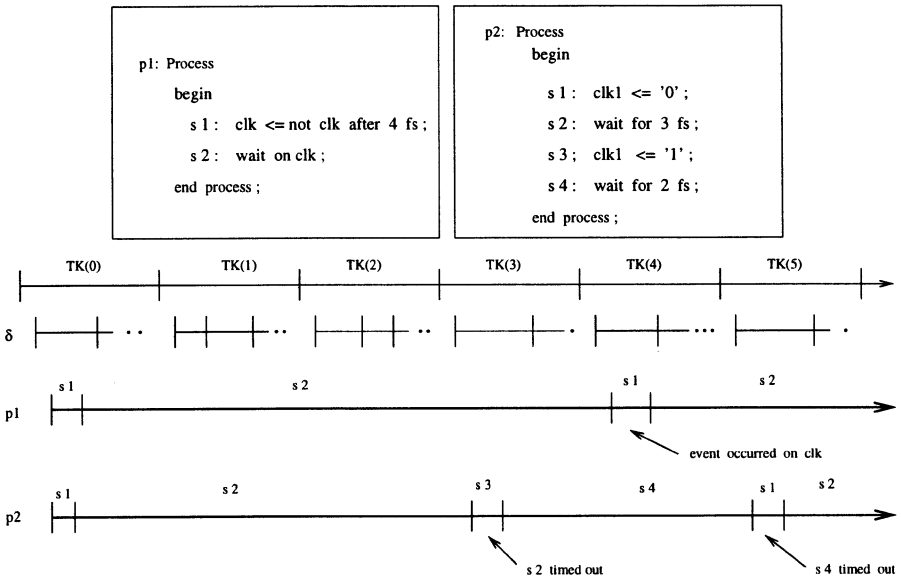


Figure 8.3. Parallel evaluation of process statements

The semantics of a *process* statement is defined a little differently from sequential statements because its evaluation occurs over an infinite interval [3] and there are no constraints to be established. Consequently, we do not define $\mathcal{E}$ and σ for a *process* statement. Instead, a set EV_P for a *process* P is defined. The notion of EV is similar to that of σ and is explained below. Evaluation of a *process* statement results in a repeated evaluation of its confined sequential statements. In Figure 8.3, the evaluation of *process* p1 leads to the repeated evaluation of s1 and s2. Let us assume that s1 is evaluated in the intervals t1, t3, t5, and so on and that s2 is evaluated in the intervals t2, t4, t6, and so on (*the duration of these intervals depends on the constraints they must satisfy*). Then, the set EV_{p1} is defined as

$$\mathsf{EV}_{p1} = \{ \langle \mathsf{s1}, \mathsf{t1} \rangle, \langle \mathsf{s2}, \mathsf{t2} \rangle, \langle \mathsf{s1}, \mathsf{t3} \rangle, \ldots , \}.$$

The evaluation intervals must also be adjacent to each other to enforce the order of evaluation of these sequential statements. Therefore, the following conditions must hold: (i) The predicates $\mathcal{E}$(s1, t1), $\mathcal{E}$(s2, t2), $\mathcal{E}$(s1, t3) and so on must be `true` and (ii) The predicates meets(t1, t2), meets(t2, t3), meets(t3, t4) and so on must be `true`. Note that EV is a set ordered by the relation meets operating on the time intervals in the set.

In general, the statement s1 could be an *if* statement or a *case* statement and therefore ⟨ s1, t1 ⟩ should be replaced by the elements of σ(s1, t1) in EV_{p1}.

Bearing these conditions in mind, the evaluation set EV_P for a *process* statement P can be formally defined by the axiom

$$\begin{aligned}
&\forall\, \mathsf{P} \in \mathsf{Process_Statements} \bullet \\
&\quad \exists\, \mathsf{EV}_P \bullet \mathsf{EV}_P = \bigcup_{i=0}^{\infty} \sigma(\mathsf{stmt}_i,\, \mathsf{t}_i) \wedge \\
&\qquad \forall\, \mathsf{i} \in N \bullet (\mathsf{stmt}_0 = \mathsf{P.ordered\text{-}statements}_0 \wedge \\
&\qquad\quad \mathsf{stmt}_{i+1} = \mathsf{SUCC}(\mathsf{stmt}_i) \wedge \\
&\qquad\quad (\mathsf{during}(\mathsf{t}_0,\, \delta_0(0)) \vee \mathsf{overlaps}(\delta_0(0),\, \mathsf{t}_0)) \wedge \\
&\qquad\quad \mathsf{meets}(\mathsf{t}_i,\, \mathsf{t}_{i+1}) \wedge \mathcal{E}(\mathsf{stmt}_i,\, \mathsf{t}_i)),
\end{aligned} \tag{8.1}$$

where SUCC is a function that merely returns the next statement in the sequence P.ordered-statements and stmt_i is a statement belonging to that sequence. If stmt_i is the last statement of the sequence, SUCC returns the first statement of the sequence. The above axiom defines the order and the time intervals in which the statements within a *process* are evaluated in a VHDL simulation. It *connects* the evaluation intervals of the sequential statements within the *process*. The initial statement of any *process* begins in the very first delta interval $\delta_0(0)$ (therefore all *process* statements are fired in the first delta interval of TK(0) though they need not be fired simultaneously). Furthermore, the relation $\mathsf{during}(\mathsf{t}_0,\, \delta_0(0)) \vee \mathsf{overlaps}(\delta_0(0),\, \mathsf{t}_0)$ holds because the evaluation of the first statement could be contained within $\delta_0(0)$ (as in the case of a *signal assignment* statement [refer to Section 8.2.2]) or extend beyond $\delta_0(0)$ (as in the case of a *wait* statement [refer to Section 8.2.4]). The following sections present the definitions of $\mathcal{E}$ and σ for sequential statements.

8.2.2 The Sequential Signal Assignment Statement

The *signal assignment* statement is used to update values of signals which represent wires in a digital circuit. Since a *signal assignment* statement with multiple waveforms is equivalent to multiple assignments with one waveform each, the Dynamic Model only deals with *signal assignments* with one waveform. Further, *signal assignments* with transport delays are represented as assignments with inertial delay and a 0 pulse rejection limit (refer to Section 3.8.4).

A *signal assignment* statement is constrained to be evaluated in a subinterval of some delta interval $\delta_i(\mathsf{j})$ following the proper order of the statements in a *process* statement. This constraint is formalized in terms of $\mathcal{E}$ as

$$\mathcal{E}(\mathsf{sa},\, \mathsf{t}) \Rightarrow \exists\, \mathsf{i},\, \mathsf{j} \bullet \mathsf{during}(\mathsf{t},\, \delta_i(\mathsf{j})).$$

Informally, the above axiom says that the predicate $\mathcal{E}$(sa,t) will be `true` only if t is a subinterval of some delta interval. Since the constraint only depends on t, this property will be referred to as μ(t) for later use. Formally,

$\mu(\mathsf{t}) \Leftrightarrow \exists\ \mathsf{i}, \mathsf{j} \bullet \mathsf{during}(\mathsf{t}, \delta_i(\mathsf{j}))$.

The effect of a signal assignment on the value of its destination is not defined by the above axiom. These effects are captured in Sections 8.3 and 8.4. *The above axiom establishes the constraint on the time interval in which a signal assignment statement is to be evaluated.* Furthermore, we need to state that this evaluation interval lies between the evaluation intervals of the statements that precede and succeed the *signal assignment* statement. These *ordering* constraints were established in the definition of EV_P in Section 8.2.1. The evaluation set of sa is given by

$\sigma(\mathsf{sa}, \mathsf{t}) = \{ \langle\ \mathsf{sa}, \mathsf{t}\ \rangle \}$,

since a *signal assignment* statement is evaluated by itself and does not involve evaluations of other statements.

The definitions of $\mathcal{E}$ and σ for *variable assignment, assert,* and *report* statements are identical and are therefore not presented here. Moreover, any condition is also evaluated within a delta interval and therefore the definition of $\mathcal{E}$ and σ for a condition evaluation is also the same. The definitions of $\mathcal{E}$ and σ for the *case* and *loop* statements are lengthy and are not presented here.

8.2.3 The Sequential If Statement

An *if* statement contains a list of conditions and corresponding sequences of statements from which one of the sequences is selected for execution depending on the value of the conditions.

The evaluation of an *if* statement involves the evaluation of the sequential statements that appear in the clause whose condition evaluates to `true` provided that all conditions before it (if any) evaluate to `false`. If all the conditions evaluate to `false`, then no statement is executed. Conditions are evaluated in time intervals t such that $\mu(\mathsf{t})$ holds. Thus, the definition of $\mathcal{E}$ for an *if* statement can be given by

$\mathcal{E}(\mathsf{if}, \mathsf{t}) \Rightarrow$

$[\ \exists\ \mathsf{i}, \mathsf{t}_1, \mathsf{t}_2 \bullet$
$\mathsf{t}_1 + \mathsf{t}_2 = \mathsf{t} \wedge \mu(\mathsf{t}_1) \wedge \mathsf{VAL}(C_i, \mathsf{t}_1) = \mathsf{true} \wedge$
$(\forall\ \mathsf{j} \in N, \mathsf{j} < \mathsf{i} \bullet \mathsf{VAL}(C_j, \mathsf{t}_1) = \mathsf{false}) \wedge \mathcal{E}(OS_i, \mathsf{t}_2) \wedge$
$\sigma(\mathsf{if}, \mathsf{t}) = \{ \langle\ C_i, \mathsf{t}_1\ \rangle \} \cup \sigma(OS_i, \mathsf{t}_2)\ ]\ \vee$

$[\ \mu(\mathsf{t}) \wedge (\forall\ \mathsf{k} \in N \bullet \mathsf{VAL}(C_k, \mathsf{t}) = \mathsf{false}) \wedge \sigma(\mathsf{if}, \mathsf{t}) = \{ \langle\ \mathsf{if}, \mathsf{t}\ \rangle \}\ ]$,

where C_i refers to $(\mathsf{if.if\text{-}choice\text{-}list})_i.\mathsf{condition}$ and OS_i refers to $(\mathsf{if.if\text{-}choice\text{-}list})_i.\mathsf{ordered\text{-}statements}$. The function VAL(E,t) returns the value of expression E in the time interval t if E has a constant value in t. If t is so large that E cannot have a constant value in t, then VAL returns UNDEFINED. Note that t is already constrained to be "small enough" in the above axiom since $\mu(\mathsf{t})$ must hold.

8.2.4 The Sequential Wait Statement

A *wait* statement causes the suspension of a *process* statement. It models circuit delays and sensitivity of components to signals. *Process* statements with sensitivity lists are equivalent to *process* statements with no sensitivity list but with an added *wait* statement that waits on the signals in the sensitivity list. For this reason, sensitivity lists are not handled explicitly.

In order to terminate the evaluation of a *wait* statement, an event must occur on a signal in its sensitivity list and its condition must evaluate to `true`. If this does not happen within the time limit specified in the timeout clause, the evaluation terminates when the time limit has elapsed. For example, in Figure 8.3, the evaluation of a *wait* statement s2 in *process* p2 terminates in $\delta_3(0)$ because the timeout condition is satisfied in that delta interval. This is modeled in terms of the predicate SATISFIES(wa, i, j, k, l) which is `true` for a *wait* statement wa whose evaluation begins in δ_i(j) and ends in δ_k(l). The predicate is defined by the axiom

$$
\begin{aligned}
&\mathsf{SATISFIES(wa,\ i,\ j,\ k,\ l)} \Rightarrow \\
&\quad \mathsf{i} \leq \mathsf{k} \wedge ((\exists\ \mathsf{s} \in \mathsf{Signals} \cup \mathsf{Ports} \bullet \\
&\qquad\qquad \mathsf{s} \in \mathsf{wa.sensitivity\text{-}list} \wedge \mathsf{event(s,}\ \delta_k(\mathsf{l})) \wedge \\
&\qquad \mathsf{VAL(wa.condition,}\ \delta_k(\mathsf{l})) = \mathsf{true}) \vee \\
&\qquad (\mathsf{wa.timeout} = \mathsf{k\text{-}i} \wedge \mathsf{l} = 0 \wedge \mathsf{k} \neq \mathsf{i}\ ) \vee \\
&\qquad (\mathsf{wa.timeout} = 0 \wedge \mathsf{l} = \mathsf{j{+}1} \wedge \mathsf{k} = \mathsf{i}\ )),
\end{aligned}
$$

where event is a predicate (formally defined in Section 8.7.1) which is `true` whenever a VHDL event occurs on a signal in a specified delta interval.

The predicate $\mathcal{E}$(wa, t) for a *wait* statement wa in a non-postponed process can now be defined. The constraints on t are that it begins in a particular delta interval δ_i(j) and ends in another delta interval δ_k(l) (*i.e.,* overlaps(δ_i(j), t) $\wedge$ overlaps(t, δ_k(l)) holds) such that the predicate SATISFIES(wa, i, j, k, l) is `true` and no intermediate interval δ_m(n) exists such that SATISFIES(wa, i, j, m, n) is `true`. Stated formally, a *wait* statement wa in a non-postponed *process* evaluates in a time interval t such that

$$
\begin{aligned}
&\mathcal{E}\mathsf{(wa,\ t)} \Rightarrow \\
&\quad \exists\ \mathsf{i, j, k, l} \in N \bullet \mathsf{overlaps}(\delta_i(\mathsf{j}),\ \mathsf{t}) \wedge \mathsf{SATISFIES(wa,\ i,\ j,\ k,\ l)} \wedge \\
&\qquad \mathsf{overlaps(t,}\ \delta_k(\mathsf{l})) \wedge \\
&\qquad (\neg\ \exists\ \mathsf{m, n} \in N, (\mathsf{m} < \mathsf{k}) \vee (\mathsf{m} = \mathsf{k} \wedge \mathsf{n} < \mathsf{l}) \bullet \\
&\qquad\qquad \mathsf{SATISFIES(wa,\ i,\ j,\ m,\ n)})
\end{aligned}
$$

holds. Note that if the timeout value of the *wait* statement is 0 units then the evaluation interval of the *wait* statement ends in the very next delta interval within the same TK(i) (thus, occurring without the elapse of [modeled] real time). If the timeout value is greater than 0 units, then the evaluation interval

ends in the first delta interval of the appropriate TK interval. The evaluation set of the *wait* statement is given by

$$\sigma(\mathsf{wa}, \mathsf{t}) = \{ \langle \mathsf{wa}, \mathsf{t} \rangle \}.$$

The predicate $\mathcal{E}$ and the set σ for *wait* statements in a postponed *process* are defined in a similar manner (refer [53]). This section concludes the process of establishing the constraints on the time intervals in which VHDL statements can be evaluated. The following sections deal with the effect of executing VHDL statements on the values of signals.

8.3 TRANSACTION LISTS

Execution of *signal* and *variable assignment* statements affect values of signals and variables respectively. Variables are updated immediately whereas signals are updated after a specified delay. In order to determine waveforms of signals, it is convenient to construct intermediate sets that store the value and delay information of a signal assignment. The LRM defines *drivers* that store the necessary information for updating signals. The execution of a *signal assignment* statement results in a *transaction* containing the value and delay information being posted on the corresponding driver. A specific algorithm (referred to as the *marking* algorithm) for adding and deleting transactions from the drivers is provided in the LRM (§8.5, ¶118). The Dynamic Model provides a declarative definition of drivers as being a list of transactions. The definition does not specify an algorithm that sequentially adds and deletes members to/from the list, rather, it *declaratively specifies the list contents for a given VHDL description.* It should be stated again that given the approach used in this work, the definition of drivers is not even necessary. Once the relationships between evaluation intervals of statements is known (as we already do), the waveforms of signals (the state space) can be directly inferred. The formalization of drivers simplifies the definition of the waveforms of *explicit* signals (Sections 8.4.1 and 8.4.2). The definitions of waveforms of *implicit* signals (Section 8.7.2) does not require the formalization of drivers because implicit signals exhibit simple behavior.

The LRM states that a unique driver is associated with a signal and a *process* if there exists a *signal assignment* in the *process* and its destination is the signal. Each execution of the assignment corresponds to a transaction being posted on the driver.

In the Dynamic Model, a transaction posted by a *signal assignment* statement is represented as a three tuple $\langle \mathsf{v}, \mathsf{t}_r, \mathsf{t}_d \rangle$ where v is the value assigned to the destination of the assignment, t_r denotes the pulse rejection limit of the assignment, and t_d denotes the delay of the assignment. The interval t_d is such that it extends from the time at which the evaluation of the assignment is complete, to the time at which the value of the assignment is slated to appear on the destination signal. For example, in Figure 8.4, since the assignment s3

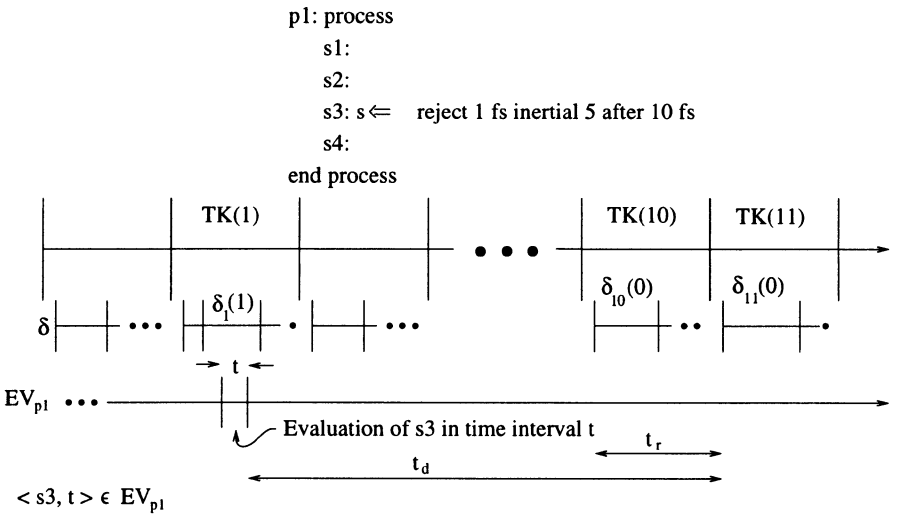


Figure 8.4. A transaction

is evaluated in t (which lies in $\delta_1(1)$) and its delay is 10 units, the destination s must acquire the value of the assignment in $\delta_{11}(0)$. Therefore, t_d extends from the end of t (*i.e.*, meets(t, t_d) holds) to the beginning of $\delta_{11}(0)$ (*i.e.*, meets(t_d, $\delta_{11}(0)$) holds). The transaction is said to *project* the value of the assignment in $\delta_{11}(0)$. Instead, if the delay of s3 is 0 units, then s must acquire the value of the assignment in the next delta interval and hence meets(t_d, $\delta_1(2)$) must hold. If the pulse rejection limit is 0 units, then $t_r = \phi$.

The formal definition of drivers proceeds in two steps. First, a *pre-marking* set ALL_TR(s, P) containing all the transactions posted by the *signal assignment* statements to s in *process* P is defined. Second, the driver Driver(s, P) denoting the 'true' transactions (transactions not preempted by the marking algorithm) is defined. Furthermore, ALL_TR(s, P) also contains a transaction ⟨ s.initial-value, ϕ, $\mathcal{T}$ ⟩ which posts the initial value of s where $\mathcal{T}$ is shown in Figure 8.1. Formally, ALL_TR(s, P) is defined as

$$
\begin{aligned}
&\text{ALL\_TR(s, P)} = \qquad\qquad (8.2)\\
&\quad \{ \langle \text{ s.initial-value}, \phi, \mathcal{T} \rangle \} \cup \\
&\quad \{ \langle \text{ v}, t_r, t_d \rangle : (\text{v} = \text{s.initial-value} \wedge t_r = \phi \wedge \text{meets}(t_d, \delta_0(0))) \vee \\
&\qquad (\exists \text{ sa, i, j, t} \bullet (\text{sa} \in \text{Signal-Assignments} \wedge \\
&\qquad\quad \langle \text{ sa, t } \rangle \in \text{EV}_P \wedge \text{sa.destination} = \text{s} \wedge \\
&\qquad\quad \text{during(t, } \delta_i(\text{j})) \wedge \text{meets(t, } t_d) \wedge \text{VAL(sa.expr, t)} = \text{v} \wedge \\
&\qquad\quad \textit{if } (\text{sa.delay} = 0) \textit{ then} \\
&\qquad\qquad \text{meets } (t_d, \delta_i(\text{j+1})) \wedge t_r = \emptyset \\
&\qquad\quad \textit{else if } (\text{sa.pulse-rejection} = 0) \textit{ then} \\
&\qquad\qquad \text{meets } (t_d, \delta_{i+sa.delay}(0)) \wedge t_r = \emptyset
\end{aligned}
$$

else if (sa.delay = sa.pulse-rejection) *then*
meets $(t_d, \delta_{i+sa.delay}(0)) \wedge t_d = t_r$
else
begins$(\delta_{i+sa.delay+sa.pulse-rejection}(0), t_r) \wedge$
meets $(t_r, \delta_{i+sa.delay}(0)) \wedge$
meets $(t_d, \delta_{i+sa.delay}(0)))$ }.

The above axiom merely states the relationships captured in Figure 8.4. If the specified delay of a *signal assignment* statement sa is 0 units, then it corresponds to an element $\langle v, t_r, t_d \rangle$ in ALL_TR where t_d extends from the end of the evaluation interval t of sa, to the beginning of the very next delta interval. If the delay is 0 units then the pulse rejection is also 0 units and thus $t_r = \emptyset$. If the specified delay is x units ($x > 0$) and pulse rejection is 0 units, then t_d extends from the end of t to the beginning of the delta interval occurring after x units and $t_r = \emptyset$ as before. If the specified delay and the pulse rejection limit are positive non-zero values then the relations on t_d and t_r are as in the example depicted in Figure 8.4.

The effect of the LRM marking algorithm can be informally explained as follows. If $\langle v, t_d, t_r \rangle$ is a transaction posted by a *signal assignment* statement and $\langle v', t'_d, t'_r \rangle$ is the transaction posted by a later assignment (to the same signal) such that the former transaction is projected at or after the time projected by the latter, then the relationship between t_d and t'_d is given by during(t'_d, t_d) $\vee$ finishes(t'_d, t_d). In such a case, the former transaction is a false transaction and must not appear in the final driver (*i.e.*, it gets preempted). If the latter transaction is such that it projects a value *after* the former transaction but the former projection falls within the pulse rejection limit of the latter, then the relation between t_d, t'_d, and t'_r is given by overlaps$(t_d, t'_d) \wedge$ overlaps(t_d, t'_r). The condition that must hold for the former transaction to be a false one in this case is either that (i) the value projected by the latter is different from the value projected by the former or that (ii) the value projected by the former is same as that projected by the latter and there exists a transaction that projects a different value at a time-stamp in between the earlier two time-stamps.

A transaction in ALL_TR(s, P) is a false transaction if it meets any of the conditions stated above and hence must not be included in the driver Driver(s, P). Thus, a driver Driver(s, P) can be defined as

$$
\begin{aligned}
&\text{Driver(s, P)} = \qquad\qquad (8.3)\\
&\quad \{ \langle v, t_r, t_d \rangle : \langle v, t_r, t_d \rangle \in \text{ALL\_TR(s, P)} \wedge \\
&\qquad \neg \exists \langle v', t'_r, t'_d \rangle \in \text{ALL\_TR(s, P)} - \{ \langle v, t_r, t_d \rangle \} \bullet \\
&\qquad\quad \text{during}(t'_d, t_d) \vee \text{finishes}(t'_d, t_d) \vee \\
&\qquad\quad ((\text{overlaps}(t_d, t'_r) \vee \text{meets}(t_d, t'_r)) \wedge \\
&\qquad\qquad (v' \neq v \vee (v' = v \wedge \exists \langle v'', t''_r, t''_d \rangle \in \text{ALL\_TR(s, P)} \bullet \\
&\qquad\qquad\qquad\qquad v'' \neq v' \wedge \text{overlaps}(t_d, t''_d) \wedge \\
&\qquad\qquad\qquad\qquad \text{overlaps}(t''_d, t'_d)))) \}.
\end{aligned}
$$

The above axiom asserts that a transaction $\langle v, t_d, t_r \rangle \in$ ALL_TR(s, P) is in Driver(s, P) if there does not exist another transaction $\langle v', t'_d, t'_r \rangle$ such that its presence would cause the preemption of the former transaction.

Every *process* tuple has a field set-of-drivers that is a set of all the drivers of all the signals that are destinations of *signal assignment* statements in the *process*. Formally,

$$
\begin{aligned}
&\forall P \in \text{Process\_Statements} \bullet \\
&\quad P.\text{set-of-drivers} = \\
&\qquad \{ \langle s, D \rangle : s \in \text{Signals} \cup \text{Ports} \wedge \\
&\qquad\quad (\exists\, sa \in \text{Signal-Assignments} \bullet \\
&\qquad\qquad sa \in pr.\text{ordered-statements} \wedge sa.\text{destination} = s) \wedge \\
&\qquad\quad D = \text{Driver}(s, P) \}.
\end{aligned}
$$

This completes the definition of the sets representing drivers of signals. The declarative style that we have used for defining the VHDL preemption mechanism is useful in that it can be used to validate other algorithms that achieve the same effect as the LRM marking algorithm.

8.3.1 The Notion of Similarity

In Section 8.3, we defined transaction sets for signals. These sets contain time intervals that are used to determine the waveforms of the signals. Therefore, instead of reasoning about waveforms of signals, we could directly reason about the time intervals in the transaction sets. Proofs in Chapter 9 construct only transaction sets. A transaction $\langle v, t_r, t_d \rangle$, as mentioned before, is defined such that the value v is acquired by the appropriate signal at a point in simulation where t_d ends. The time interval t_r and the point at which t_d begins have no effect on the waveform of the signal (once it has been determined whether the transaction will be preempted or not). Therefore, if two transactions $\langle v, t_r, t_d \rangle$ and $\langle v', t'_r, t'_d \rangle$ are such that the relation

$$v = v' \wedge \text{ends}(t_d, t'_d).$$

holds, then the two transactions would have the same effect on the waveform of a signal. Such two transactions are said to be *similar*. Further, we say that two drivers D and D′ are *similar* if, for every transaction in D, there exists a *similar* transaction in D′ and for every transaction in D′ there exists a *similar* transaction in D. The same definition holds for similarity between *pre-marking* sets.

The significance of similarity is that two *similar* drivers lead to the same waveform for the signal they drive. This idea will be revisited in Section 9.1.

8.4 THE STATE SPACE

VHDL allows a hierarchical description of hardware and therefore signals and ports within a block may be associated as actuals with a formal port of a block at a lower level in the hierarchy. The Static Model represents these associations as the set Port-Associations (refer to Section 3.4.2).

The value of a signal s at the lowest level in the hierarchy changes due to the execution of *signal assignment* statements with destination s. These changes must also be reflected in the ports or signals that are associated (or connected) with s at a higher level. Similarly, changes at the top of the hierarchy must flow down to the lower levels. Upward flow of information is achieved by calculating the *driving value* of all signals (therefore driving values are not defined for ports of mode `in`) and downward flow is achieved by calculating *effective values* for all signals (therefore effective values are not defined for ports of mode `out`). For this book, it is assumed that the effective values are the *current values* or the desired waveforms.[4] We first define the driving value of signals and then their effective values. *The set of effective values of a signal over the complete simulation time constitutes the 'waveform' of the signal.*

8.4.1 Driving Values

The driving value of a signal (§12.6.2, ¶165, Ł480) is defined in the LRM as "the value that the signal provides as a source of other signals." The driving value for any signal can be determined from the driving value of its *sources.* The sources for a signal are the ports that are connected to it (lower down in the hierarchy) and the drivers associated with it.

For a signal s, if there is no *signal assignment* statement whose destination is s, and if s is not associated with any other port, it has no source. This may happen if a signal is declared in an architecture but no assignment is made to it or if a port is *not* of mode `in` and is at the lowest level in a hierarchy and no assignment is made to it. This is formally stated in terms of a predicate nullsources(s) as

$$\begin{aligned}\text{nullsources(s)} \equiv \neg\, \exists\ \text{sa} \in \text{Signal-Assignments} \bullet\ \text{sa.destination} = \text{s} \wedge \\ \neg\, \exists\ \text{pa} \in \text{Port-Associations} \bullet\ \text{pa.actual} = \text{s},\end{aligned}$$

which states that nullsources(s) is `true` if s has no source and `false` otherwise. For a signal which has no source, its driving value is the initial value for all time. Formally,

$$\begin{aligned}\forall\ \text{s, t: } (\text{s} \in \text{Signals} \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \text{nullsources(s)} \bullet \\ \mathcal{F}_{dr-val}(\text{s, t}) = \text{s.initial-value},\end{aligned}$$

where $\mathcal{F}_{dr-val}$ is a function similar to VAL and defines the driving value for a signal during a time interval.[5] For the remaining cases, the set of sources for

a signal is constructed and is used to define the driving value for that signal. The set of sources for a signal s is the union of the set of all drivers associated with s and the set of all ports which are formals and are involved in a port association with s and s is the actual in the association.

Considering only drivers first, recall from Section 8.3 that a driver D (where D = Driver(s, P)) stores information about the value to be assigned to s and the time at which this value is to be assigned. Referring to Figure 8.4, if $\langle$ 5, t_d, t_r $\rangle \in$ D, then for any time interval t if meets(t_d, t) $\vee$ before(t_d, t) holds, the value contributed by D in t will be 5 if no other value is scheduled for s between the upper bound of t_d and the lower bound of t by a different transaction. Thus, the function D_Val which returns the value contributed by a driver in time interval t is defined as

$$\begin{aligned}&\text{D\_Val(D, t)} = \text{v} \Leftrightarrow \\ &\quad \langle\, \text{v}, t_d, t_r \,\rangle \in \text{D} \wedge (\text{meets}(t_d, \text{t}) \vee \text{before}(t_d, \text{t})) \wedge \\ &\qquad \neg\exists\, \langle\, \text{v}', t'_d, t'_r \,\rangle \in \text{D} - \{\, \langle\, \text{v}, t_d, t_r \,\rangle\, \} \bullet \\ &\qquad\quad \text{endsafter}(t'_d, t_d) \wedge \text{endsafter}(\text{t}, t'_d).\end{aligned}$$

It should be noted that D_Val(D, t) (and hence all other functions defined below) is valid only for time intervals where the value of the driver is constant. One such convenient interval is a delta interval. The reader may find it useful to replace every occurrence of t by a delta interval in the following axioms. We now define the set All_Drivers which is a collection of the values contributed by all the drivers associated with a signal in a given time interval t. If the contributed value is `null` (due to a null assignment to the signal associated with the driver), it is not included in the set and the corresponding driver is said to be "disconnected." Formally, this is stated as

$$\begin{aligned}&\forall\, \text{s}: ((\text{s} \in \text{Signals}) \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \neg\, \text{nullsources(s)} \bullet \\ &\quad \text{All\_Drivers(s, t)} = \{\, \text{v} : \exists\, \text{D}, \\ &\qquad \langle\, \text{s}, \text{D} \,\rangle \in \bigcup_{P \in Process-Statements} \text{P.set-of-drivers} \bullet \\ &\qquad\qquad \text{v} = \text{D\_Val(D, t)} \wedge \text{v} \neq \texttt{null}\, \}.\end{aligned}$$

The set All_Ports consists of values derived from any ports that may be associated with the given signal. This is formulated as

$$\begin{aligned}&\forall\, \text{s}: ((\text{s} \in \text{Signals}) \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \neg\, \text{nullsources(s)} \bullet \\ &\quad \text{All\_Ports(s, t)} = \{\, \text{v} : \exists\, \text{pa} \in \text{Port-Associations} \bullet \\ &\qquad\qquad \text{pa.actual} = \text{s} \wedge \\ &\qquad\qquad \text{v} = \text{pa.fn-f2a}(\mathcal{F}_{dr-val}(\text{pa.formal}, \text{t}))\}.\end{aligned}$$

The above axiom applies the function $\mathcal{F}_{dr-val}$ recursively to determine the values contributed by the ports connected to signal s. In general, the driving value of a signal that is an actual in a port association depends on the driving value of the formal lower down in the hierarchy. At the lowest level in the hierarchy,

the driving value of the formal will be determined by signal assignments made to it (*i.e.,* the driving value will be determined by the drivers). The set of all sources can now be defined as the union of All_Drivers and All_Ports. Formally,

$$\text{All\_Sources(s, t)} = \text{All\_Ports(s, t)} \cup \text{All\_Drivers(s, t)}.$$

Thus, the set All_Sources(s, t) contains the entire set of values contributed by the various sources of signal s in the time interval t. If the set has more than one value, the resolution function will be needed to generate the value that is to be assigned to s. Note that All_Sources(s, t) could be empty even if s has sources but they are all disconnected during the time interval t. In the case where s is not a resolved signal (*i.e.,* s.res-fn $= \phi$), the set All_Sources(s, t) must have only one element and this is the value that must be assigned to s in t. This is stated formally as

$$\begin{aligned}&\forall\ \text{s, t: } ((\text{s} \in \text{Signals}) \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \text{s.res-fn} = \phi \wedge \\ &\quad \neg\ \text{nullsources(s)} \bullet \\ &\quad \mathcal{F}_{dr-val}(\text{s, t}) = \text{v} \wedge \text{v} \in \text{All\_Sources(s, t)}.\end{aligned}$$

It would be an error if the signal or port is unresolved and the set All_Sources has more than one element in it. In the case of resolved signals with at-least one source, if All_Sources is empty and the signal kind is `register` then the value of the signal is unchanged from the previous time interval. Otherwise the driving value is determined by applying the resolution function to All_Sources. This is formally stated as

$$\begin{aligned}&\forall\ \text{s, t: } ((\text{s} \in \text{Signals}) \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \text{s.res-fn} \neq \phi \wedge \\ &\quad \neg\ \text{nullsources(s)} \wedge \\ &\quad \text{All\_Sources(s, t)} = \phi \wedge \text{s.signal-kind} = \texttt{register} \bullet \\ &\qquad \mathcal{F}_{dr-val}(\text{s, t}) = \mathcal{F}_{dr-val}(\text{s, } \hat{t}) \wedge \text{meets}(\hat{t}, \text{t}),\end{aligned}$$

$$\begin{aligned}&\forall\ \text{s, t: } ((\text{s} \in \text{Signals}) \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \text{s.res-fn} \neq \phi \wedge \\ &\quad \neg\ \text{nullsources(s)} \wedge \\ &\quad \neg(\text{All\_Sources(s, t)} = \phi) \wedge \neg(\text{s.signal-kind} = \texttt{register}) \bullet \\ &\qquad \mathcal{F}_{dr-val}(\text{s, t}) = \text{s.res-fn(All\_Sources(s, t))}.\end{aligned}$$

This completes the definition of the driving values for signals. The reader may find a one to one correspondence between the axioms presented herein and the rules for updating the driving values presented in the LRM (§12.6.2, ¶165).

8.4.2 Effective Values

The LRM (§12.6.2, ¶165, L481) defines the effective value of a signal to be the value "obtainable by evaluating a reference to the signal within an expression." The axioms defining effective values are similar to the rules presented in the

LRM (§12.6.2, ¶166). The effective value of declared signals, ports of mode `buffer`, and unconnected ports of mode `inout` is equal to their driving values. Formally, this is expressed as

$$\forall\ s\text{: } s \in \text{Signals} \vee (\ s \in \text{Ports} \wedge (\text{s.mode} = \texttt{buffer} \vee (\text{s.mode} = \texttt{inout} \wedge \text{unconnected(s)}))) \bullet \mathcal{F}_{eff-val}(s, t) = \mathcal{F}_{dr-val}(s, t),$$

where unconnected(s) is a predicate that is `true` if s is a formal in a port association and is associated with an actual which is the reserved keyword open. For a connected port of mode `in` or `inout`, the effective value is defined to be "the effective value of the actual part of the association element that associates an actual with the signal." Note that the effective value flows downwards in the hierarchy of ports (*i.e.*, it flows from the actual to the formal). Formally, the transmission of effective values down the hierarchy (complete with type conversion) is expressed as

$$\forall\ \text{pa: pa} \in \text{Port-Associations} \wedge (\text{pa.formal.mode} = \texttt{in} \vee \text{pa.formal.mode} = \texttt{inout}) \bullet \mathcal{F}_{eff-val}(\text{pa.formal}, t) = \text{pa.fn-a2f}\ (\mathcal{F}_{eff-val}(\text{pa.actual}, t)).$$

In the case where the port is an unconnected port of mode `in`, the effective value is given by the default value associated with the port. Formally,

$$\forall\ s, t\text{: } s \in \text{Ports} \wedge \text{unconnected(s)} \wedge \text{s.mode} = \texttt{in} \bullet \mathcal{F}_{eff-val}(s, t) = \text{s.initial-value}.$$

Having defined the effective values of signals for a given delta interval, their waveform over the entire simulation can be defined.

8.5 WAVEFORMS

A waveform of a signal is a set of two-tuples $\langle\ v, t\ \rangle$ where v is a value and t is a time interval. Two separate notions of waveforms are discussed below. The "real" waveform of a signal s, $\mathcal{W}(s)$, is the set of two-tuples $\langle\ v, t\ \rangle$ where t is a delta interval[6] and v is the value of s in that interval. In addition, it includes the tuple $\langle$ s.initial-value, $\hat{T}\ \rangle$ where $\hat{T}$ is as shown in Figure 8.1. Formally,

$$\mathcal{W}(s) = \{\ \langle\ \text{s.initial-value}, \hat{T}\ \rangle\ \} \cup \{\ \langle\ v, t\ \rangle : t = \delta_i(j) \wedge v = \mathcal{F}_{eff-val}(s, \delta_i(j))\ \}.$$

We can now state the theorem for the equivalence of two VHDL descriptions with respect to a selected set of signals that are common to both descriptions.

Theorem 8.1 *If S is a set of signals common to two VHDL descriptions V1 and V2, V1 and V2 are equivalent with respect to S if for all signals s ∈ S, $\mathcal{W}(s)$ in V1 = $\mathcal{W}(s)$ in V2.*

8.6 OBSERVABILITY

It is sometimes useful to ignore the delta transitions of values of signals since they only represent temporary instability in the hardware. One need only look at the stable or *observable* values on the signal. Correspondingly, we define the "observable" waveform of s, $\mathcal{W}'(s)$ as a the set of two tuples $\langle$ v, t $\rangle$ where t is a time-keeper interval and v is the last value acquired by s in the interval. Formally,

$$\mathcal{W}'(s) = \{ \langle \text{s.initial-value}, \hat{\mathcal{T}} \rangle \} \cup \{ \langle v, t \rangle : t = TK(i) \wedge v = \mathcal{F}_{eff-val}(s, \delta_i(j)) \wedge \neg\exists\, k{:}\ k > j \bullet \mathcal{F}_{eff-val}(s, \delta_i(j)) \neq \mathcal{F}_{eff-val}(s, \delta_i(k)) \}.$$

Essentially, the observable waveform ignores the transient behavior of signals and only stores the final stable value in each time keeper interval. This definition can be used to prove "observable equivalence" of two VHDL descriptions or to validate transformations that preserve "observable behavior".

Theorem 8.2 *If S is a set of signals common to two VHDL descriptions V1 and V2, V1 and V2 are observably equivalent with respect to S if for all signals* $s \in S$, $\mathcal{W}'(s)$ *in V1* $= \mathcal{W}'(s)$ *in V2.*

8.7 ATTRIBUTES

8.7.1 S'Event

In VHDL, s'event is a function that returns the value `true` in a simulation cycle if the value of s has changed in that cycle, otherwise it returns `false`. This function is modeled as a predicate event(s, δ_i(j)) given by

$$\text{event}(s, \delta_i(j)) \Leftrightarrow \exists\, \hat{t} \bullet \text{meets}(\hat{t}, \delta_i(j)) \wedge \mathcal{F}_{eff-val}(s, \delta_i(j)) \neq \mathcal{F}_{eff-val}(s, \hat{t}).$$

Note that the definition of event is provided only for restricted time intervals (namely the delta intervals). Recall that this predicate was used in the definition of the semantics of a *wait* statement (Section 8.2.4).

8.7.2 S'Delayed(T)

For every expression s'delayed(T) that appears in a VHDL description, the LRM defines a new implicit signal that has the same waveform as s but delayed by T units of time. The LRM states that this implicit signal must be updated by executing the *process* statement

```
Process (s)
begin
      r <= transport s after T;
end process;.
```

The value of r in the above *process* statement stores the value of s'delayed(T). However, note that in the Dynamic Model, the behavior of s'delayed(T) can be defined without formalizing this *process* statement.

The waveform of s'delayed(T) is different for the case when $T = 0$ and the case when $T > 0$.

1. Case $T = 0$: In this case, the waveform of s'delayed(T) is the "real" waveform of s shifted by a delta interval. Formally,

$$\mathcal{W}(\text{s'delayed(0)}) = \{ \langle \text{ s.initial-value}, \hat{T} \rangle \} \cup \{ \langle \text{ v, t} \rangle : \exists \text{ i,j} \bullet \langle \text{ v}, \delta_i(\text{j}) \rangle \in \mathcal{W}(\text{s}) \wedge \text{t} = \delta_i(\text{j+1}) \}.$$

2. Case $T > 0$: In this case, the waveform of s'delayed(T) is the "observable" waveform of s shifted by T units of time (with respect to the time-keeper). Formally,

$$\mathcal{W}(\text{s'delayed(T)}) = \{ \langle \text{ s.initial-value}, \hat{T} \rangle \} \cup \{ \langle \text{ v, t} \rangle : \exists \text{ i} \bullet \langle \text{ v}, \text{TK(i)} \rangle \in \mathcal{W}'(\text{s}) \wedge \text{t} = \text{TK(i+T)} \}.$$

This section demonstrates some powerful features of the Dynamic Model. First, the difference in behavior of s'delayed for the cases $T = 0$ and $T > 0$ is exposed in the model. This difference is not very apparent in an operational characterization of the VHDL simulation cycle. Second, we have shown that it is possible to determine the waveform of a signal (s'delayed(T) in this case) without constructing any driver for the signal. Indeed, the waveform of s'delayed(T) is simply defined in terms of the waveform of s.

8.8 CONCLUSIONS

This chapter provided a dynamic semantic model of VHDL. The approach used in defining the semantics is unique in that it views the behavior of a VHDL description in terms of the waveforms of the signals rather than the description of the simulation. An alternative mechanism of simulation that can yield the same waveform for a given set of signals is automatically correct with respect to that set of signals. Unfortunately, the definition of waveforms of signals is complicated and forced us to first define certain intermediate sets storing transaction and driver information. These definitions, however, were not required for implicit signals. Further, the definition of these sets simplifies proofs presented in the next chapter.

Notes

1. Files also form a part of the state space but they are not covered in this work.

2. The term *virtual* time was coined by Jefferson [28] to refer to simulation time in his description of the Time Warp paradigm for synchronizing parallel discrete event simulations.

3. It is assumed that simulation does not stop at any time.

4. The only case when they may not be equal is that of arrays which are not characterized herein.

5. The time intervals should be chosen such that the value of the signal is constant during the interval.

6. t has been chosen to be a delta interval for the sake of convenience.

9 APPLICATIONS OF THE DYNAMIC MODEL

This section examines the applications of the Dynamic Model in formally reasoning about VHDL programs. In particular, the model is used to validate two transformation rules (Sections 9.2 and 9.3) and to derive conditions under which no transaction preemption occurs (Section 9.4). The results derived herein are aimed at CAD tool optimization.

9.1 SIMILARITY REVISITED

In the following sections, transformations are proved by showing the equivalence of VHDL descriptions before and after the transformation. Equivalence is established with respect to a given set of signals common to the two descriptions. In Section 8.3.1 we showed that the waveforms of two signals will be identical if their drivers are *similar*. We can now state the condition for the equivalence of two VHDL descriptions in terms of similarity.

Theorem 9.1 *If S is a set of signals common the two VHDL descriptions V1 and V2, then V1 is equivalent to V2 with respect to S if for every signal $s \in S$, the set of drivers for s in V1 are similar to the set of drivers for s in V2.*

9.2 PROCESS FOLDING

Communication overhead is a bottleneck in parallel simulation. In a parallel VHDL simulator, *process* statements are executed as parallel threads which

communicate with one another. Process folding [31] is an optimization technique that combines two *process* statements into a single *process* statement, thereby reducing the number of threads in the simulation which in turn speeds up the simulation. Results have indicated an improvement by a factor of 2.2 after folding. In this section, we validate a special case of process folding.

Consider the *process* statements P1 and P2 shown below.

```
P1:  process                          P2:  process
begin                                 begin
  Seq 1:  ⟨sequence of statements⟩      Seq' 1:  ⟨sequence of statements⟩
  W1:  wait for X ns;                   W2:  wait for X ns;
  Seq 2:  ⟨sequence of statements⟩      Seq' 2:  ⟨sequence of statements⟩
end process P1;                       end process P2;
```

These *process* statements can be folded into a single *process*, P3, shown below

```
P3:  process
begin
    Seq 1:  ⟨ sequence of statements ⟩
    Seq' 1:  ⟨ sequence of statements ⟩
    W3:  wait for X ns;
    Seq 2:  ⟨ sequence of statements ⟩
    Seq' 2:  ⟨ sequence of statements ⟩
end process P3;
```

under the following assumptions.

1. If S1 and S2 are two *signal assignment* statements such that S1 $\in$ Seq 1 $\cup$ Seq 2 and S2 $\in$ Seq' 1 $\cup$ Seq' 2, then S1.destination $\neq$ S2.destination. Thus we are ignoring resolved signals.

2. If D1, D2, and D3 are the set of declarations of *process* statements P1, P2, and P3 respectively, then D3 = D1 $\cup$ D2.

3. $\forall$ S $\in$ Seq 1 $\cup$ Seq 2 $\cup$ Seq' 1 $\cup$ Seq' 2 $\bullet$ S $\notin$ Wait-Statements $\cup$ If-Statements $\cup$ Loop-Statements $\cup$ Case-Statements.

Proof: To validate process folding, it is sufficient to show that a driver for any arbitrary signal a in the given VHDL description before folding is *similar* to the corresponding driver for a after folding. This ensures that the waveforms of all signals in the given description are preserved after the transformation. The evaluations of the three processes is shown in Figure 9.1.

Let there be an assignment to a signal a in *process* statement P1. Let the *pre-marking* set of a in P1 be ALL_TR(a, P1) and that in P3 be ALL_TR(a, P3). We first show that ALL_TR(a, P1) and ALL_TR(a, P3) are *similar* and then show that if a transaction is preempted in ALL_TR(a, P1) (due to marking),

it will also be preempted in ALL_TR(a, P3), thus proving that Driver(a, P1) is *similar* to Driver(a, P3). It is clear from the rules of the transformation that no sequential statement is deleted from the VHDL source code as a result of the transformation and that their ordering within the *process* is maintained.

Lemma 9.2.1 *If a sequential statement* S ∈ Seq 1 *is evaluated in delta interval* δ_i(j) *before the transformation, then it is evaluated in* δ_i(j) *after the transformation.*

Proof: In general, it can be shown that if a sequence of statements does not contain a *wait* statement, the entire sequence is constrained to be evaluated within a delta interval. Consider the evaluation of P1. Since the first statement of Seq 1 is evaluated in $\delta_0(0)$ (Axiom 8.1), it follows that all statements in Seq 1 will be evaluated in $\delta_0(0)$. From Axiom 8.1 it also follows that $\mathcal{E}$(W1, t_0) is `true` where t_0 is such that the relation overlaps($\delta_0(0)$, t_0) ∧ overlaps(t_0, $\delta_X(0)$) holds. Assume that S is evaluated in $\delta_{kX}(0)$ and $\mathcal{E}$(W1, t_k) is `true` (where X is the timeout value of the wait statement and the relation overlaps($\delta_k(0)$, t_k) ∧ overlaps(t_k, $\delta_{(k+1)X}(0)$) holds). It can be easily shown that S is also evaluated in $\delta_{(k+1)X}(0)$ and $\mathcal{E}$(W1, t_k) is true (where the relation overlaps($\delta_{(k+1)}(0)$, t_{k+1}) ∧ overlaps(t_{k+1}, $\delta_{(k+2)X}(0)$) holds). Hence, by induction, S is evaluated in the intervals $\delta_{kX}(0)$ (k = 0, 1,... , n,...). Since Seq′ 1 does not have any *wait* statements either, the same reasoning can be used to show that in the evaluation of P3, S is evaluated in the intervals $\delta_{kX}(0)$ (k = 0, 1,... , n,... ,).

The above proof can be easily extended to show that Lemma 9.2.1 is true for any statement S where S ∈ Seq 1 ∪ Seq′ 1 ∪ Seq 2 ∪ Seq′ 2.

Corollary 9.2.1 *For any signal* a *that is the destination for any assignment statement in* P1 *(and* P3*), the sets* ALL_TR(a, P1) *and* ALL_TR(a, P3) *are similar.*

Proof: From Lemma 9.2.1, it follows that if a transaction is posted (on ALL_TR(a, P1)) by a signal assignment with destination a in the interval $\delta_i(0)$, a corresponding transaction is posted (on ALL_TR(a, P3)) by the same assignment statement in the same delta interval. The two transactions are *similar* because they are posted in the same delta interval and have the same delay. Hence ALL_TR(a, P1) is *similar* to ALL_TR(a, P3).

Lemma 9.2.2 *If a transaction* ⟨ v, t_d, t_r ⟩ *in* ALL_TR(a, P1) *is preempted, then the corresponding transaction* ⟨ v1, $t1_d$, $t1_r$ ⟩ *in* ALL_TR(a, P3) *will also get preempted.*

Proof: Let the transaction ⟨ v, t_d, t_r ⟩ be preempted by another transaction ⟨ v′, t'_d, t'_r ⟩ in ALL_TR(a, P1). We know from Axiom 8.3 that one of the relations during(t'_d, t_d), finishes(t'_d, t_d), overlaps(t_d, t'_r), or meets(t_d, t'_r) must

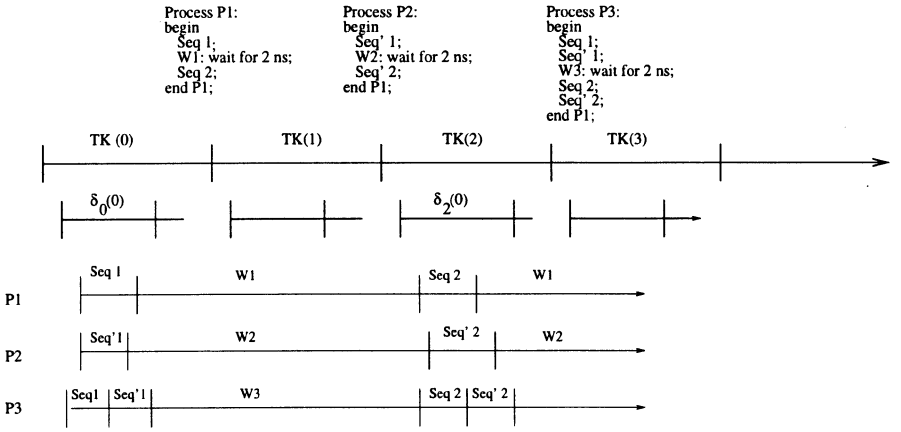


Figure 9.1. Execution of folded process

hold. Consider the case where during(t'_d, t_d) holds. If the transactions in ALL_TR(a, P3) corresponding to $\langle$ v, t_d, t_r $\rangle$ and $\langle$ v', t'_d, t'_r $\rangle$ are $\langle$ v1, $t1_d$, $t1_r$ $\rangle$ and $\langle$ v1', $t1'_d$, $t1'_r$ $\rangle$ respectively, then it can be shown that during($t1'_d$, $t1_d$) holds and therefore $\langle$ v1, $t1_d$, $t1_r$ $\rangle$ will get preempted in ALL_TR(a, P3). The same can be proven in the other three cases. A formal proof of the above result is available in Appendix A. From this result, it follows that the drivers of a in P1 and P3 are *similar*.

Corollary 9.2.2 *For any signal* a, Driver(a, P1) *is similar to* Driver(a, P3).

Since *similar* drivers lead to identical waveforms, the state space of the VHDL description is preserved by the transformation. Therefore, process folding is determined to be a valid transformation.

9.3 SIGNAL COLLAPSING

In a given VHDL description, if information flows from one signal to another, then the intermediate signals in the path can sometimes be eliminated from the simulation model if they are not referenced anywhere else in the VHDL description. Consider, for example, two *process* statements of a VHDL description as shown below.

```
P1: process                                  P2: process
begin                                        begin
S1: b <= transport f1(a) after X1 ns;        S2: c <= transport f2(b) after X2 ns;
end process P1;                              end process P2;
```

If signal b is not referenced in any other region of the VHDL description, it can be eliminated from the description by replacing *process* statements P1 and P2 with the *process* statement P3 shown below.

```
P3:  process
begin
S3: c <= transport f2(f1(a)) after X1 + X2 ns;
end process P3;
```

Such *process* statements occur frequently in common VHDL descriptions especially those that use a *dataflow* style for describing hardware or when *signal assignment* statements are used to model wire delays. The transformation (replacing P1 and P2 with P3) eliminates a signal and a *process* from the original description, thus improving simulation performance. Interestingly, the proof method used to validate this transformation exposes certain conditions under which the transformation is not valid (*i.e.*, the behavior of P3 is not equivalent to the behavior of P1 and P2 together). The proof is presented below.

Proof: The proof is presented under the following assumptions.

1. $X_1 > 0, X_2 > 0$.
2. The function f_1 is bijective.
3. There is at most one event on signal a in any TK(i). This assumption is made only to simplify the proof presented here. It turns out that the results of the proof are same without the assumption.

The proof strategy is to compare the waveform of signal c, starting with an arbitrary waveform for a, before and after the transformation. Let $\mathcal{W}(a)$ be an arbitrary waveform for a where

$$\begin{aligned}\mathcal{W}(a) = \{ & \langle \mathcal{V}^{a}_{initial}, \hat{\mathcal{T}} \rangle, \\ & \langle \mathcal{V}^{a}_{initial}, \delta_0(0) \rangle, \langle v^1_0, \delta_0(1) \rangle, \ldots, \\ & \langle v^0_1, \delta_1(0) \rangle, \langle v^1_1, \delta_1(1) \rangle, \ldots, \\ & \vdots \\ & \langle v^0_k, \delta_k(0), \ldots, \\ & \vdots \\ & \}.\end{aligned}$$

The above equation asserts that the value of a in $\hat{\mathcal{T}}$ and $\delta_0(0)$ is $\mathcal{V}^{a}_{initial}$ (where $\mathcal{V}^{a}_{initial}$ stands for a.initial-value), in $\delta_0(1)$ is v^1_0 and in general, the value of a in $\delta_p(q)$ is v^q_p. We define a set $\mathcal{EV}(a)$ to be the set of all delta intervals in which an event occurs on a. Let

$$\mathcal{EV}(a) = \{ \delta_{m_1}(n_1), \delta_{m_2}(n_2), \ldots, \delta_{m_k}(n_k) \}.$$

It follows that $v^{a}_{initial} \neq v^{n_1}_{m_1}$, $v^{n_1}_{m_1} \neq v^{n_2}_{m_2}, \ldots, v^{n_{k-1}}_{m_{k-1}} \neq v^{n_k}_{m_k}$. Further, from Assumption 3, it follows that $m_1 \neq m_2 \neq \cdots \neq m_k$. We can now determine the resulting waveforms of c before and after the transformation.

Case 1 (before transformation): In this case, the waveform of the intermediate signal b is determined first. Since *process* P1 is sensitive to a, statement S1 will be evaluated in the delta interval $\delta_0(0)$ and in the intervals $\delta_{m_1}(\mathsf{n}_1)$, $\delta_{m_2}(\mathsf{n}_2)$,..., $\delta_{m_k}(\mathsf{n}_k)$ (this can be inferred by evaluating the set EV_{P1}). Each evaluation of S1 corresponds to a transaction element in ALL_TR(b, P1). Thus,

$$\begin{aligned}\mathsf{ALL\_TR(b,\ P1)} = \{\ &\langle\ \mathcal{V}^{\ b}_{initial},\ \phi,\ \mathcal{T}\ \rangle,\ \langle\ \mathsf{f}_1(\mathcal{V}^{\ a}_{initial}),\ \phi,\ \mathsf{T}\ \rangle,\\ &\langle\ \mathsf{f}_1(\mathsf{v}^{n_1}_{m_1}),\ \phi,\ \mathsf{t}_1\ \rangle,\ldots,\ \langle\ \mathsf{f}_1(\mathsf{v}^{n_k}_{m_k}),\ \phi,\ \mathsf{t}_k\ \rangle,\ldots,\ \}\end{aligned}$$

where the relations meets($\mathcal{T}$, TK(0)), meets(T, $\delta_{X_1}(0)$), meets(t_1, $\delta_{m_1+X_1}(0)$),..., meets(t_k, $\delta_{m_k+X_1}(0)$) hold (refer Axiom 8.2). The first element of ALL_TR, $\langle\ \mathcal{V}^{\ b}_{initial},\ \phi,\ \mathcal{T}\ \rangle$ is a default element present in all *pre-marking* sets of signals which ensures that the initial value of the signal is assigned to the signal in time interval $\hat{\mathcal{T}}$. The element $\langle\ \mathsf{f}_1(\mathcal{V}^{\ a}_{initial}),\ \phi,\ \mathsf{T}\ \rangle$ is the transaction posted by the evaluation of S1 in $\delta_0(0)$. The remaining elements are transactions posted by evaluations of S1 in intervals $\delta_{m_1}(\mathsf{n}_1)$, $\delta_{m_2}(\mathsf{n}_2)$,..., $\delta_{m_k}(\mathsf{n}_k)$ respectively. Since $m_1 \neq m_2 \neq \cdots \neq m_k$, it can be determined, using the methods presented in Section 9.2, that no transaction in ALL_TR(b, P1) will be preempted. Therefore, it follows that ALL_TR(b, P1) = Driver(b, P1). Hence, the waveform of b, which can be determined directly by examining ALL_TR(b, P1) is given by

$$\begin{aligned}\mathcal{W}(\mathsf{b}) = \{\ &\langle\ \mathcal{V}^{\ b}_{initial},\ \hat{\mathcal{T}}\ \rangle,\\ &\langle\ \mathcal{V}^{\ b}_{initial},\ \delta_0(0)\ \rangle,\ldots,\\ &\vdots\\ &\langle\ \mathsf{f}_1(\mathcal{V}^{\ a}_{initial}),\ \delta_{X_1}(0)\ \rangle,\ldots,\\ &\vdots\\ &\langle\ \mathsf{f}_1(\mathsf{v}^{n_1}_{m_1}),\ \delta_{m_1+X_1}(0)\ \rangle,\ldots,\\ &\vdots\\ &\langle\ \mathsf{f}_1(\mathsf{v}^{n_k}_{m_k}),\ \delta_{m_k+X_1}(0)\ \rangle,\ldots,\\ &\vdots\\ &\}.\end{aligned}$$

Since f_1 is bijective and $\mathcal{V}^{\ a}_{initial} \neq \mathsf{v}^{n_1}_{m_1}$, $\mathsf{v}^{n_1}_{m_1} \neq \mathsf{v}^{n_2}_{m_2}$,..., $\mathsf{v}^{n_{k-1}}_{m_{k-1}} \neq \mathsf{v}^{n_k}_{m_k}$, it follows that $\mathsf{f}_1(\mathcal{V}^{\ a}_{initial}) \neq \mathsf{f}(\mathsf{v}^{n_1}_{m_1})$, $\mathsf{f}_1(\mathsf{v}^{n_1}_{m_1}) \neq \mathsf{f}(\mathsf{v}^{n_2}_{m_2})$,..., $\mathsf{f}_1(\mathsf{v}^{n_{k-1}}_{m_{k-1}}) \neq \mathsf{f}(\mathsf{v}^{n_k}_{m_k})$. Assume for the time being that $\mathcal{V}^{\ b}_{initial} \neq \mathsf{f}_1(\mathcal{V}^{\ a}_{initial})$ (Assumption 4). Then the set $\mathcal{EV}(\mathsf{b})$ is given by

$$\mathcal{EV}(\mathsf{b}) = \{\ \delta_{X_1}(0),\ \delta_{m_1+X_1}(0),\ldots,\ \delta_{m_k+X_1}(0)\ \}.$$

Since *process* P2 is sensitive to b, statement S2 is evaluated in $\delta_0(0)$ and the intervals in $\mathcal{EV}(\mathsf{b})$. Therefore, the set ALL_TR(c, P2) is given by

$$\mathsf{ALL\_TR(c,\ P2)} = \{ \langle \mathcal{V}^{\,c}_{initial},\ \phi,\ \mathcal{T} \rangle,\ \langle \mathsf{f}_2(\mathcal{V}^{\,b}_{initial}),\ \phi,\ \mathsf{T}' \rangle,\ \langle \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})),\ \phi,\ \mathsf{T}'' \rangle,\ \langle \mathsf{f}_2(\mathsf{f}_1(\mathsf{v}^{n_1}_{m_1})),\ \phi,\ \mathsf{t}'_1 \rangle, \ldots, \langle \mathsf{f}_2(\mathsf{f}_1(\mathsf{v}^{n_k}_{m_k})),\ \phi,\ \mathsf{t}'_k \rangle \}$$

where the following relations hold:
$\mathsf{meets}(\mathsf{T}',\ \delta_{X_2}(0)) \wedge \mathsf{meets}(\mathsf{T}'',\ \delta_{X_1+X_2}(0)) \wedge \mathsf{meets}(\mathsf{t}'_1,\ \delta_{m_1+X_1+X_2}(0)) \wedge \ldots \wedge \mathsf{meets}(\mathsf{t}'_k,\ \delta_{m_k+X_1+X_2}(0))$.

Case 2 (after transformation): In this case, since the value of a is directly assigned to c, we can determine the set ALL_TR(c, P3) by simply examining $\mathcal{W}(\mathsf{a})$ and $\mathcal{EV}(\mathsf{a})$. Thus,

$$\mathsf{ALL\_TR(c,\ P3)} = \{ \langle \mathcal{V}^{\,c}_{initial},\ \phi,\ \mathcal{T} \rangle,\ \langle \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})),\ \phi,\ \mathsf{T}''' \rangle,\ \langle \mathsf{f}_2(\mathsf{f}_1(\mathsf{v}^{n_1}_{m_1})),\ \phi,\ \mathsf{t}''_1 \rangle, \ldots, \langle \mathsf{f}_2(\mathsf{f}_1(\mathsf{v}^{n_k}_{m_k})),\ \phi,\ \mathsf{t}''_k \rangle \}$$

where the relations $\mathsf{meets}(\mathsf{T}''',\ \delta_{X_1+X_2}(0))$, $\mathsf{meets}(\mathsf{t}''_1,\ \delta_{m_1+X_1+X_2}(0)), \ldots,$ $\mathsf{meets}(\mathsf{t}''_k,\ \delta_{m_k+X_1+X_2}(0))$ hold.

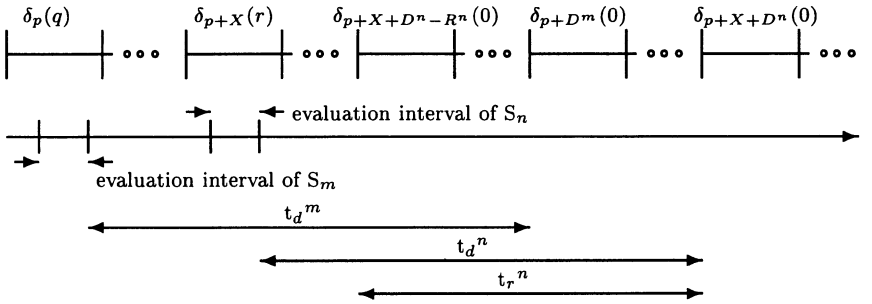
Comparing the two sets ALL_TR(c, P2) and ALL_TR(c, P3), it can be observed that the waveform of c will be the same in both cases if $\mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathcal{V}^{\,b}_{initial})$. However, in Case 1, signal c will be active in $\delta_{X_2}(0)$ (due to the presence of the transaction $\langle \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})),\ \phi,\ \mathsf{T}''' \rangle$) regardless of whether $\mathcal{V}^{\,c}_{initial}$ is equal to $\mathsf{f}_2(\mathcal{V}^{\,b}_{initial})$ or not. Hence, for the transformation, to be valid, the behavior of the VHDL description should not be dependent on the *activity* of c. For example, no reference must be made to the implicit signal c'active in the VHDL description. Returning to Assumption 4, if we now consider the case where $\mathcal{V}^{\,b}_{initial} = \mathsf{f}_1(\mathcal{V}^{\,a}_{initial})$, then $\delta_{X_1}(0) \notin \mathcal{EV}(\mathsf{b})$ and $\langle \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})),\ \phi,\ \mathsf{T}'' \rangle \notin \mathsf{ALL\_TR(c,\ P2)}$. In this case, the condition that must hold is $\mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})) \wedge \mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathcal{V}^{\,b}_{initial})$.

Summarizing, signal collapsing is a valid transformation (under the assumptions 1-3 stated earlier) if

1. $\mathcal{V}^{\,b}_{initial} = \mathsf{f}_1(\mathcal{V}^{\,a}_{initial}) \ \wedge \ \mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathsf{f}_1(\mathcal{V}^{\,a}_{initial})) \wedge \mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathcal{V}^{\,b}_{initial})$ or
   $\mathcal{V}^{\,b}_{initial} \neq \mathsf{f}_1(\mathcal{V}^{\,a}_{initial}) \wedge \mathcal{V}^{\,c}_{initial} = \mathsf{f}_2(\mathcal{V}^{\,b}_{initial})$.
2. The behavior of the VHDL description does not depend on the *activity* of signal c.

Note that in the common case where f_1 and f_2 are identity functions, condition 1 reduces to

$$(\mathcal{V}^{\,c}_{initial} = \mathcal{V}^{\,b}_{initial}).$$



S_m: a $\Leftarrow$ reject R^m inertial E1 after D^m fs
S_n: a $\Leftarrow$ reject R^n inertial E2 after D^n fs

Figure 9.2. Marking

9.4 ELIMINATION OF MARKING

Marking, as mentioned in Section 8.3, is the mechanism defined by the LRM to maintain drivers of signals. In some cases, simulation of a VHDL description can be optimized if the compiler can determine that marking is not necessary for a signal a in a *process* P. In this section, we derive conditions that are *sufficient* to ascertain that no marking is required for a in P. We say that marking is not necessary for a in P if no transaction on a gets preempted by another transaction on a in P. Equivalently, with respect to the Dynamic Model, no marking is necessary for a in P if ALL_TR(a, P) = Driver(a, P).

Recall from Section 8.3 that a transaction $\langle\ v,\ t_r,\ t_d\ \rangle$ in ALL_TR(a, P) will be preempted by another transaction $\langle\ v',\ t'_r,\ t'_d\ \rangle$ in ALL_TR(a, P) if the condition

$$\begin{aligned}&\mathsf{during}(t'_d,\ t_d)\ \vee\ \mathsf{finishes}(t'_d,\ t_d)\ \vee\\ &((\mathsf{overlaps}(t_d,\ t'_r)\ \vee\ \mathsf{meets}(t_d,\ t'_r))\ \wedge\\ &\quad (v' \neq v\ \vee\ (v' = v\ \wedge\ \exists\ \langle\ v'',\ t''_r,\ t''_d\ \rangle \in \mathsf{ALL\_TR}(a,\ P),\\ &\qquad v'' \neq v'\ \wedge\ \mathsf{overlaps}(t_d,\ t''_d)\ \wedge\ \mathsf{overlaps}(t''_d,\ t'_d))))\end{aligned}$$

is satisfied. Since it is impractical to determine the values that will be assigned to a signal during simulation we assume that $\langle\ v,\ t_r,\ t_d\ \rangle$ will be preempted by $\langle\ v',\ t'_r,\ t'_d\ \rangle$ if the stronger relation $\mathcal{R}\ (t_d,\ t'_d,\ t'_r)$ holds where

$$\begin{aligned}\mathcal{R}\ (t_d,\ t'_d,\ t'_r) \equiv\ &\mathsf{during}(t'_d,\ t_d\ )\ \vee\ \mathsf{finishes}(t'_d,\ t_d)\ \vee\ \mathsf{overlaps}(t_d,\ t'_r)\ \vee\\ &\mathsf{meets}(t_d,\ t'_r).\end{aligned}$$

Consider the evaluation of a *process* P which has one or more *signal assignment* statements with destination a (as shown in Figure 9.2). Let the evaluation of one such signal assignment (S_m) occur in δ_p(q) and of another such

assignment (S_n) occur X units later in $\delta_{p+X}(r)$. If P has only one assignment statement, then n = m. The first evaluation (of S_m) corresponds to the element $\langle v^m, t_r^m, t_d^m \rangle$ in ALL_TR(a, P) and the second evaluation (of S_n) corresponds to $\langle v^n, t_r^n, t_d^n \rangle$ in ALL_TR(a, P). From Axiom 8.2, we see that the relations $\text{meets}(t_d^m, \delta_{p+D^m}(0))$ and $\text{meets}(t_d^n, \delta_{p+X+D^n}(0))$ hold. Consider the following cases.

1. **The evaluations of S_m and S_n occur in the same delta interval**: In this case, $(X = 0 \wedge q = r)$. For marking to be eliminated, the relation $\neg\, \mathcal{R}\ (t_d^m, t_d^n, t_r^n)$ must hold. It can be inferred using interval logic that $\neg\, \mathcal{R}\ (t_d^m, t_d^n, t_r^n)$ holds if $(D^n - D^m) > R^n$ holds. This condition also implicitly states that D^n and R^n must be greater than zero.

   Interpretation: Two assignments to a can occur in the same delta interval if there are more than one assignment statements with destination a (*i.e.*, $m \neq n$) or if the assignment S_m occurs in a loop (*i.e.*, $m = n$). The two cases are discussed below.

   (a) $m \neq n$: In general, if *process* P has *signal assignment* statements S_1, $S_2, \ldots, S_k$ with destination a, delays $D^1, D^2, \ldots, D^k$ and rejection limits $R^1, R^2, \ldots, R^k$, then no preemption occurs if the following condition holds:

   $$\forall t: 1 \leq t < k \bullet (D^{t+1} - D^t) > R^{t+1}$$

   (b) $m = n$: In this case, the desired condition reduces to $(D^m - D^m) > R^m$ which is always false since pulse rejection limits are always non-negative. Hence, if a *signal assignment* statement with destination a appears in a loop statement in *process* P, marking cannot be eliminated for the driver of a in P (*i.e.*, preemption can possibly occur).

2. **The evaluations of S_m and S_n occur in different delta intervals**: In this case, the condition $(X > 0 \vee (X = 0 \wedge q \neq r))$ is `true`. Using interval logic, we can determine that $\neg\, \mathcal{R}\ (t_d^m, t_d^n, t_r^n)$ holds if $(D^n - D^m + X) > R^n$ holds. This condition can be rewritten as $X > D^m - (D^n - R^n)$.

   Interpretation: This case is related to the simulation semantics and the above condition cannot be verified by a simple syntactic check alone. From the above condition it is seen that, in the worst case, the time period between any two firings of P (*i.e.*, X) must be greater than D^m. In general, X must be greater than the largest delay of any *signal assignment* statement in P whose destination is a. This condition can be verified if some information about the sensitivity list of P is available. For example, if P is sensitive to a clock whose time period is known, then the above condition can be verified.

Summarizing the above results, no transaction preemption will occur in ALL_TR(a, P) if the following conditions hold:

1. No *signal assignment* statement with destination a occurs in a loop statement.

2. If P has *signal assignment* statements S_1, S_2,... , S_k with destination a, delays D^1, D^2,... , D^k and rejection limits R^1, R^2,... , R^k, then $\forall$ t: $1 \leq t < k \bullet (D^{t+1} - D^t) > R^{t+1}$.

3. If S_1, S_2,... , S_k are statements as in the above condition and X is the minimum time period between any two firings of P, then $X > D^m$ where $\forall$ t: $1 \leq t < k \bullet D^m \geq D^t$ and D^m is the delay of one of the statements S_1, S_2,... , S_k.

Note that the above conditions do not capture all cases where preemption does not occur. They only guarantee that no preemption will occur.

9.5 SUMMARY

In the above proofs, some of the applications of the Dynamic Model were demonstrated. The validity of process folding was proved by reasoning about time intervals that make up the driver and showing that these drivers are *similar*. In Section 9.3 we proved the validity of signal collapsing. It is important to point out here that the proof unearthed certain conditions that must hold for the transformation to be valid (see last paragraph of Section 9.3). The necessity for these conditions is not apparent by merely looking at the VHDL programs in question. Finally, in Section 9.4 it was shown that the Dynamic Model can correctly predict the conditions that must hold if no transaction on a signal will ever be pre-empted. These proofs establish results that can be used to optimize CAD tools. Process folding is already known to yield performance benefits [31]. Signal elimination reduces runtime memory required by the simulator. Finally, if a VHDL compiler can determine that no transactions for a signal can ever be pre-empted it can handle simulation events on the signal more efficiently.

10 A FRAMEWORK FOR PROVING EQUIVALENCES USING PVS

10.1 THE DYNAMIC MODEL

This chapter presents an embedding of the dynamic semitics of VHDL in PVS. The semantics is embedded by defining a set of equivalence axioms between VHDL descriptions. We present a shallow embedding of the semantics that consists of a set of functions that describe the transformations that a signal undergoes during the simulation process.

The semantics of the VHDL is defined operationally in the VHDL LRM [27] using the *simulation cycle* approach. An operational semantics of VHDL would hence be able to prove properties about specific descriptions. In order to achieve this, the simulation kernel and the elaborated set of processes and the set of drivers must be embedded in the semantics. Bickford and Jamsek [7] state that from an operational semantics it is very difficult to prove properties about the descriptions. They found that most of the proofs are *meta-level* arguments where one needs to prove things by induction on the length of the run, quantifying over all possible environments. Hence, the semantic model developed herein consists of two parts, a declarative approach and an axiomatic approach. While the declarative style allows the user to read and understand the semantics easily, the axiomatic style allows the user to reason about high-level properties in a simpler fashion. In the rest of the chapter, we describe the subset of VHDL that the dynamic model covers, define the meaning of

```
postponement_t: TYPE = {not_postponed, postponed}

process_t     : TYPE = [# postponement_v: postponement_t,
                          os_v: list[seq_stmt_t]
                       #]

vhdl_desc_t   : TYPE = setof[process_t]
```

Figure 10.1. The types corresponding to concurrent statements

equivalence of two descriptions, and the embedding of the dynamic model in PVS.

10.1.1 The VHDL Subset

In Chapter 3 constructs recognized by the static model were identified. Chapter 5 defined a reduction algebra that acted on the static model and converted it to a reduced model. The reduced model is used as input to the dynamic model and this section presents a formalization of the constructs in the reduced model and their embedding in PVS.

Concurrent Statements. The only concurrent statement recognized by the dynamic model is the process statement. A VHDL description is a set of process statements. The mathematical representation of the process statement in the reduced model is a 2-tuple given by

pr = ⟨ postponement, ordered-statements ⟩
postponement = [postponed | not_postponed]

where the variable postponement indicates whether the process statement is postponed and the variable ordered-statements represents the list of statements that are executed sequentially. The embedding in PVS models the process statement as a record containing two elements, `postponement_v` and `os_v` that correspond to the variables postponement and ordered-statements respectively. The record element `postponement_v` is a variable of the enumerated type `postponement_t`, while the element `os_v` is a variable that represents a list of sequential statements using the pre-defined abstract data type `list` and the user-defined abstract data type `seq_stmt_t`. The type, `vhdl_desc_t`, that represents a VHDL program is defined as a set of process statements. The PVS specification corresponding to the description above is shown in Figure 10.1.

Sequential Statements. In Figure 10.1, we find that the list of sequential statements in a process statement is defined using the abstract data type

```
slist_t                : TYPE = setof[signal_t]
time_t                 : TYPE = nat
severity_t             : TYPE = {note, warning, error, failure}
seq_stmt_t: DATATYPE
BEGIN
  null: null?
  wait(slist: slist_t, condition: property_t, time: time_t): wait?
  sa(destn: signal_t, prejection: nat, waveform: property_t): sa?
  va(destn: variable_t, expression: property_t): va?
  if_(condition: property_t, then_: list[seq_stmt_t],
         else_: list[seq_stmt_t]): if_?
  loop(condition: property_t, os: list[seq_stmt_t]): loop?
END seq_stmt_t
```

Figure 10.2. The types corresponding to static constructs of VHDL

`seq_stmt_t`. The sequential statements that are recognized by the reduced model of VHDL shown below using the BNF notation.

sequential statement ::=
wait statement
| signal assignment statement
| variable assignment statement
| if statement
| loop statement
| null statement

The tuple representation presented in Chapter 3 is supported using the definition of `seq_stmt_t` as shown in Figure 10.2. The wait statement is generated using the constructor `wait` that takes as arguments a sensitivity list, `slist`, a condition to wait on, `condition`, and a timeout clause, `time`. Note that the sensitivity list is represented as a set of signals using the type `slist_t` and time is kept track using the pre-defined type `nat`. Some simulation researchers may contend that this must be modified to be a subrange of `nat`, from `0` to `TIME'HIGH`. However, as formal methods researchers we feel that the maximum value of time, `TIME'HIGH` is purely an implementation oriented constant and hence the need for it is not felt in the formal representation.

10.1.2 Equivalence

Van Tassel [50] defines equivalence of VHDL descriptions based on the user suppling some signals of interest. In this effort two VHDL descriptions are equivalent if (i) the same set of input signals is provided to both the descriptions, and (ii) they produce the same waveforms for the set of user-specified signals. More precisely, the effective values of these signals are identical for both

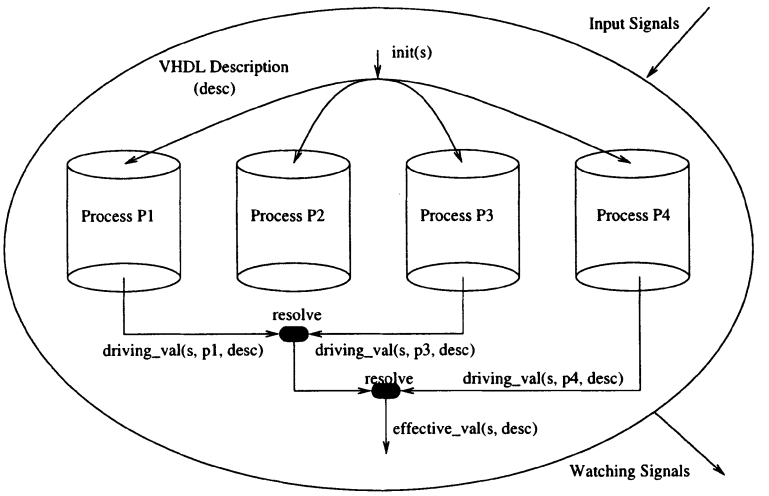


Figure 10.3. A signal value passes through a description

the descriptions. However, there is an inherent problem with this definition: the second condition cannot be verified since (a) the computation of waveforms from time 0 to the maximum value of the VHDL type `TIME`, `TIME'HIGH`, requires simulation, and (b) the comparison of two such waveforms is computationally complex.

This problem can be overcome by defining a canonical form for a waveform on a signal. The second condition can be restated as: "*they produce the same canonical form on a set of user-specified signals.*" A canonical form for the waveform can be obtained by evaluating the sequence of transformations on the initial value of the signal. The steps performed to obtain the canonical form are best explained using an example.

Figure 10.3 shows a VHDL description, `desc`, containing four process statements. Process statements `p1`, `p3`, and `p4` contain signal assignments with the target `s`, while the process statement `p2` does not. Hence, `p2` does not contribute to the canonical form of the waveform of signals `s`. The process statement `p1` contributes `driving_val(s, p1, desc)` (also called a *property*) towards the canonical form. The properties `driving_val(s, p1, desc)`, `driving_val(s, p3, desc)`, and `driving_val(s, p4, desc)` are resolved as shown in Figure 10.3 to obtain the canonical form, `effective_val(s, desc)`. Although the function `resolve` is similar to resolution functions in VHDL, it is not a resolution function. It captures the functionality of the resolution function through a sequence of calls. Using this approach, we can show that if a resolution function behaves differently for different permutations of the input, the description will produce erroneous results.

Essentially, if the same set of signals are provides to both the descriptions and the canonical form for the effective values of all the user-specified signals

are the same in both the descriptions, the two descriptions are equivalent. In addition, the abstraction used for timing follows a linear time semantics and the actual values on the outputs is used to check equivalence. The values at the delta cycles *need not be the same for the signals.*

10.1.3 Dynamic Model Embedding

The dynamic model semantics, partly explained in this section has been created mainly for the purpose of proving the equivalence of two VHDL descriptions.[1] The embedding in PVS, hence, contains an axiom that defines the equivalence of VHDL descriptions. As defined in Section 10.1.2, the dynamic model of two descriptions has been defined with respect to a set of signals that are being watched. The axiom that defines the equivalence is defined in PVS as

```
equiv_1_ax: AXIOM
     equiv(desc_1, desc_2, is_1, is_2, ws)
         =  ((is_1 = is_2)
          & (FORALL (s: (ws)):
               effective_val(s, desc_1) = effective_val(s, desc_2))).
```

In the above axiom the function `equiv` tests the two VHDL descriptions `desc_1` and `desc_2` for equivalence with respect to the set of signals `ws`. The set of signals input to the description `desc_1` is `is_1` and the set of signals input to the description `desc_2` is `is_2`. The axiom asserts that the set of signals input to both the descriptions must be the same, and the effective value of all the signals that are being watched is the same in both the descriptions.

Effective Values. The VHDL LRM (§12.6.2, L481) defines the effective value of a signal to be "the value obtainable by evaluating a reference to the signal within an expression." The effective value of a signal is computed using the driving values contributed by every process statement that drives the signal. If a process statement does not drive a signal it is ignored in the computation of effective value of the signal. The axioms that formalize this is shown below.

```
effective_val_hi_empty: AXIOM
  (FORALL (desc: (sets[process_t].empty?)):
          effective_val_hi(s, desc, desc_1) = null)

effective_val_hi_add: AXIOM
  effective_val_hi(s, add(p, desc), desc_1) =
    IF drives?(os_v(p), s)
    THEN mf_val(s, resolve(driving_val_h(s, os_v(p), init(s), desc_1),
                  effective_val_hi(s, desc, desc_1)))
    ELSE effective_val_hi(s, desc, desc_1)
    ENDIF
```

The effective value is determined using a constructive definition. The constructors for the set of processes are `emptyset` and `add`. The empty set of processes does not contribute to the effective value; more precisely, it contributes a `null` value (which is later ignored) to the effective value. When the effective value is computed on a non-empty set of process statements (`add(p, desc)`), two cases arise. If the process statement `p` does not drive the signal `s`, then the effective value in the rest of the description is considered. If the process drives the signal, then the driving value of the signal by the process is resolved with the effective value obtained from the rest of the description. The function `mf_val` has been introduced to detect changes in the value of the effective value of the signal to provide support for the incorporation of the `'event` attribute.

Driving Values. The VHDL LRM defines the driving value of a signal as "the value that the signal provides as a source of other signals." (§12.6.2, L480) The driving value for any signal can be determined from the driving value of its *sources*. The sources for a signal are the ports that are connected to it and drivers associated with it as a result of signal assignments made to it.

More precisely, the driver for a signal `s` is present in every process statement that assigns a value to the signal through a signal assignment statement, and in every process statement that assigns a value to the formal part of the port association for which the signal `s` is the actual. The function that calculates the canonical form of the driving value is shown in Figure 10.4. The canonical form of the driving value is calculated across all the sequence of sequential statements in every process statement. More precisely, the driving value of the signal `s`, contributed by every process statement that drives the signal `s` is computed by *passing the initial value of the signal through* the set of sequential statements of the process. Hence, the function `driving_val_h` takes four parameters: (i) `s`, the signal for which the driving value needs to be computed, (ii) `os`, the sequence of sequential statements across which the signal needs to be passed through., (iii) `pr`, the property before the first element in the sequence `os`,[2] and (iv) `desc`, the description in which the driving value of the signal `s` is calculated.

The first case shown in Figure 10.4 describes the effect of a signal assignment statement whose destination is the same as the signal `s`. In this case, the property of the signal is modified using the function `seq_prop` and the modified property is passed as the new property for the recursive computation using the `driving_val_h` function. However, in the second case when the signal assignment statement does not assign to the signal `s`, the statement is ignored and the canonical form of the driving value given by the rest of the sequential statements is computed. Since we do not handle type conversions at the port associations we need to check only for these two cases. Two more cases that deal with the actual associated with the formal to which a signal assignment is made need to be added, if the type conversions at port associations are handled.

The wait statement contributes a non-null property to the canonical form for the driving value of the signal `s`. The if statement contributes a property:

```
driving_val_h(s, os, pr, desc):
  RECURSIVE property_t = CASES os OF
    null: pr,
    cons(x, y):
      COND
      sa?(x) & destn(x) = s
      -> driving_val_h(s, y, append(seq_prop(s, x, desc), pr), desc),

      sa?(x) & NOT destn(x) = s
      -> driving_val_h(s, y, pr, desc),

      wait?(x)
      -> driving_val_h(s, y, append(seq_prop(s, x, desc), pr), desc),

      if_?(x)
      -> driving_val_h(s, y, append(seq_prop(s, x, desc), pr), desc),

      loop?(x)
      -> driving_val_h(s, y, append(seq_prop(s, x, desc), pr), desc),

      va?(x)
      -> driving_val_h(s, y, pr, desc),

      null?(x)
      -> driving_val_h(s, y, pr, desc)

      ENDCOND
    ENDCASES
  MEASURE length(os)
```

Figure 10.4. The definition of the function `driving_val_h`

however, this property may be null if none of the sequential statements present in the *then* and the *else* clauses contribute to a change in the driving value. In a similar fashion, if all the sequential statements that form a part of the loop statement do not contribute to a change in the driving value, a null value is contributed by the loop statement.

Earlier, axioms were defined to assert that for a signal that has no source its driving value is the initial value for all the time. The first axiom asserted that for an arbitrary signal `s`, if there is no signal assignment statement whose destination is `s`, and if `s` is not associated with any port, it has no source. The second axiom asserted that for every signal that has no source the driving value is the same as the initial value. In this effort, this property can be easily derived from the given axioms. Thus, the axioms that define the driving values and

```
drives?(os, s): RECURSIVE bool =
   CASES os OF
   null: false,
   cons(x, y):
   COND
    sa?(x) & destn(x) = s  -> true,
    sa?(x) & destn(x) /= s -> drives?(y, s),
    wait?(x)               -> drives?(y, s),
    if_?(x)          -> drives?(then_(x), s) OR
                        drives?(else_(x), s) OR
                        drives?(y, s),
    loop?(x)         -> drives?(os(x), s) OR drives?(y, s),
    va?(x)           -> drives?(y, s),
    null?(x)         -> drives?(y, s)
   ENDCOND
   ENDCASES
   MEASURE total_length(os)
```

Figure 10.5. The definition of the function `drives?`

the effective values allow the user to infer more properties rather than stating all the properties that can be inferred, as axioms.

An interesting function that has not yet been discussed is `drives?` (Figure 10.5). This function determines, if a signal `s`, is driven by a list of sequential statements `os`. The result is computed by cases. If the list is empty, it does not drive the signal. If the list is non-empty (`cons(x, y)`), the head of the list, `x`, can be one of the six sequential statements in the reduced model. Hence, the analysis splits into cases on the type of the statement `x`. If `x` is a signal assignment statement with the destination `s`, then it drives the signal, and hence returns a value of `true`. If the statement `x` is a signal assignment statement, but with a destination that is not `s`, or if it is one of wait statement, variable assignment statement, or null statement the result is `drives?(s, y)`. More precisely, if the statement `x` does not drive the signal `s`, it is the result of the driving value computed using the rest of the list. Furthermore, if the statement `x` is an if statement, then the result is computed using the statements in the then and else clauses of the if statement and the rest of the statements in the list. A similar computation is carried out for loop statements. We see that the definition of the function has been specified in an intuitively obvious manner and the case analysis is automatically carried out by the abstract data type facility in PVS. For more information on the dynamic model the interested reader is encouraged to read the specification and the proofs from the WWW [48].

10.2 VALIDATION OF THE SEMANTICS

The evolution of a semantics is similar to that of software. The software life-cycle consists of requirements analysis, design, construction, testing, installation, and maintenance [46]. During requirements analysis, the needs of the project that a computer system must meet are identified. In the development of semantics, the constraints of the language, as specified in the Language Reference Manual, are captured formally. During the design phase, the software engineers decide how to create the software; they start with a specification, decide the target medium, and identify whether the system meets the implicit and explicit requirements of the target medium. In the development of semantics, a formal/informal specification of the language is prepared. The developers decide the technique to be used for specifying the semantics, such as an axiomatic specification or a denotational specification. In addition, the developers decide the target medium to be used for the specification, such as PVS, HOL, or an EA-machine. During the testing stage, a software is tested to check whether it meets the requirements defined earlier in the life-cycle. In semantics development, it is difficult to verify all the constraints present in the Language Reference Manual. This chapter provides an overview of the steps taken to validate the semantics developed for VHDL. Maintenance of the software and the semantics are made as changes in the requirement are identified.

10.2.1 Validation

Semantics validation is the process of testing the functionality and the correctness of the semantics. Semantics validation is performed primarily for two reasons: (1) defect detection, and (2) reliability estimation. The problem of applying a semantics to defect detection is that the semantics can only suggest the presence of faults, and not their absence (unless the testing is exhaustive). The problem of applying semantics testing to reliability estimation is that the inputs needed for testing may be flawed. In both the cases, the mechanism that determines whether the output is correct is impossible to develop. The key to semantics testing is finding the minimum set of failure cases — something that requires a deep analysis of the semantics itself. For most languages this is computationally infeasible, since it requires that all the features of the languages be tested for all possible inputs. It is hence, commonplace to test as many features of the language as possible (within a set of resource constraints).

Kloos and Breuer [18] note that, for a semantics of VHDL to be considered correct/acceptable, it has to be validated against Van Tassel's semantics [50] or another strongly established semantics. However, note that since the semantics presented in this effort has a greater coverage than Van Tassel's semantics, we are only able to validate the semantics on the results of the common portion. Kloos and Breuer [18] assert that the tested or proven agreement on the common portion confers confidence in the overall correctness. Hence, in this effort we consider the case studies made by Van Tassel and prove them to be correct using our semantics.

```
process (A, B)
begin
  C <= A nand B after 1 ns;
end process;
```

```
process (A, B)
begin
  TMP <= A and B after 1 ns;
end process;

process (TMP)
begin
  C <= not TMP after 0 ns;
end process;
```

Figure 10.6. Two equivalent specifications for a NAND gate

All the studies followed the following general pattern:

1. Specify in VHDL, a high-level behavioral description of the device.
2. Specify in VHDL, a low-level implementation level description of the same device.
3. Identify a set of signals that we need to watch for outputs.
4. Provide as input to the translator the results of steps 1, 2, and 3. The translator checks them for syntactic validity and generates a PVS theorem (a verification condition) that needs to be proven for the two descriptions to be equivalent.
5. Using the semantics and the set of existing results, prove the required theorem.

10.2.2 NAND gate

To demonstrate the work with operational semantics and how zero delay assignments are dealt with, Van Tassel uses a NAND gate specification as an example and proves the equivalence of the behavior and implementation of the gate. The behavioral specification uses the VHDL primitive boolean operator **nand**, while the implementation specifies an AND gate and a NOT gate working together. More precisely, the equivalence of the two descriptions (where `A`, `B`, `C`, and `TMP` are boolean signals) shown in Figure 10.6 with respect to the signal `C` is demonstrated. Although the equivalence is obvious, the example illustrates that,

1. The translator is capable of handling multiple process statements with sensitivity lists,

```
process (A, B)
begin
 C <= not(A and B) after 1 fs;
end process;
```

```
process (A, B)
begin
 C <= not A or not B after 1 fs;
end process;
```

Figure 10.7. The negation of a conjunction is a disjunction of negations

2. The translator is able to generate theorems that take care of the type of the signals (in this case `boolean`),
3. The semantics is capable of handling inertial delays,
4. The semantics handles zero delayed assignments, and
5. The semantics defines rules for the operators on the boolean type, such as `nand`, `not`, and `and`.

10.2.3 DeMorgan Property

DeMorgan property is a logical theorem that asserts that the complement of a conjunction is the disjunction of the complements and vice-versa [54]. In mathematical notation,

$$\neg(\mathcal{A} \wedge \mathcal{B}) = \neg\mathcal{A} \vee \neg\mathcal{B} \tag{10.2}$$

$$\neg(\mathcal{A} \vee \mathcal{B}) = \neg\mathcal{A} \wedge \neg\mathcal{B}. \tag{10.3}$$

Negation of a conjunction is a disjunction of negations. This example specifies in VHDL, the DeMorgan property shown in the Equation 10.1, and demonstrates the equivalence. The motivation behind this example is to show that it is possible to prove the equivalence of different architectures that implement the same property in different ways. It specifies that the negation of a conjunction of two boolean terms is equivalent to the disjunction of the negation of each of the terms. Presented in Figure 10.7 are the two descriptions that were proven to be equivalent with respect to the signal `C`.

Negation of a Disjunction is a Conjunction of Negations. This example specifies in VHDL, the DeMorgan property shown in the Equation 10.2, and demonstrates the equivalence. The motivation behind this example is to demonstrate the capabilities of the semantics, such as handling logical operators and process statements with sensitivity lists. Figure 10.8 presents the two process statements that were shown to be equivalent with respect to the signal `C`.

```
process (A, B)
begin
 C <= not(A or B) after 1 fs;
end process;
```

```
process (A, B)
begin
 C <= not A and not B after 1 fs;
end process;
```

Figure 10.8. The negation of a disjunction is a conjunction of negations

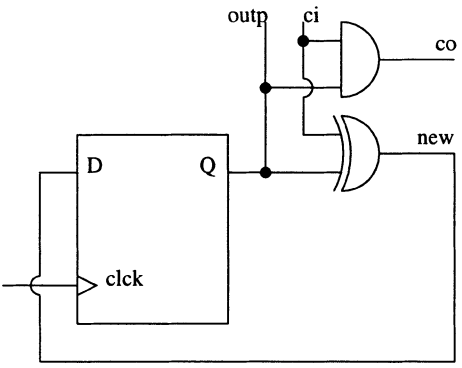


Figure 10.9. The implementation level circuit of the counter cell

10.2.4 Counter Cell

This example is used to demonstrate the notion of a state in a circuit. The circuit under consideration is one cell of a binary counter. The cell consists of four signals: a clock input, `clck`, a carry input, `ci`, a carry output, `co`, and an output, `outp`, that reflects the current state of the counter. The device is sensitive to the rising edge of the clock and the cell counts only when the carry input is high. The carry output must be set to high when the cell counts up to a high value. The equivalence of the behavior of a counter cell, and its implementation using a D-type flip-flop, an AND gate and an XOR gate (Figure 10.9) has been demonstrated. Note that the third process statement in the implementation level description, the specification of the D-flip flip, contains an `if-then-else` statement. The goal is to prove that the two descriptions shown in Figure 10.10 are equivalent with respect to the signals `co` and `outp`.

The behavior of the cell presents an algorithmic view of the function of the cell and is intuitively obvious. In the implementation level circuit, the central component is the D-type flip flop. There is a delay of one femto-second from the D-input to the Q-output, when the rising edge of the clock, `clck`, is detected. The conjunction of the output of the D-flip flop along with the input carry, `ci`, produces the carry output. The output and the carry input are also passed though an XOR gate to generate the next value for the input to the D-flip flop. It is assumed that the logic gates have zero delay.

```
process (ci, outp)
begin
  co <= ci and outp;
end process;

process (clck)
begin
 if (clck and ci) then
   outp <= not outp after 1 fs;
 end if;
end process;
```

```
process (ci, outp)
begin
  co <= ci and outp;
end process;

process (ci, outp)
begin
  newp <= ci xor outp;
end process;

process (clck)
begin
  if (clck) then
    outp <= newp after 1 fs;
  end if;
end process;
```

Figure 10.10. Two equivalent specifications of a counter cell

As noted before, the goal is to prove the equivalence with respect to the signals `co` and `outp`. The process statement that generates the value for the signal `co` is similar in both cases, and hence the equivalence with respect to `co` is simple to achieve — the signal `co` is generated in both the cases by the conjunction of the signals `ci` and `outp`. The difficulty arises while proving the equivalence with respect to the signal `outp`. The proof is carried out in PVS by case analysis on all the input values (true, false) that each of the inputs can take. It is worth noting at this point that the semantics is capable of handling state based descriptions and `if-then-else` statements.

In addition, this example demonstrates that the semantic embedding can be used to perform synthesis of hardware. Currently, synthesis from VHDL is one of the more important applications of the language with high user demand. When VHDL is used for synthesis, there is a difference between the results from simulation before synthesis and the results from simulation after synthesis. During simulation before synthesis, the intended behavior of the circuit is validated, and during simulation after synthesis the behavior of the resultant hardware is determined. If there is a discrepancy between these two behaviors, it would be extremely dangerous and can lead to incorrect implementations. However, using the formal framework developed, equivalence between the basic constructs used for synthesis can be checked for equivalence. This leads to the production of high-quality synthesis software and circuits with a very high degree of reliability.

```
process (clk)
begin
  if (clk) then
     output <= output xor input;
  end if;
end process;
```

```
process (clk)
begin
  if (clk) then
    if (output) then
      output <= not input;
    else
      output <= input;
    end if;
  end if;
end process;
```

Figure 10.11. Two equivalent specifications of a parity checker

10.2.5 Parity Checker

This example is used to illustrate how initialization is done and other subtleties of the semantics. The basic premise is to have a device with one input and one output which satisfies the following two constraints :

- The output at time 0 is high.
- The output is high when an even number of highs have been received on the input.

The derivation begins as in the earlier examples, with a program specifying the behavior of the device in terms of a high-level VHDL design. The parity checker consists of four signals : a clock input, `clk`, an input, `input`, and an output, `output`, that reflects the current output of the parity checker. The parity checker is sensitive to the rising edge of the clock. In the behavioral specification, the output is initialized to 1 and it uses the VHDL primitive boolean operator `xor` where the two operands are the initial value of `output` and `input`, while the implementation specifies using `if-then-else` construct to check whether even number of highs have been received as input. As noted before, The goal is to prove that the two descriptions shown in Figure 10.11 are equivalent with respect to the signal `output`. The proof is carried out in PVS similar to the approach described in the counter cell example.

Thus this parity checker example illustrates that the semantics supports nested `if-then-else` statements. In addition, this example demonstrates how initialization is dealt with. It also indicates that the semantics can be used in hardware synthesis.

10.3 DEVELOPING PROOFS OF OPTIMIZATIONS

Parallel Discrete Event VHDL Simulation (PDEVS) is the execution of a single discrete event VHDL simulation program on a parallel computer [20]. In

PDEVS, sub-assemblies of simulation (also known as *logical processes*) operate autonomously and exchange time-stamped event messages. VHDL descriptions contain substantial amounts of parallelism; yet, it is surprisingly difficult to parallelize them in practice. This difficulty arises from the *fine grained* nature of the logical processes. The logical processes generated for VHDL simulations are fine-grained since the individual tasks to be performed by a process statement is relatively small in terms of code size and execution time. Although, a large number of fine-grained logical processes appears to be a good supposition for high speed-ups, the amount of message passing between the logical processes becomes high causing a reduction in the effective simulation speed. Hence, there is a need to generate *coarse-grained* logical processes, as opposed to fine-grained logical processes, for simulating VHDL descriptions. This can be performed by combining VHDL process statements. Another way to effectively increase the simulation speed is by eliminating unwanted signals.

In the remainder of this chapter we informally describe two optimizations to the simulation process: (1) Process folding and (2) Signal collapsing, and present a brief outline of the proofs for both.

10.3.1 Process Folding

Informal Specification. Process folding [31] is an optimization that combines two VHDL process statements into a single VHDL process statement. Figure 10.12 presents two VHDL descriptions that are shown to be equivalent under certain conditions described later in this section. The first description consists of a set of processes $\mathcal{D}$and two processes $\mathcal{P}$ and $\mathcal{Q}$. The second description consists of the same set of processes, $\mathcal{D}$, and the process $\mathcal{PQ}$. The process $\mathcal{PQ}$ contains more code than both the processes $\mathcal{P}$ and $\mathcal{Q}$ and executes for a time (real time) longer than each of the processes. The folding of the processes $\mathcal{P}$ and $\mathcal{Q}$ as the process $\mathcal{PQ}$ results in the increase in the granularity of simulation. Studies performed by McBrayer and Wilsey [31] show that there is a speed-up by a factor of 2.2. Process folding asserts that two processes $\mathcal{P}$ and $\mathcal{Q}$ can be folded into a process $\mathcal{PQ}$ (Figure 10.12) if

(i) The processes $\mathcal{P}$ and $\mathcal{Q}$ are not postponed processes,

(ii) The processes $\mathcal{P}$ and $\mathcal{Q}$ contain no resolved signals,

(iii) The wait statement W, contains only a timeout expression, and

(iv) All the sequential statements in OS_1, OS_2, OS_3, and $OS_4$are *no-wait blocks.* More precisely, they could contain signal assignment statements, variable assignment statements, null statements, loop statements that do not contain a wait statement, and if statements that do not contain wait statements in their *then* and *else* blocks.

The validity of this transformation can be argued informally. However, we cannot be certain that the conditions stated above are necessary and sufficient;

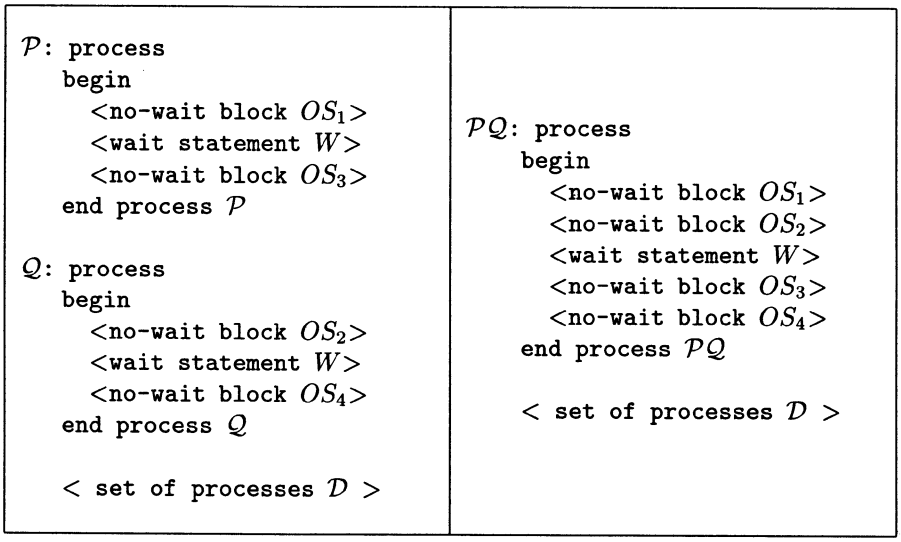


Figure 10.12. Description of process folding

Table 10.1. The cases generated during the proof

Sl. No.	s present in destination		
	$\mathcal{P}$	$\mathcal{Q}$	$\mathcal{PQ}$
1.	√	√	√
2.	√	√	×
3.	√	×	√
4.	√	×	×
5.	×	√	√
6.	×	√	×
7.	×	×	√

prior to this effort, the only evidence for the correctness of process folding was an informal analysis and empirical testing.

Formal Verification. The objective of the verification activity is to show that the process folding optimization results in the same canonical form of waveforms on all the signals that are being watched under the conditions given in Section 10.3.1. Consider an arbitrary signal s that is being watched as a signal of interest. Since the signal s should be present in the destination of a signal assignment statement in either the process $\mathcal{P}$, $\mathcal{Q}$, or $\mathcal{PQ}$, seven cases arise as shown in Table 10.1. In cases 1 and 2, the signal s is present in the destination of the processes $\mathcal{P}$ and $\mathcal{Q}$. Hence, it is a resolved signal. However,

our premise is that there are no resolved signals between the processes $\mathcal{P}$ and $\mathcal{Q}$ (condition (ii)). This leads to a contradiction and the goal is proved. In cases 4 and 6, the signal is present in the destination of one of $\mathcal{P}$ or $\mathcal{Q}$, but is not present in the process $\mathcal{PQ}$. This is impossible, since every signal assignment statement present in the process $\mathcal{P}$ (or $\mathcal{Q}$ as the case may be) is also present in the process $\mathcal{PQ}$. Hence, a contradiction exists and cases 4 and 6 are discharged. Case 7 asserts that there is a signal assignment statement in the process $\mathcal{PQ}$ with **s** as the destination, while no such assignment is present in $\mathcal{P}$ or $\mathcal{Q}$. This, however, cannot be true, since any signal assignment statement in the process $\mathcal{PQ}$, must either be present in the process $\mathcal{P}$, or the process $\mathcal{Q}$; Case 7 is thus discharged. The proofs of cases 3 and 5 are complicated and can be found elsewhere [48].

Proof embedding in PVS. This section deals with the embedding of the proof of process folding in PVS, to give the reader an idea of how a proof is performed in PVS. The initial sequent for the proof of process folding is shown below.

```
process_fold :

  |-------
{1} (FORALL (desc: vhdl_desc_t, is_1: setof[signal_t],
             os_1: list[seq_stmt_t], os_2: list[seq_stmt_t],
             os_3: list[seq_stmt_t], os_4: list[seq_stmt_t],
             t: time_t, ws: setof[signal_t]):
    LET p_1: process_t
        = (# postponement_v := not_postponed,
             os_v := append(append(os_1, cons(wait(emptyset, TRUE, t),
                                              null)), os_3)
             #),
        p_2: process_t
        = (# postponement_v := not_postponed,
             os_v := append(append(os_2, cons(wait(emptyset, TRUE, t),
                                              null)), os_4)
             #),
        p_3: process_t
        = (# postponement_v := not_postponed,
             os_v := append(append(append(append(os_1, os_2),
                                          cons(wait(emptyset, TRUE, t),
                                               null)),
                                    os_3), os_4)
             #)
      IN no_waits(os_1) & no_waits(os_2)
           & no_waits(os_3) & no_waits(os_4) & no_resolved(p_1, p_2)
       =>
       equiv(add(p_1, add(p_2, desc)), add(p_3, desc), is_1, is_1, ws))
```

To get a better understanding of the proof sequent, we quickly recapitulate that a process statement is a record that contains two variables, namely, `postponement_v` and `os_v`. The variable `postponement_v` indicates whether a process statement is a postponed process statement, and `os_v` denotes the list of sequential statements present in the process statement. The variables `os_1`, `os_2`, `os_2`, and `os_3` represent the *no-wait blocks* OS_1, OS_2, OS_3, and OS_4respectively. The variables `p_1`, `p_2`, and `p_3` denote the process statements $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{PQ}$ respectively. The constructor for wait statements, `wait`, generates the wait statement `wait(emptyset, TRUE, t)`, that corresponds to the wait statement Wdefined in Section 10.3.1. The conditions generated using the function call `no_wait` assert that the list of sequential statements are *no-wait blocks*, and the boolean statement `no_resolved(p_1, p_2)` asserts that there are no resolved signals between the processes `p_1` and `p_2`. The set `ws` denotes the set of signals that are being watched. In this case, it is universally quantified, since the process folding optimization has to hold regardless of the set of signals that are being watched. As noted in Section 10.1 the sequent splits into seven sub-goals. The first sub-goal is shown below.

```
process_fold.1 :

{-1}    drives?(os_v(p2), s!1)
{-2}    drives?(os_v(p1), s!1)
{-3}    drives?(os_v(p3), s!1)
[-4]    no_waits(os!1)
[-5]    no_waits(os!2)
[-6]    no_waits(os!3)
[-7]    no_waits(os!4)
[-8]    no_resolved(p1, p2)
  |-------
{1}    mf_val(s!1, ...)  =  mf_val(s!1, ... )
```

The arbitrary signal that is being watched is `s!1`. The first two antecedents (`drives?(os_v(p2), s!1)` and `drives?(os_v(p1), s!1)`) assert that the signal `s!1` is driven by the processes $\mathcal{Q}$ and $\mathcal{P}$ respectively. Furthermore, the eigth antecedent (`no_resolved(p1, p2)`) asserts that there are no resolved signals between the processes `p1` and `p2`. Hence, using the definition of the function `no_resolved` (shown below) the sub-goal is proven. The proof of other subgoals is complicated and the interested reader is encouraged to read the proofs from the World Wide Web [48].

```
no_resolved(p_1, p_2): bool =
  (FORALL (s: signal_t):
      NOT(drives?(os_v(p_1), s) & drives?(os_v(p_1), s)))
```

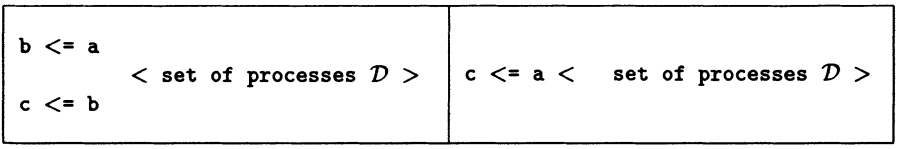


Figure 10.13. Description of signal collapsing

10.3.2 Signal Collapsing

Informal Specification. Signal Collapsing is an optimization that combines two conditional concurrent signal assignment statements into one by eliminating an intermediate signal. Figure 10.13 presents two VHDL descriptions that are equivalent under certain conditions. The first description consists of a set of processes $\mathcal{D}$and two concurrent signal assignment statements, `b <= a` and `c <= b`. The second description consists of the same set of processes, $\mathcal{D}$, and the concurrent signal assignment statement, `c <= a`. During the proof process for validating the equivalence the following conditions were derived.

(i) The signal `b` must not be a resolved signal, *i.e.,* no process in the description $\mathcal{D}$should drive the signal `b`,

(ii) The actual signals connected to the these via an association must be different. More precisely, `a /= b`, `a /= c` and `b /= c`, and

(iii) The signal `b` must not be used by any process in $\mathcal{D}$. More precisely, the signal `b` should not drive any other signal or be a part of a sensitivity set in a wait statement.

(iv) The signal `b` is not present in the set of signals that is being watched for outputs.

An interesting point to be noted is that the concurrent signal assignments are converted into their corresponding process statements in the reduced model before the folding is done. After the folding has been completed, a process statement which is equivalent to the signal assignment statement `c <= a` is generated.

Formal Verification. Figure 10.13 presents a description of signal collapsing. It was noted in Section 10.3.2 that the concurrent signal assignment statements are converted into their corresponding process statements before the collapsing is done. Hence, let $\mathcal{P}$ denote the process statement generated by rewriting the concurrent signal assignment statement `b <= a`, $\mathcal{Q}$ denote the process statement generated by rewriting the concurrent signal assignment statement `c <= b`, and $\mathcal{PQ}$ denote the process statement generated by rewriting the concurrent signal assignment statement `c <= a`.

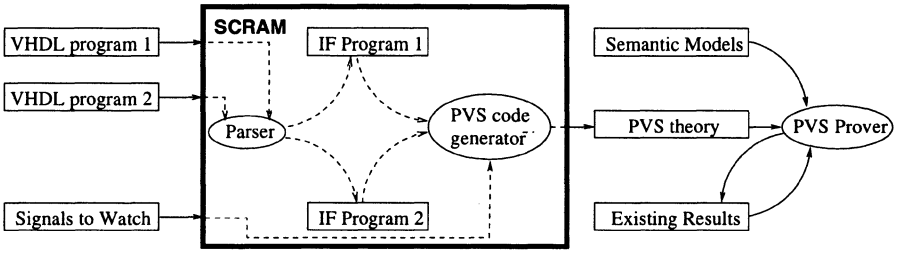


Figure 10.14. The framework illustrated

The cases generated during the proof process are similar to the cases generated for process folding, although not in the same order. However, we will use Table 10.1 as an enumeration for the cases generated since the order does not matter in the discussion presented here. In cases 1 and 2, the processes $\mathcal{P}$ and $\mathcal{Q}$ drive the signal s. However, we know that $\mathcal{P}$ drives the signal b, $\mathcal{Q}$ drives the signal c and the signals b and c are not the same (condition (ii)). Hence a contradiction is established. In case 3, the processes $\mathcal{P}$ and $\mathcal{PQ}$ drive the signal s. However, $\mathcal{P}$ and $\mathcal{PQ}$ cannot drive the same signal since $\mathcal{P}$ drives the signal b and $\mathcal{PQ}$ drives the signal c, and the signals b and c are not the same (condition (ii)). Hence a contradiction is established and the proof sub-goal is discharged. In case 4, the signal s is present only in the process $\mathcal{P}$. This means that the signal s is the signal b, since b is the only signal driven by the process $\mathcal{P}$. However, this is a contradiction since the signal s is one of the signals present in the set of signals that are being watched, while the signal b is not present in the set of signals that are being watched (condition (iv)). In cases 6 (or 7), the signal being watched is present in $\mathcal{Q}$ (or $\mathcal{PQ}$) and not present in $\mathcal{PQ}$ (or $\mathcal{Q}$). However, the only signal driven by $\mathcal{Q}$ (or $\mathcal{PQ}$) is c and the only signal driven by the process $\mathcal{PQ}$ (or $\mathcal{Q}$) is also c. Hence there cannot exist such an s. This leads to a contradiction and the sub-goal is discharged. The proof for case 5 is complicated and can be found elsewhere [48].

10.4 APPLICATIONS TO PRACTICAL USE

A static and a dynamic model for VHDL embedded in PVS was proposed in Chapter 3 and Section 10.1 and the validity of the models was demonstrated in Section 10.2. The specification of the static and dynamic models in PVS allows the user to reason about equivalences of classes of VHDL descriptions. More precisely, the semantic models accept specifications of VHDL patterns and prove their equivalence; equivalence proofs of specific instances naturally follow. However, there is a fundamental problem in using the embedding of the semantics in PVS — the VHDL source descriptions to be checked for equivalence need to be specified in the formal notation that PVS accepts. If an error happens during the conversion of the VHDL description into the PVS language, the purpose behind establishing a formal semantics would be defeated and the

scalability with respect to the size of the designs would be minimal. It is worth noting at this point that the translation process is not simple. It needs to perform

- syntactic validation of the source code,
- capture the type correctness conditions in the input,
- well-formedness checking of the VHDL source, and
- transform VHDL constructs from the static model into the reduced model, using the reduction algebra.

Since all the above operations are non-trivial a translator has been built to convert VHDL descriptions into the PVS language.

10.4.1 The Translator

The translator can be broadly divided into two parts. The front-end parser that performs syntactic and static semantic correctness and converts the VHDL description to an Internal Intermediate Representation (IIR), and the back end code-generator that publishes the all the nodes for the reduced IIR. The aim of the SAVANT project at the University of Cincinnati is to build an extensible, object-oriented intermediate form (IIR) for the hardware description language VHDL. The VHDL analyzer built into SAVANT is called SCRAM, and the SCRAM parser is constructed as an LL(2) grammar using the Purdue Compiler Construction Tool Set (PCCTS) parser generator. SAVANT is extensible since the Intermediate Form is defined using a standard called Advanced Intermediate Representation with Extensibility (AIRE).

AIRE. The Advanced Intermediate Representation with Extensibility is a set of specifications that support the inter-operability of components of HDL tools and partially compiled HDL models. As a part of the work, it defines an IIR, a memory-based class structure and methods for providing interaction within a single virtual address space. The AIRE has been designed such that it delivers on key user requirements such as availability, capacity, extensibility, performance, portability, reusability, and security. The advantage of using the AIRE specifications is that all the tools that adhere to the standard can be *plugged-in* easily. More precisely, if a tool generates AIRE compatible IIR, it can be used as a front-end for any application that complies with the specification. The object-oriented design of IIR internal intermediate representation,design facilitates reusability and is described in the next section.

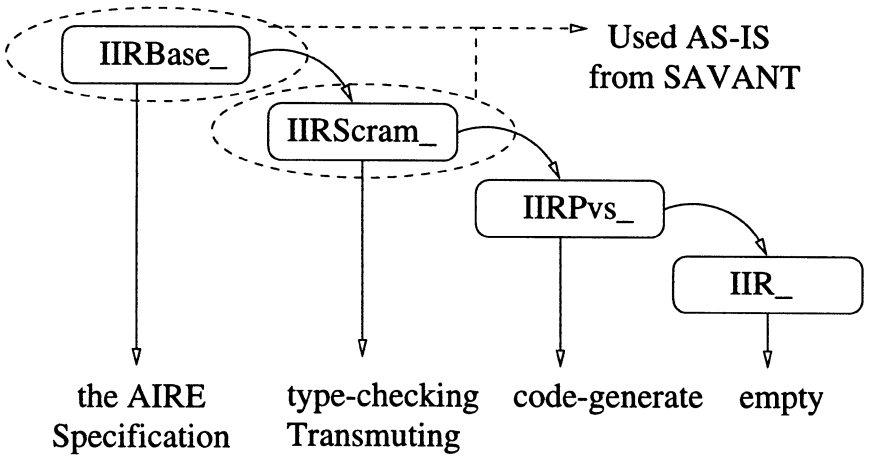


Figure 10.15. The class hierarchy used to generate PVS theories

Design. The AIRE uses a set of objects that represent an Abstract Syntax Tree, and methods associated with those objects that form an Application Programming Interface (API) to store design information. The SAVANT project generates the derivation hierarchy as specified by the AIRE — all the classes are rooted from the definition of a the class `IIR`, and the class `IIR_SequentialStatement` extends the class `IIR_Statement` and so on. The translator uses the class hierarchy shown in Figure 10.15.

Classes with the prefix `IIRBase_` contain public interface as specified in the AIRE specification and private data members that support the implementation of the methods in the public interface. Classes with the prefix `IIRScram_` contain two kinds of methods: methods that can be used by all, and methods that have restricted use. The methods that can be used by all allow the user to generate the VHDL source in a standard form, to publish C++ files that can be simulated after integration with the TyVIS and the WARPED [30] kernels, and to transmute the constructs in the static model to the reduced model. The methods that are defined for restricted use are mainly typechecking routines and symbol-table manipulation functions. Classes with the prefix `IIRPvs_` contain methods that generate the PVS code that corresponds to the object on which they are called. Classes with the prefix `IIR_` contain no data. The only methods defined in them are empty constructors and destructors.

In this project the methods with the prefix `IIRBase_` and `IIRScram_` were used as-is from the public releases of SAVANT. The class hierarchy for the SAVANT project does not include any classes with the prefix `IIRPvs_`. The classes with this prefix were introduced using the notion of extensibility as described in the AIRE. The translator now takes as input, the two VHDL descriptions to be checked for equivalence and the set of signals that are of

interest and generates a PVS theory that contains a THEOREM which when proven asserts that the two VHDL descriptions provided as input are equivalent.

10.4.2 The Translation Process

The translation of VHDL source code into PVS specifications resembles a syntax-directed translation. Figure 10.16 presents the PVS specification generated by checking the equivalence of the specifications shown in Figure 10.6 (the intermediate signal is called d instead of TMP). Furthermore, the figure contains annotations that describe the translation process. The semantic model developed is initially imported (line 3). This is performed to use the results that have already been proven using the embedding and to provide definitions for the types and the constructs to be used in the current definition. Lines 11-14 declare the signals that have been specified in the VHDL source description. Lines 19-25 assert that the signals declared are boolean. Line 33 asserts that the signal watched is c alone — this is obtained from the user via the command line argument of SCRAM. Lines 37-42 contain the translation for a process statement with a sensitivity list. Line 37 asserts that the process statement is not postponed and lines 39-41 contain the specification of the list of ordered statements present in the process. Note that line 40 is obtained by transmuting a process statement with a sensitivity list into a process statement without one, but with a wait statement appended to it, as specified by the reduction algebra. The other lines present in the specification are obtained by translating other processes in a similar fashion.

10.4.3 Applications

Many of the results proven in this effort have been obtained by using the translator. All the results discussed in Section 10.2 were proven using the translator. The VHDL descriptions to be proven as equivalent were provided as input to the translator and the PVS theory containing a THEOREM was generated. The generated theorem was then proven using the axioms and the results that are already a part of the semantic embedding. The optimizations proven in Section 10.3 were used to verify specific instances where the optimization could be applied.

Notes

1. The compete set of axioms and the theorems can be found on the WWW [48]

2. It is equal to the initial value of the signal before the first sequential statement of every process statement.

```
 1: AvarAtmpA_desc_two_a001R5: THEORY     | ---> The theory generated by the translator
 2: BEGIN
 3:   IMPORTING new_vhdl_sem              | ---> Importing the semantic embedding of VHDL
 4:
 5:   pvs_fs : nat = 1                    |
 6:   pvs_ns : nat = 100000               | ---> The definitions of time that may be required
                                          | in the specification
 7:   pvs_s : nat = 100000000000000       |
 8:
 9:   is_1, ws: VAR setof[signal_t]       | ---> Sets of arbitrary signals used to define
                                         | input sets and the user specified signals
10:
11:   a: signal_t                        |
12:   b: signal_t                        | ---> Delcarations in PVS that correspond to the
13:   c: signal_t                        |     signal declarations in the VHDL source
14:   d: signal_t                        |
15:
16:
17:   AvarAtmpA_desc_two_a001R5 : THEOREM | ---> The theorem needed to be proven
18:
19:   bool_prop?(wave(a)) &              |
20:   bool_prop?(wave(b)) &              |
21:   bool_prop?(wave(c)) &              |
22:   bool_prop?(wave(d)) &              | ---> Assert that the signals are boolean
23:   bool_prop?(wave(a)) &              |
24:   bool_prop?(wave(b)) &              |
25:   bool_prop?(wave(c)) &              |
26:
27:      a /= b                         |
28:    & a /= c                         |
29:    & a /= d                         | ---> Assert that each signal is different from the other
30:    & b /= c                         |
31:    & b /= d                         |
32:    & c /= d                         |
33:   & ws = add(c, emptyset)            | ---> Assert that the only signal to watch is c
34:   =>
35:
36:   equiv(
37:    add((# postponement_v := not_postponed,                          | ---> process (a, b)
38:          os_v :=
39:    cons(sa(d, 5 * pvs_fs, after(5 * pvs_fs, and_(wave(a), wave(b)))),  | ---> d <= a and b after 5 fs;
40:    cons(wait(add(a, add(b, emptyset)), true, INFTY),                    | ---> Wait statement introduced by transmuting
41:    null))                                                               | ---> end process
42:     #),
43:    add((# postponement_v := not_postponed,                          | ---> process (d)
44:          os_v :=
45:    cons(sa(c, 0, not_(wave(d))),                                    | ---> c <= not d;
46:    cons(wait(add(d, emptyset), true, INFTY),                         | ---> Wait statement introduced by transmuting
47:    null))                                                           | ---> end process
48:    #),
49:    emptyset)),
50:        postponement_v := not_postponed,                                    | ---> process (a, b)
51:   os_v :=
52:    cons(sa(c, 5 * pvs_fs, after(5 * pvs_fs, nand_(wave(a), wave(b)))),       | ---> c <= a nand b after 5 fs;
53:    cons(wait(add(a, add(b, emptyset)), true, INFTY),       | ---> Wait statement introduced by transmuting
54:       null))                                  | ---> end process
55:       #),
56:      emptyset),
57:      is_1,                                   |
58:      is_1,                                   | ---> The input set of signals to both the descriptions is the same
59:
60:      ws)
61: END AvarAtmpA_desc_two_a001R5
```

Figure 10.16. The annotated PVS specification of the NAND gate

11 CONCLUSIONS

In the previous chapters, a declarative semantics for VHDL was presented. Essentially, a set of time intervals spanning the simulation period was defined and constraints over the evaluation intervals of processes were established against this set. Intermediate sets were constructed defining transaction lists maintained by every process for each signal to which an assignment has been made in that process. However, the definition of the transaction lists was declarative and it was pointed out that this intermediate definition was not necessary but merely simplified the semantics. The embeddeing of the model in PVS was discussed, the validity of the model was proven, the need for a translation unit to convert VHDL source level descriptions to PVS language was argued, and a specific implementation of the translator using SAVANTwas shown. In addition, proofs methodologies were demonstrated using the embedding of the models in PVS. In particular, the validation of *process folding* and *signal collapsing* was presented. The aim of this project is to explore the diverse applications of the semantics and to demonstrate its utility in CAD tool optimization.

11.1 CONTRIBUTIONS OF THIS RESEARCH

CAD tool optimization has not been studied in the context of the application of formal methods to VHDL. In this research, an attempt is made to define a semantics, that is independent of the simulation cycle defined in the VHDL LRM [27].

The reduction algebra outlined in Chapter 5 simplifies the definition of the dynamic semantics and also the construction of VHDL simulators. Consider a parser that transforms VHDL code into its static model representation. The reduction algebra can be applied to this representation and and a core set can be produced. A VHDL simulator can then be built only for this core set. For example, the simulator need not recognize any concurrent statement except for the process statement.

The dynamic model aims at defining a semantics that is independent of the simulation cycle of VHDL hence providing a basis against which different and possibly more efficient simulation techniques could be developed and compared. For example, parallel simulators using distributed synchronization protocols such as time warp [20, 28] can be constructed against this definition. An operational definition of the simulation cycle totally disallows any such investigations. Furthermore, the nature of the semantics supports proofs of equivalences between a VHDL description and a reduced form (by the application of some reduction algebra). Thus optimization techniques such as folding process statements for improving simulation performance can be formally defended [31].

The nature of the semantics supports proofs of equivalences of patterns of VHDL descriptions. A framework has been established to validate optimizations to the simulation process such as *process folding* and *signal collapsing.* These improve the confidence in the safety of implementing such optimizations in the post elaboration stage. In addition, equivalences of specific VHDL descriptions at different levels of abstraction can be proven. Thus, a low level implementation can be verified against a high level behavioral model of the circuit. Some examples that illustrate this have been shown in Chapter 10.

The central design goal of PVS is not merely to prove theorems but also provide useful feedback from failed and partial proofs by serving as a rigorous skeptic. The initial specification [31] did not require that the folded processes not have any resolved signals. During the process of proving process folding without this pre-condition, we were *stuck* at a subgoal that could not be proved. To verify that the pre-conditions stated (Section 10.3.1) were necessary, we did not add the pre-condition that OS_1 (Section 10.3.1) be a *no-wait block.* Again, we were *stuck* at a subgoal that required this condition to be true.

All the pre-conditions that are necessary for the signal collapsing optimization were derived in the course of the proof. The proof starts with signal collapsing with no pre-conditions. We would then get *stuck* at a sub-goal that would require that one of the pre-conditions be true. Using this method, we can obtain the minimum set of pre-conditions necessary for the optimization to be valid. Thus, useful insights are provided by formal specification and verification.

Although the specifications and proofs are important results in themselves, this activity represents an application of formal methods to the optimization of simulation. The chief objective of this approach is to support reasoning

about various alternative methods for optimizing simulation and to obtain the necessary and sufficient pre-conditions for each optimization method.

Last but not the least, an attempt has been made to cover most of the important issues related to the dynamic semantics of VHDL. Important to note are the characterizations of guarded signals (signals of kind *bus* and *register*), shared variables, postponed processes and communication between architectures via net-lists. These issues have usually been ignore in other research attempts.

11.2 FUTURE WORK

Though the semantics presented is complete by itself, procedures and functions still need to be characterized. Essentially, the notion of a state and a memory will need to be incorporated into the model in order to handle scoping of signals , passing of parameters by value and by reference and recursive calls to functions and procedures. Further, for this work it has been assumed that exit statements can appear only at one level (the highest) inside a loop statement. Static rewriting of nested exit statements to equivalent forms with exit statements at only the highest level will be a topic for future investigation. Alternatively, nested exit statements could be incorporated into the semantics directly.

Although well formedness issues for the static semantics have been formulated [53], similar work has not been done for the dynamic model. For example, that a time expression cannot evaluate to a negative value at any stage of the simulation is an issue for well formedness. Future work will address these issues.

The proof methodologies presented in this work demonstrate the utility of the model. However, the techniques used are not yet mature and need to be refined. In particular, the definition of strategies that can automatically prove the PVS descriptions generated for specific instances of an optimization can be examined. Moreover, other applications of the model in proof methodologies can be explored. In particular, the application of the model to speed up the TimeWarp based VHDL simulator can be investigated. The application of the model to behavioral synthesis has been illustrated and can be pursued.

Appendix A

In Section 9.2, an informal proof shows that Process folding preserves the semantics of VHDL description. In this section we present the formal proofs for the same.

Note: In the following proofs, the relation $meets(b, a)$ is represented as $b : a$ for ease of understanding.

Consider two transactions m_1 and n_1 (before folding) such that $m_1 = \langle v_m, a, c \rangle$ and $n_1 = \langle v_n, b, d \rangle$. Let m_2 and n_2 be the corresponding transactions after folding, for m_1 and n_1 respectively. We know that the values of the transactions are the same before and after folding. Hence, we can state that $m_2 = \langle v_m, a', c \rangle$ and $n_2 = \langle v_n, b', d \rangle$. We will show that the transaction n_2 marks the transaction m_2 if and only if the transaction n_1 marks the transaction m_1. We know that the transaction n_1 marks the transaction m_1 if any of the following conditions hold:

1. *during(b, a)*
2. *finishes(b, a)*
3. *overlaps(b, a)*

In the following sections we analyze each of these cases.

A.1 THE RELATION DURING(B,A) HOLDS

We know that n_2 marks m_2 if $during(b', a')$ is true. Hence it suffices to prove that

$$during(b, a) \Leftrightarrow during(b', a').$$

Without loss of generality we can state that it is sufficient to prove the condition,

$$during(b, a) \Rightarrow during(b', a').$$

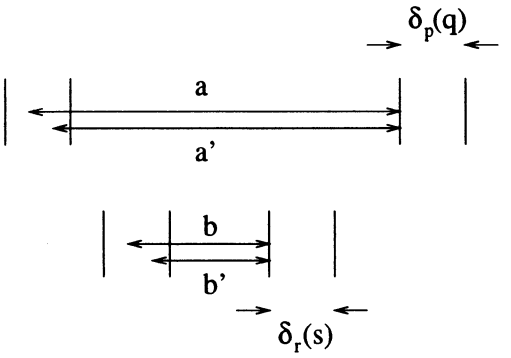


Figure A.1. The case when the relation *during* is true

The relation *during* is expressed in terms of the relations *meets* as follows,

$$\begin{aligned} during(b,a) \quad = \quad & \forall m \cdot m : a \Rightarrow \exists k \cdot m : k \wedge k : b \wedge \\ & \forall n \cdot a : n \Rightarrow \exists l \cdot b : l \wedge l : n. \end{aligned}$$

We can state that $meets(a, \delta_p(q))$ for some p and q and $meets(b, \delta_r(s))$ for some r and s. These form the premises 2 and 3 in the proof shown in Table A.1. We can infer that $meets(a', \delta_p(q))$ and $meets(b', \delta_r(s))$ are true from lemma 9.2.1. These form the premises 4 and 5 in Table A.1. Lemma 9.2.1 states that the ordering of any two statements is preserved during folding. We see, from Figure A.1, that b starts after a. Hence, b' starts after a'. Hence there exists a time interval k such that the following condition holds,

$$\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b'.$$

This is stated as premise 6 in Table A.1. The rest of the proof is self explanatory.

We have automated the proofs using **PVS**(Prototype Verification System). **PVS** is a proof assistant developed at SRI. Given below is the theory input to **PVS**, for modeling this problem. The conjecture DuringCon is the one to be proved for the case when the relation *during(b,a)* holds.

```
finishes_during_overlaps : THEORY
  BEGIN

  TimeInterval : NONEMPTY_TYPE

  meets : [TimeInterval, TimeInterval → bool]

  nuts
```

Table A.1. Gentzen style proof for the case in which *during(b, a)* is true

1	$during(b, a)$	*Assumption*
2	$a : \delta_p(q)$	*Premise*
3	$b : \delta_r(s)$	*Premise*
4	$a' : \delta_p(q)$	*Premise*
5	$b' : \delta_r(s)$	*Premise*
6	$\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b'$	*Premise*
7	$\forall m \cdot (m : a) \Rightarrow \exists k \cdot m : k \wedge k : b \wedge$ $\forall n \cdot (a : n) \Rightarrow \exists l \cdot b : l \wedge l : n$	*Definition of during on 1*
8	$m_1 : a \Rightarrow \exists k \cdot m_1 : k \wedge k : b \wedge$ $a : n_1 \Rightarrow \exists l \cdot b : l \wedge l : n_1$	$\forall$ *-elimination on 7*
9	$a : n_1 \Rightarrow\Rightarrow \exists l \cdot b : l \wedge l : n_1$	$\wedge$ *-elimination on 8*
10	$a' : n_1$	*Assumption*
11	$i : j \wedge i : k \wedge l : k \Rightarrow i : j$	*Axiom*
12	$a' : n_1 \wedge a : \delta_p(q) \wedge a' : \delta_p(q) \Rightarrow a : n_1$	*Substitution*
13	$a' : n_1 \wedge a : \delta_p(q) \wedge a' : \delta_p(q)$	$\wedge$ *-introduction on 10,2,4*
14	$a : n_1$	$\Rightarrow$ *-elimination on 12,13*
15	$\exists l \cdot b : l \wedge l : n_1$	$\Rightarrow$ *-elimination on 9,14*
16	$b : l_1 \wedge l_1 : n_1$	l_1*-arbitrary, Assumption*
17	$b : l_1$	$\wedge$ *-elimination on 16*
18	$b : l_1 \wedge b : \delta_r(s) \wedge b' : \delta_r(s) \Rightarrow b' : l_1$	*Using Axiom*
19	$b : l_1 \wedge b : \delta_r(s) \wedge b' : \delta_r(s)$	$\wedge$ *-introduction on 17,3,5*
20	$b' : l_1$	$\Rightarrow$ *-elimination on 18,19*
21	$l_1 : n_1$	$\wedge$ *-elimination on 16*
22	$b' : n_1 \wedge l_1 : n_1$	$\wedge$ *-introduction on 20,21*
23	$\exists z \cdot b' : z \wedge z : n_1$	$\exists$ *-introduction on 22*
24	$b : l_1 \wedge l_1 : n_1 \Rightarrow \exists z \cdot b' : z \wedge z : n_1$	$\Rightarrow$ *-introduction on 16,23*
25	$\forall l \cdot (b : l \wedge l : n) \Rightarrow \exists z \cdot b' : z \wedge z : n_1$	$\forall$ *-introduction on 24*
26	$\exists z \cdot b' : z \wedge z : n_1$	$\exists$ *-elimination on 15,25*
27	$a' : n_1 \Rightarrow \exists z \cdot b' : z \wedge z : n_1$	$\Rightarrow$ *-introduction on 10,26*
28	$\forall n \cdot a' : n_1 \Rightarrow \exists z \cdot b' : z \wedge z : n_1$	$\forall$ *-introduction on 27*
29	$\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b' \wedge$ $\forall n \cdot a' : n \Rightarrow \exists z \cdot b' : z \wedge z : n_1$	$\wedge$ *-introduction on 6,28*
30	$during(b', a')$	*from the definition of during*
31	$during(b, a) \Rightarrow during(b', a')$	$\Rightarrow$ *-introduction on 1,30*

QED

δ : [nat, nat $\rightarrow$ TimeInterval]

tx, ty : VAR TimeInterval

t_1, t_2, t_3, t_4 : VAR TimeInterval

p, q, r, s : VAR nat

finishes : [TimeInterval, TimeInterval → bool]

during : [TimeInterval, TimeInterval → bool]

overlaps : [TimeInterval, TimeInterval → bool]

a, b : VAR TimeInterval

adash, bdash : VAR TimeInterval

AllensAxiomOne : AXIOM
$\text{meets}(t_1, t_2) \wedge \text{meets}(t_1, t_3) \wedge \text{meets}(t_4, t_2) \Rightarrow \text{meets}(t_4, t_3)$

FinishesDef : AXIOM
finishes(tx, ty) =
((∀ (m : TimeInterval) :
meets(m, ty) ⇒
(∃ (k : TimeInterval) :
meets(m, k) ∧ meets(k, tx)))∧
(∀ (n : TimeInterval) : meets(ty, n) ⇒ meets(tx, n)))

DuringDef : AXIOM
during(tx, ty) =
((∀ (m : TimeInterval) :
meets(m, ty) ⇒
(∃ (k : TimeInterval) :
meets(m, k) ∧ meets(k, tx)))∧
(∀ (n : TimeInterval) :
meets(ty, n) ⇒
(∃ (l : TimeInterval) : meets(tx, l) ∧ meets(l, n))))

OverlapsDef : AXIOM
overlaps(ty, tx) =
((∀ (m : TimeInterval) :
meets(m, ty) ⇒
(∃ (k : TimeInterval) :
meets(m, k) ∧ meets(k, tx)))∧
(∀ (n : TimeInterval) :
meets(tx, n) ⇒
(∃ (k : TimeInterval) :
meets(ty, k) ∧ meets(k, n)))∧
(∀ (O : TimeInterval), (p : TimeInterval) :
meets(O, tx) ∧ meets(ty, p) ⇒
(∃ (k : TimeInterval) :

```
            meets(O, k) ∧ meets(k, p))))

FinishesCon : CONJECTURE
   meets(a, δ_p(q))∧
      meets(adash, δ_p(q))∧
        meets(b, δ_r(s))∧
          meets(bdash, δ_r(s))∧
            (∀ (m : TimeInterval) :
               meets(m, adash) ⇒
                 (∃ (k : TimeInterval) :
                   meets(m, k)∧
                     meets(k,
                       bdash)))∧
              finishes(b, a) ⇒
     finishes(bdash, adash)

DuringCon : CONJECTURE
   meets(a, δ_p(q))∧
      meets(adash, δ_p(q))∧
        meets(b, δ_r(s))∧
          meets(bdash, δ_r(s))∧
            (∀ (m : TimeInterval) :
               meets(m, adash) ⇒
                 (∃ (k : TimeInterval) :
                   meets(m, k)∧
                     meets(k,
                       bdash)))∧
              during(b, a) ⇒
     during(bdash, adash)

OverlapsCon : CONJECTURE
   meets(a, δ_p(q))∧
      meets(adash, δ_p(q))∧
        meets(b, δ_r(s))∧
          meets(bdash, δ_r(s))∧
            (∀ (m : TimeInterval) :
               meets(m, adash) ⇒
                 (∃ (k : TimeInterval) :
                   meets(m, k)∧
                     meets(k,
                       bdash)))∧
              (∀ (O : TimeInterval),
                (p : TimeInterval) :
                  meets(O, bdash)∧
                    meets(adash, p) ⇒
                   (∃ (k :
                   TimeInterval) :
                     meets(O, k)∧
                       meets(k, p)))∧
```

overlaps(a, b) $\Rightarrow$
overlaps(adash, bdash)

END finishes_during_overlaps

Verbose proof for `DuringCon`.

`DuringCon`:

$$
\begin{array}{|l}
\hline
\{1\}\quad (\forall\ (a:\ \text{TimeInterval}, \text{adash}:\ \text{TimeInterval}, b:\ \text{TimeInterval},\\
\quad \text{bdash}:\ \text{TimeInterval}, p:\ \{i:\ \text{integer} \mid\ i \geq 0\},\\
\quad q:\ \{i:\ \text{integer} \mid\ i \geq 0\}, r:\ \{i:\ \text{integer} \mid\ i \geq 0\},\\
\quad s:\ \{i:\ \text{integer} \mid\ i \geq 0\}):\\
\quad \text{meets}(a, \delta_p(q)) \wedge\\
\quad\ \text{meets}(\text{adash}, \delta_p(q)) \wedge\\
\quad\ \text{meets}(b, \delta_r(s)) \wedge\\
\quad\ \text{meets}(\text{bdash}, \delta_r(s)) \wedge\\
\quad\ (\forall\ (m:\ \text{TimeInterval}):\\
\quad\ \text{meets}(m,\\
\quad\ \text{adash}) \Rightarrow\\
\quad\ (\exists\ (k:\\
\quad\ \text{TimeInterval}):\\
\quad\ \text{meets}(m, k) \wedge\\
\quad\ \text{meets}(k,\\
\quad\ \text{bdash}))) \wedge\\
\quad\ \text{during}(b, a) \Rightarrow\\
\quad \text{during}(\text{bdash}, \text{adash}))\\
\hline
\end{array}
$$

`DuringCon`:

$$
\begin{array}{|l}
\hline
[1]\quad (\forall\ (a:\ \text{TimeInterval}, \text{adash}:\ \text{TimeInterval}, b:\ \text{TimeInterval},\\
\quad \text{bdash}:\ \text{TimeInterval}, p:\ \{i:\ \text{integer} \mid\ i \geq 0\},\\
\quad q:\ \{i:\ \text{integer} \mid\ i \geq 0\}, r:\ \{i:\ \text{integer} \mid\ i \geq 0\},\\
\quad s:\ \{i:\ \text{integer} \mid\ i \geq 0\}):\\
\quad \text{meets}(a, \delta_p(q)) \wedge\\
\quad\ \text{meets}(\text{adash}, \delta_p(q)) \wedge\\
\quad\ \text{meets}(b, \delta_r(s)) \wedge\\
\quad\ \text{meets}(\text{bdash}, \delta_r(s)) \wedge\\
\quad\ (\forall\ (m:\ \text{TimeInterval}):\\
\quad\ \text{meets}(m,\\
\quad\ \text{adash}) \Rightarrow\\
\quad\ (\exists\ (k:\\
\quad\ \text{TimeInterval}):\\
\quad\ \text{meets}(m, k) \wedge\\
\quad\ \text{meets}(k,\\
\quad\ \text{bdash}))) \wedge\\
\quad\ \text{during}(b, a) \Rightarrow\\
\quad \text{during}(\text{bdash}, \text{adash}))\\
\hline
\end{array}
$$

Repeatedly Skolemizing and flattening,

`DuringCon:`

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, \text{adash}') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}')))$
{-6}	$\text{during}(b', a')$
{1}	$\text{during}(\text{bdash}', \text{adash}')$

Rewriting using DuringDef,

`DuringCon:`

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, \text{adash}') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}')))$
{-6}	$((\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, a') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):$ $\text{meets}(m,k) \wedge \text{meets}(k, b')))\wedge$ $(\forall\ (n:\ \text{TimeInterval}):$ $\text{meets}(a', n) \Rightarrow$ $(\exists\ (l:\ \text{TimeInterval}):$ $\text{meets}(b', l) \wedge \text{meets}(l, n))))$
{1}	$\text{during}(\text{bdash}', \text{adash}')$

Rewriting using DuringDef,

DuringCon:

$$
\begin{array}{|ll}
\{\text{-}1\} & \text{meets}(a', \delta_{p'}(q')) \\
\{\text{-}2\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{\text{-}3\} & \text{meets}(b', \delta_{r'}(s')) \\
\{\text{-}4\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{\text{-}5\} & (\forall\ (m:\ \text{TimeInterval}): \\
 & \quad \text{meets}(m, \text{adash}') \Rightarrow \\
 & \quad\quad (\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}'))) \\
\{\text{-}6\} & ((\forall\ (m:\ \text{TimeInterval}): \\
 & \quad \text{meets}(m, a') \Rightarrow \\
 & \quad\quad (\exists\ (k:\ \text{TimeInterval}): \\
 & \quad\quad\quad \text{meets}(m,k) \wedge \text{meets}(k, b'))) \wedge \\
 & \quad (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad\quad \text{meets}(a', n) \Rightarrow \\
 & \quad\quad\quad (\exists\ (l:\ \text{TimeInterval}): \\
 & \quad\quad\quad\quad \text{meets}(b', l) \wedge \text{meets}(l, n)))) \\
\hline
\{1\} & ((\forall\ (m:\ \text{TimeInterval}): \\
 & \quad \text{meets}(m, \text{adash}') \Rightarrow \\
 & \quad\quad (\exists\ (k:\ \text{TimeInterval}): \\
 & \quad\quad\quad \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}'))) \wedge \\
 & \quad (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad\quad \text{meets}(\text{adash}', n) \Rightarrow \\
 & \quad\quad\quad (\exists\ (l:\ \text{TimeInterval}): \\
 & \quad\quad\quad\quad \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n))))
\end{array}
$$

Applying propositional simplification and decision procedures,

DuringCon:

$$
\begin{array}{|ll}
\{\text{-}1\} & \text{meets}(a', \delta_{p'}(q')) \\
\{\text{-}2\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{\text{-}3\} & \text{meets}(b', \delta_{r'}(s')) \\
\{\text{-}4\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{\text{-}5\} & (\forall\ (m:\ \text{TimeInterval}): \\
 & \quad \text{meets}(m, \text{adash}') \Rightarrow \\
 & \quad\quad (\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}'))) \\
\{\text{-}6\} & (\forall\ (m:\ \text{TimeInterval}): \\
 & \quad \text{meets}(m, a') \Rightarrow \\
 & \quad\quad (\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, b'))) \\
\{\text{-}7\} & (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad \text{meets}(a', n) \Rightarrow \\
 & \quad\quad (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(b', l) \wedge \text{meets}(l, n))) \\
\hline
\{1\} & (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad \text{meets}(\text{adash}', n) \Rightarrow \\
 & \quad\quad (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n)))
\end{array}
$$

Deleting some formulas,

DuringCon:

$$
\begin{array}{|ll}
\{-1\} & \text{meets}(a', \delta_{p'}(q')) \\
\{-2\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{-3\} & \text{meets}(b', \delta_{r'}(s')) \\
\{-4\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{-5\} & (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad \text{meets}(a', n) \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(b', l) \wedge \text{meets}(l, n))) \\
\hline
\{1\} & (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad \text{meets}(\text{adash}', n) \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n)))
\end{array}
$$

Skolemizing,

DuringCon:

$$
\begin{array}{|ll}
\{-1\} & \text{meets}(a', \delta_{p'}(q')) \\
\{-2\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{-3\} & \text{meets}(b', \delta_{r'}(s')) \\
\{-4\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{-5\} & (\forall\ (n:\ \text{TimeInterval}): \\
 & \quad \text{meets}(a', n) \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(b', l) \wedge \text{meets}(l, n))) \\
\hline
\{1\} & \text{meets}(\text{adash}', n') \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n'))
\end{array}
$$

Instantiating the top quantifier in -5 with the terms: n!1,

DuringCon:

$$
\begin{array}{|ll}
\{-1\} & \text{meets}(a', \delta_{p'}(q')) \\
\{-2\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{-3\} & \text{meets}(b', \delta_{r'}(s')) \\
\{-4\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{-5\} & \text{meets}(a', n') \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(b', l) \wedge \text{meets}(l, n')) \\
\hline
\{1\} & \text{meets}(\text{adash}', n') \Rightarrow \\
 & \quad\ (\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n'))
\end{array}
$$

Applying propositional simplification and decision procedures,

we get 2 subgoals:

DuringCon.1:

{-1}	$(\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(b', l) \wedge \text{meets}(l, n'))$
{-2}	$\text{meets}(\text{adash}', n')$
{-3}	$\text{meets}(a', \delta_{p'}(q'))$
{-4}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-5}	$\text{meets}(b', \delta_{r'}(s'))$
{-6}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{1}	$(\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n'))$

Skolemizing,

DuringCon.1:

{-1}	$\text{meets}(b', l') \wedge \text{meets}(l', n')$
[-2]	$\text{meets}(\text{adash}', n')$
[-3]	$\text{meets}(a', \delta_{p'}(q'))$
[-4]	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
[-5]	$\text{meets}(b', \delta_{r'}(s'))$
[-6]	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
[1]	$(\exists\ (l:\ \text{TimeInterval}):\ \text{meets}(\text{bdash}', l) \wedge \text{meets}(l, n'))$

Instantiating the top quantifier in 1 with the terms: l!1,

DuringCon.1:

{-1}	$\text{meets}(b', l') \wedge \text{meets}(l', n')$
[-2]	$\text{meets}(\text{adash}', n')$
[-3]	$\text{meets}(a', \delta_{p'}(q'))$
[-4]	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
[-5]	$\text{meets}(b', \delta_{r'}(s'))$
[-6]	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{1}	$\text{meets}(\text{bdash}', l') \wedge \text{meets}(l', n')$

Applying propositional simplification and decision procedures,

DuringCon.1:

{-1}	$\text{meets}(b', l')$
{-2}	$\text{meets}(l', n')$
[-3]	$\text{meets}(\text{adash}', n')$
[-4]	$\text{meets}(a', \delta_{p'}(q'))$
[-5]	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
[-6]	$\text{meets}(b', \delta_{r'}(s'))$
[-7]	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{1}	$\text{meets}(\text{bdash}', l')$

Deleting some formulas,

`DuringCon.1:`

{-1}	meets(b', l')
[-2]	meets(b', $\delta_{r'}(s')$)
[-3]	meets(bdash$'$, $\delta_{r'}(s')$)
{1}	meets(bdash$'$, l')

Applying AllensAxiomOne where t1 gets b!1, t2 gets $\delta_{r'}(s')$, t3 gets l!1, t4 gets bdash!1,

`DuringCon.1:`

{-1}	meets(b', $\delta_{r'}(s')$)$\wedge$ meets(b', l') $\wedge$ meets(bdash$'$, $\delta_{r'}(s')$) $\Rightarrow$ meets(bdash$'$, l')
{-2}	meets(b', l')
[-3]	meets(b', $\delta_{r'}(s')$)
[-4]	meets(bdash$'$, $\delta_{r'}(s')$)
{1}	meets(bdash$'$, l')

Applying propositional simplification and decision procedures,

This completes the proof of `DuringCon.1`.

`DuringCon.2:`

{-1}	meets(adash$'$, n')
{-2}	meets(a', $\delta_{p'}(q')$)
{-3}	meets(adash$'$, $\delta_{p'}(q')$)
{-4}	meets(b', $\delta_{r'}(s')$)
{-5}	meets(bdash$'$, $\delta_{r'}(s')$)
{1}	meets(a', n')
{2}	($\exists$ (l : TimeInterval) : meets(bdash$'$, l) $\wedge$ meets(l, n'))

Deleting some formulas,

`DuringCon.2:`

[-1]	meets(adash$'$, n')
[-2]	meets(a', $\delta_{p'}(q')$)
[-3]	meets(adash$'$, $\delta_{p'}(q')$)
[1]	meets(a', n')

Applying AllensAxiomOne where t1 gets adash!1, t2 gets $\delta_{p'}(q')$, t3 gets n!1, t4 gets a!1,

`DuringCon.2:`

	{-1}	meets(adash$'$, $\delta_{p'}(q')$)$\wedge$ meets(adash$'$, n') $\wedge$ meets(a', $\delta_{p'}(q')$) $\Rightarrow$ meets(a', n')
	[-2]	meets(adash$'$, n')
	[-3]	meets(a', $\delta_{p'}(q')$)
	[-4]	meets(adash$'$, $\delta_{p'}(q')$)
	[1]	meets(a', n')

Applying propositional simplification and decision procedures,

This completes the proof of `DuringCon.2`.

Q.E.D.

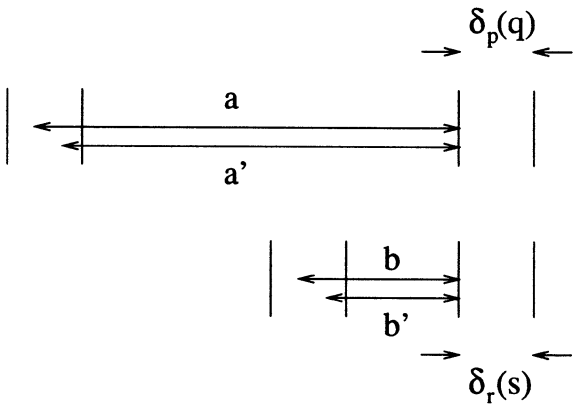


Figure A.2. The case when the relation *finishes* is true

A.2 THE RELATION FINISHES(B,A) HOLDS

We know that the transaction n_1 marks the transaction m_1 if the relation $finishes(b, a)$ is true. The relation *finishes* is shown in Figure A.2. If teh relation $finishes(b, a)$ is true, we need to prove that the relation $finishes(b', a')$ is also true. This is shown in Table A.2. Stated formally, we need to prove that

$$finishes(b, a) \Rightarrow finishes(b', a').$$

We know that

$$finishes(b, a) = (\forall m \cdot m : a \Rightarrow \exists k \cdot m : k \wedge k : b) \wedge (\forall n \cdot a : n \Rightarrow b : n).$$

The premises 2, 3, 4, 5 and 6 are the same as the premises using in the previous proof and the rest of the proof is self explanatory.

Given below is the automated proof obtained using **PVS** for the conjecture FinishesCon. The conjecture FinishesCon, essentially states that given the premises and the axioms,

$$finishes(b, a) \Rightarrow finishes(b', a')$$

Verbose proof for `FinishesCon`.

Table A.2. Gentzen style proof for the case in which *finishes(b, a)* is true

1	$finishes(b, a)$	*Assumption*
2	$a : \delta_p(q)$	*Premise*
3	$b : \delta_r(s)$	*Premise*
4	$a' : \delta_p(q)$	*Premise*
5	$b' : \delta_r(s)$	*Premise*
6	$\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b'$	*Premise*
7	$i : j \wedge i : l \wedge k : j \Rightarrow k : l$	*Axiom*
8	$(\forall m \cdot m : a \Rightarrow \exists k \cdot m : k \wedge k : b) \wedge (\forall n \cdot a : n \Rightarrow b : n)$	*Definition of finishes on 1*
9	$\forall n \cdot a : n \Rightarrow b : n$	*$\wedge$-elimination on 8*
10	$a : n_1 \Rightarrow b : n_1$	*$\forall$-elimination on 9*
11	$a' : n_1$	*Assumption*
12	$a' : \delta_p(q) \wedge a' : n_1 \wedge a : \delta_p(q)$	*$\wedge$-introduction on 2,4,11*
13	$a' : \delta_p(q) \wedge a' : n_1 \wedge a : \delta_p(q) \Rightarrow a : n_1$	*Using Axiom*
14	$a : n_1$	*$\Rightarrow$-elimination on 12,13*
15	$b : n_1$	*$\Rightarrow$-elimination on 10,14*
16	$b : n_1 \wedge b : \delta_r(s) \wedge b' : \delta_r(s)$	*$\wedge$-introduction on 3,5,15*
17	$b : n_1 \wedge b : \delta_r(s) \wedge b' : \delta_r(s) \Rightarrow b' : n_1$	*Using Axiom*
18	$b' : n_1$	*$\Rightarrow$-elimination on 16,17*
19	$a' : n_1 \Rightarrow b' : n_1$	*$\Rightarrow$-introduction on 11,18*
20	$\forall n \cdot a' : n \Rightarrow b' : n$	*$\forall$-introduction on 19*
21	$(\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b') \wedge (\forall n \cdot a' : n \Rightarrow b' : n)$	*$\wedge$-introduction on 6,20*
22	$finishes(b', a')$	*Definition of finishes*
23	$finishes(b, a) \Rightarrow finishes(b', a')$	*$\Rightarrow$-introduction on 1,22*

QED

```
FinishesCon:
  |
  |-------------------------------------------------------------
  | {1}  (∀ (a : TimeInterval, adash : TimeInterval, b : TimeInterval,
  |         bdash : TimeInterval, p : {i : integer | i ≥ 0},
  |         q : {i : integer | i ≥ 0}, r : {i : integer | i ≥ 0},
  |         s : {i : integer | i ≥ 0}) :
  |       meets(a, δp(q))∧
  |         meets(adash, δp(q))∧
  |           meets(b, δr(s))∧
  |             meets(bdash, δr(s))∧
  |               (∀ (m : TimeInterval) :
  |                  meets(m,
  |                    adash) ⇒
  |                  (∃ (k :
  |                  TimeInterval) :
  |                    meets(m, k)∧
  |                      meets(k,
  |                        bdash)))∧
  |                 finishes(b, a) ⇒
  |           finishes(bdash, adash))
```

`FinishesCon:`

$$
\begin{array}{|l}
\hline
\text{[1]} \quad (\forall\ (a:\ \text{TimeInterval}, \text{adash}:\ \text{TimeInterval}, b:\ \text{TimeInterval},\\
\qquad \text{bdash}:\ \text{TimeInterval}, p:\ \{i:\ \text{integer} \mid\ i \geq 0\}, \\
\qquad q:\ \{i:\ \text{integer} \mid\ i \geq 0\}, r:\ \{i:\ \text{integer} \mid\ i \geq 0\}, \\
\qquad s:\ \{i:\ \text{integer} \mid\ i \geq 0\}):\\
\qquad \text{meets}(a, \delta_p(q)) \wedge \\
\qquad\quad \text{meets}(\text{adash}, \delta_p(q)) \wedge \\
\qquad\qquad \text{meets}(b, \delta_r(s)) \wedge \\
\qquad\qquad\quad \text{meets}(\text{bdash}, \delta_r(s)) \wedge \\
\qquad\qquad\qquad (\forall\ (m:\ \text{TimeInterval}):\\
\qquad\qquad\qquad\quad \text{meets}(m,\\
\qquad\qquad\qquad\qquad \text{adash}) \Rightarrow \\
\qquad\qquad\qquad\qquad (\exists\ (k:\\
\qquad\qquad\qquad\qquad \text{TimeInterval}):\\
\qquad\qquad\qquad\qquad\quad \text{meets}(m, k) \wedge \\
\qquad\qquad\qquad\qquad\qquad \text{meets}(k,\\
\qquad\qquad\qquad\qquad\qquad\quad \text{bdash}))) \wedge \\
\qquad\qquad\qquad\quad \text{finishes}(b, a) \Rightarrow \\
\qquad\qquad\quad \text{finishes}(\text{bdash}, \text{adash}))\\
\hline
\end{array}
$$

Repeatedly Skolemizing and flattening,

`FinishesCon:`

$$
\begin{array}{|ll}
\{\text{-1}\} & \text{meets}(a', \delta_{p'}(q')) \\
\{\text{-2}\} & \text{meets}(\text{adash}', \delta_{p'}(q')) \\
\{\text{-3}\} & \text{meets}(b', \delta_{r'}(s')) \\
\{\text{-4}\} & \text{meets}(\text{bdash}', \delta_{r'}(s')) \\
\{\text{-5}\} & (\forall\ (m:\ \text{TimeInterval}):\\
 & \qquad \text{meets}(m, \text{adash}') \Rightarrow \\
 & \qquad\quad (\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m, k) \wedge \text{meets}(k, \text{bdash}'))) \\
\{\text{-6}\} & \text{finishes}(b', a') \\
\hline
\{1\} & \text{finishes}(\text{bdash}', \text{adash}') \\
\hline
\end{array}
$$

Rewriting using FinishesDef,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, \text{adash}') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}')))$
{-6}	$((\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, a') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):$ $\text{meets}(m,k) \wedge \text{meets}(k, b')))\wedge$ $(\forall\ (n:\ \text{TimeInterval}):\ \text{meets}(a', n) \Rightarrow \text{meets}(b', n)))$
{1}	$\text{finishes}(\text{bdash}', \text{adash}')$

Rewriting using FinishesDef,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, \text{adash}') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):\ \text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}')))$
{-6}	$((\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, a') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):$ $\text{meets}(m,k) \wedge \text{meets}(k, b')))\wedge$ $(\forall\ (n:\ \text{TimeInterval}):\ \text{meets}(a', n) \Rightarrow \text{meets}(b', n)))$
{1}	$((\forall\ (m:\ \text{TimeInterval}):$ $\text{meets}(m, \text{adash}') \Rightarrow$ $(\exists\ (k:\ \text{TimeInterval}):$ $\text{meets}(m,k) \wedge \text{meets}(k, \text{bdash}')))\wedge$ $(\forall\ (n:\ \text{TimeInterval}):$ $\text{meets}(\text{adash}', n) \Rightarrow \text{meets}(\text{bdash}', n)))$

Applying propositional simplification and decision procedures,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (m :\ \text{TimeInterval}) :\ \text{meets}(m, \text{adash}') \Rightarrow (\exists\ (k :\ \text{TimeInterval}) :\ \text{meets}(m, k) \wedge \text{meets}(k, \text{bdash}')))$
{-6}	$(\forall\ (m :\ \text{TimeInterval}) :\ \text{meets}(m, a') \Rightarrow (\exists\ (k :\ \text{TimeInterval}) :\ \text{meets}(m, k) \wedge \text{meets}(k, b')))$
{-7}	$(\forall\ (n :\ \text{TimeInterval}) :\ \text{meets}(a', n) \Rightarrow \text{meets}(b', n))$
{1}	$(\forall\ (n :\ \text{TimeInterval}) :\ \text{meets}(\text{adash}', n) \Rightarrow \text{meets}(\text{bdash}', n))$

Deleting some formulas,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (n :\ \text{TimeInterval}) :\ \text{meets}(a', n) \Rightarrow \text{meets}(b', n))$
{1}	$(\forall\ (n :\ \text{TimeInterval}) :\ \text{meets}(\text{adash}', n) \Rightarrow \text{meets}(\text{bdash}', n))$

Skolemizing,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$(\forall\ (n :\ \text{TimeInterval}) :\ \text{meets}(a', n) \Rightarrow \text{meets}(b', n))$
{1}	$\text{meets}(\text{adash}', n') \Rightarrow \text{meets}(\text{bdash}', n')$

Instantiating the top quantifier in -5 with the terms: n!1,

FinishesCon:

{-1}	$\text{meets}(a', \delta_{p'}(q'))$
{-2}	$\text{meets}(\text{adash}', \delta_{p'}(q'))$
{-3}	$\text{meets}(b', \delta_{r'}(s'))$
{-4}	$\text{meets}(\text{bdash}', \delta_{r'}(s'))$
{-5}	$\text{meets}(a', n') \Rightarrow \text{meets}(b', n')$
{1}	$\text{meets}(\text{adash}', n') \Rightarrow \text{meets}(\text{bdash}', n')$

Applying AllensAxiomOne where t1 gets b!1, t2 gets $\delta_{r'}(s')$, t3 gets n!1, t4 gets bdash!1,

FinishesCon:

{-1}	meets$(b', \delta_{r'}(s'))\wedge$ meets$(b', n') \wedge$ meets$(\text{bdash}', \delta_{r'}(s')) \Rightarrow$ meets(bdash', n')
{-2}	meets$(a', \delta_{p'}(q'))$
{-3}	meets$(\text{adash}', \delta_{p'}(q'))$
{-4}	meets$(b', \delta_{r'}(s'))$
{-5}	meets$(\text{bdash}', \delta_{r'}(s'))$
{-6}	meets$(a', n') \Rightarrow$ meets(b', n')
{1}	meets$(\text{adash}', n') \Rightarrow$ meets(bdash', n')

Applying AllensAxiomOne where t1 gets adash!1, t2 gets $\delta_{p'}(q')$, t3 gets n!1, t4 gets a!1,

FinishesCon:

{-1}	meets$(\text{adash}', \delta_{p'}(q'))\wedge$ meets$(\text{adash}', n') \wedge$ meets$(a', \delta_{p'}(q')) \Rightarrow$ meets(a', n')
{-2}	meets$(b', \delta_{r'}(s'))\wedge$ meets$(b', n') \wedge$ meets$(\text{bdash}', \delta_{r'}(s')) \Rightarrow$ meets(bdash', n')
{-3}	meets$(a', \delta_{p'}(q'))$
{-4}	meets$(\text{adash}', \delta_{p'}(q'))$
{-5}	meets$(b', \delta_{r'}(s'))$
{-6}	meets$(\text{bdash}', \delta_{r'}(s'))$
{-7}	meets$(a', n') \Rightarrow$ meets(b', n')
{1}	meets$(\text{adash}', n') \Rightarrow$ meets(bdash', n')

Applying propositional simplification and decision procedures,

This completes the proof of FinishesCon.

Q.E.D.

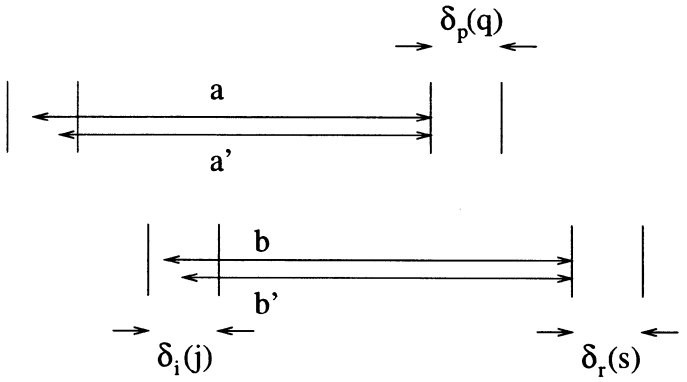


Figure A.3. The relation *overlaps* between transactions

A.3 THE RELATION OVERLAPS(A,B) HOLDS

In this case, the transaction n_1 marks the transaction m_1 since the relation $overlaps(a, b)$ is true (refer to Figure A.3). The proof goal is to show that this relation is preserved for the corresponding transactions after folding. More precisely we need to show that

$$overlaps(a, b) \Rightarrow overlaps(a', b')$$

holds. The proof proceeds by contradiction. We start by assuming that $overlaps(a', b')$ is false. Since the ordering of statements is preserved, it is clear that b' starts after a' $(\exists k \cdot a' : k \wedge k : b')$. Therefore, if $overlaps(a', b')$ is false (our assumption), then $before(a', b')$ must be true (modified assumption). The proof steps will show that $before(a, b)$ is true if $before(a', b')$ is true. Since we already know that $overlaps(a, b)$ is true, we also know that $before(a, b)$ cannot be true. This contradicts the assumption that $overlaps(a', b')$ is false. The important steps in the proof are discussed below. The premises for the proof are the same as the premises stated in the earlier proofs. These premises are stated below.

Table A.4 contains the set of axioms and definitions that will be used throughout the proof. The first axiom (Allen's axiom) states that if there are two intervals i and k which meet the same interval j and if one the intervals i meets another interval l, then the interval k meets the interval l. Axiom 2 and Axiom 3 provide definitions for the relations *overlaps* and *before* respectively. Two new axioms on time intervals are added which make the proof easier to read. Axiom 5 states that if an interval i meets an interval j and if the interval j meets another interval k, then the interval i meets the interval $j + k$. Axiom 6 states that two time intervals i and j satisfy the relation *before* if and only

Table A.3. Premises for the *overlaps* case

1	$a : \delta_p(q)$	*Premise*
2	$b : \delta_r(s)$	*Premise*
3	$a' : \delta_p(q)$	*Premise*
4	$b' : \delta_r(s)$	*Premise*
5	$overlaps(\delta_i(j), b)$	*Premise*
6	$overlaps(\delta_i(j), b')$	*Premise*
7	$\forall m \cdot m : a' \Rightarrow \exists k \cdot m : k \wedge k : b'$	*Premise*

Table A.4. Axioms and definitions for the *overlaps* case

1	$\forall i, j, k, l \cdot i : j \wedge i : l \wedge k : j \Rightarrow k : l$	*Allen's axiom*
2	$\forall a, b \cdot overlaps(a, b) =$	*Definition of overlaps*
3	$\forall a, b \cdot before(a, b) = \exists k \cdot a : k \wedge k : b$	*Definition of before*
4	$\forall i, j, k \cdot i : j \wedge j : k \Rightarrow i : (j + k)$	
5	$\forall i, j, k \cdot i : j \wedge j : k \Rightarrow (i + j) : k$	
6	$\forall i, j \cdot before(i, j) \Leftrightarrow \neg overlaps(i, j)$	*Mutually exclusive*

if they do not satisfy the relation *overlaps*. In Table A.5 we state our (modified) assumption that $before(a', b')$ is true and get an equivalent form using the definition for *before* stated in Table A.4.

Table A.5. Assumption and its equivalent form

1	$before(a', b')$	*Assumption*
2	$\exists k \cdot a' : k \wedge k : b'$	*Using definition 3 on 1*

The proof now proceeds by deriving the desired contradiction for each possible relation between the delta intervals $\delta_i(j)$ and $\delta_p(q)$. All the possible relations between any two delta intervals are listed as the clauses of the tautology

$$\delta_i(j) : \delta_p(q) \vee \delta_p(q) : \delta_i(j) \vee \delta_p(q) \quad = \quad \delta_i(j) \vee before(\delta_p(q), \delta_i(j)) \vee before(\delta_i(j), \delta_p(q)).$$

The tables A.6 through A.10 present the derivation of the contradiction in each of the 5 cases (corresponding to the 5 clauses of the tautology.)

Since a contradiction is achieved in each of the 5 cases, our assumption that $overlaps(a', b')$ is false in incorrect. Hence $overlaps(a', b')$ is true. The initial

Table A.6. The case when $\delta_i(j) : \delta_p(q)$ holds

1	$\delta_i(j) : \delta_p(q)$	*Assumption*
2	$\neg before(a, b)$	*Assumption*
3	$a' : k_1 \wedge k_1 : b'$	$\exists$ *-elimination on 2*
4	$a' : k_1$	$\wedge$ *-elimination on 3*
5	$\delta_i(j) : \delta_p(q) \wedge a' : k_1 \wedge a' : \delta_p(q)$	$\wedge$ *-introduction on 1, 4 and axiom 3*
6	$\delta_i(j) : k_1$	$\Rightarrow$ *-elimination using 5 on axiom 1*
7	$k_1 : b'$	$\wedge$ *-elimination on 3*
8	$\delta_i(j) : k_1 \wedge k_1 : b'$	$\wedge$ *-introduction on 6 and 7*
9	$\exists k \cdot \delta_i(j) : k \wedge k : b'$	$\exists$ *-introduction on 8*
10	$before(\delta_i(j), b')$	*Using definition 3 on 9*
11	$\neg overlaps(\delta_i(j), b')$	$\Rightarrow$ *-elimination on axiom 6 using 10*
12	$overlaps(\delta_i(j), b')$	*Using axiom 5*
13	$before(a, b)$	$\neg$*-introduction on 2 using 11 and 12*

assumption was that $overlaps(a, b)$ holds. We discharge the assumption by stating that

$$overlaps(a, b) \Rightarrow overlaps(a', b').$$

This completes the proof for the goal that the transaction n_1 marks the transaction m_1 when the relation $overlaps(a, b)$ holds. In each of the cases (defined by the relations *finishes*, *during*, and *overlaps*) we find that the transaction n_2 marks the transaction m_2 if the transaction n_1 marks the transaction m_1. Therefore it can be inferred that if a transaction is a `false` transaction before folding, its corresponding transaction after folding will also be a `false` transaction. This establishes the validity of process folding. This proof has also been automated using the PVS theorem prover; details are available in [53].

Table A.7. The case when $\delta_p(q) : \delta_i(j)$ holds

1	$\delta_p(q) : \delta_i(j)$	*Assumption*
2	$overlaps(\delta_i(j), b)$	*Using premise 5*
3	$\forall m \cdot m : \delta_i(j) \Rightarrow \exists k \cdot m : k \wedge k : b$	$\wedge$ *-elimination after using axiom 2 on 2*
4	$\forall \delta_p(q) : \delta_i(j) \Rightarrow \exists k \cdot \delta_p(q) : k \wedge k : b$	*Instantiating m with* $\delta_p(q)$
5	$\exists k \cdot \delta_p(q) : k \wedge k : b$	$\Rightarrow$ *-elimination on 4 using 1*
6	$\delta_p(q) : k_2 \wedge k_2 : b$	*Introducing a specific constant*
7	$a : \delta_p(q)$	*Premise 1*
8	$\delta_p(q) : k_2$	$\wedge$ *-elimination on 6*
9	$k_2 : b$	$\wedge$ *-elimination on 6*
10	$a : (\delta_p(q) + k_2)$	*Using axiom 4 on 7 and 8*
11	$(\delta_p(q) + k_2) : b$	*Using axiom 5 on 8 and 9*
12	$a : (\delta_p(q) + k_2) \wedge (\delta_p(q) + k_2) : b$	$\wedge$ *-introduction on 10 and 11*
13	$\exists k \cdot a : k \wedge k : b$	$\exists$ *-introduction on 12*
14	$before(a, b)$	*Using definition 3 on 13*

Table A.8. The case when $\delta_i(j) = \delta_p(q)$ holds

1	$\delta_p(q) = \delta_i(j)$	*Assumption*
2	$overlaps(\delta_i(j), b)$	*Using premise 5*
3	$\forall m \cdot m : \delta_i(j) \Rightarrow \exists k \cdot m : k \wedge k : b$	$\wedge$ *-elimination after using axiom 2 on 2*
4	$a : \delta_i(j) \Rightarrow \exists k \cdot a : k \wedge k : b$	*Instantiating m with a*
5	$a : \delta_p(q) \Rightarrow \exists k \cdot a : k \wedge k : b$	*Substituting using 1 on 4*
6	$\exists k \cdot a : k \wedge k : b$	$\Rightarrow$ *-elimination on 5 using premise 1*
7	$before(a, b)$	*Using definition 3 on 6*

Table A.9. The case when $before(\delta_p(q), \delta_i(j))$ holds

1	$before(\delta_p(q), \delta_i(j))$	*Assumption*
2	$\exists m \cdot m : \delta_i(j) \Rightarrow \exists k \cdot m : k \wedge k : b$	*Using definition 2 on premise 5*
3	$\exists k \cdot \delta_p(q) : k \wedge k : \delta_i(j)$	*Using definition 3 on 1*
4	$\delta_p(q) : k_3 \wedge k_3 : \delta_i(j)$	*Introducing a specific constant*
5	$k_3 : \delta_i(j)$	$\wedge$ *-elimination on 4*
6	$k_3 : \delta_i(j) \Rightarrow \exists k \cdot k_3 : k \wedge k : b$	*Instantiating m in 2 by* k_3
7	$\exists k \cdot k_3 : k \wedge k : b$	$\Rightarrow$ *-elimination using 5 on 6*
8	$k_3 : k_4 \wedge k_4 : b$	*Introducing a specific constant*
9	$k_3 : k_4$	$\wedge$ *-elimination on 8*
10	$k_4 : b$	$\wedge$ *-elimination on 8*
11	$\delta_p(q) : k_3$	$\wedge$ *-elimination on 4*
12	$\delta_p(q) : (k_3 + k_4)$	*Using definition 4 on 11, 9*
13	$(k_3 + k_4) : b$	*Using definition 5 on 9,10*
14	$a : \delta_p(q)$	*premise 1*
15	$a : (\delta_p(q) + k_3 + k_4)$	*Using definition 4 on 14,12*
16	$(\delta_p(q) + k_3 + k_4) : b$	*Using definition 5 on 12,13*
17	$a : (\delta_p(q) + k_3 + k_4) \wedge (\delta_p(q) + k_3 + k_4) : b$	$\wedge$ *-introduction on 15,16*
18	$\exists k \cdot a : k \wedge k : b$	$\exists$ *-introduction on 17*
19	$before(a, b)$	*Using definition 3 on 18*

Table A.10. The case when $before(\delta_i(j), \delta_p(q))$ holds

1	$before(\delta_i(j), \delta_p(q))$	*Assumption*
2	$\neg before(a, b)$	*Assumption*
3	$\exists k \cdot \delta_i(j) : k \wedge k : \delta_p(q)$	*Using definition 3 on 1*
4	$\delta_i(j) : k_6 \wedge k_6 : \delta_p(q)$	*Introducing a specific constant*
5	$\exists k \cdot a' : k \wedge k : b'$	*Restating*
6	$a' : k_5 \wedge k_5 : b'$	*Introducing a specific constant*
7	$a' : k_5$	$\wedge$ *-elimination on 6*
8	$k_6 : \delta_p(q)$	$\wedge$ *-elimination on 4*
9	$a' : k_5 \wedge a' : \delta_p(q) \wedge k_6 : \delta_p(q)$	$\wedge$ *-introduction on 7, premise 1 and 8*
10	$k_6 : k_5$	$\Rightarrow$ *-elimination using 9 on axiom 1*
11	$\delta_i(j) : k_6$	$\wedge$ *-elimination on 4*
12	$\delta_i(j) : (k_6 + k_5)$	*Using axiom 4 on 11,10*
13	$k_5 : b'$	$\wedge$ *-elimination on 6*
14	$(k_6 + k_5) : b'$	*Using axiom 5 on 10,13*
15	$\delta_i(j) : (k_6 + k_5) \wedge (k_6 + k_5) : b'$	$\wedge$ *-introduction on 12,14*
16	$\exists k \cdot \delta_i(j) : k \wedge k : b'$	$\exists$ *-introduction on 15*
17	$before(\delta_i(j), b')$	*Using definition 3 on 16*
18	$\neg overlaps(\delta_i(j), b')$	$\Rightarrow$ *-elimination on axiom 6 using 17*
19	$overlaps(\delta_i(j), b')$	*Using axiom 5*
20	$before(a, b)$	$\neg$*-introduction on 2 using 18 and 19*

References

[1] A. BARTSCH, H. EVEKING, H.-J. FÄRBER, M. KELELATCHEW, J. PINDER, AND U. SCHELLING. LOVERT - A logic verifier of register-transfer level descriptions. In *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design* (1989), North-Holland, pp. 522–531.

[2] AGERWALA, T. Putting Petri Nets to Work. *Computer 12*, 12 (December 1979), 85–94.

[3] ALLEN, J. F. An interval-based representation of temporal knowledge. *Proc. 7th Int. Joint Conf. on Artificial Intelligence* (Aug. 1981), 221–226.

[4] ALLEN, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM 26* (Nov. 1983), 832–843.

[5] ALLEN, J. F. Towards a general theory of action and time. *Artificial Intelligence 23* (1984), 123–154.

[6] BARTON, D. L. A functional characterization of elements of the VHDL simulation cycle. In *VHDL Users' Group Spring 1991 Conference* (Cincinnati, OH, April 1991), pp. 91–96.

[7] BICKFORD, M., AND JAMSEK, D. Formal specification and verification of VHDL. *Lecture Notes in Computer Science 1166* (1996), 310–317.

[8] BÖRGER, E., GLÄSSER, U., AND MÜLLER, W. A formal definition of an abstract VHDL'93 simulator by EA-machines. In *Formal Semantics for VHDL*, C.D. Kloos and P.T. Breuer, Eds., vol. 307 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Madrid, Spain, Mar. 1995, ch. 4.

[9] BORRIONE, D. Special Issue on VHDL Semantics. *Formal Methods in System Design 7*, 1/2 (Aug. 1995).

[10] BORRIONE, D., BOUAMAMA, H., DEHARBE, D., AND LE FAOU, C. HDL-Based integration of formal methods and CAD tools in the prevail environment. *Lecture Notes in Computer Science 1166* (1996), 450–458.

[11] BORRIONE, D., AND SALEM, A. Denotational Semantics of a Synchronous VHDL Subset. *Formal Methods in System Design 7*, 1/2 (Aug. 1995), 53–72. Kluwer Academic Publishers, ISSN: 0925-9856.

[12] BOULTON, R., GORDON, A., GORDON, M., HARRISON, J., HERBERT, J., AND VAN TASSEL, J. Experiences with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design: Theory, Practice and Experience*, V. Stavridou, T. F. Melham, and R. T. Boute, Eds. North-Holland, June 1992, pp. 129–156.

[13] BOYER, R. S., AND YU, Y. Automated correctness proofs of machine code programs for a commercial microprocessor. Technical Report UTEXAS.CS//CS-TR-91-33, University of Texas at Austin, Department of Computer Sciences, Nov. 1991.

[14] BREUER, P. T., SÁNCHEZ FERNANDEZ, L., AND DELGADO KLOOS, C. A Functional Semantics for Unit-Delay VHDL. In *Formal Semantics for VHDL*, C.D. Kloos and P.T. Breuer, Eds., vol. 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Madrid, Spain, Mar. 1995, ch. 2.

[15] CROW, J., OWRE, S., RUSHBY, J., SHANKAR, N., AND SRIVAS, M. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Apr. 1995. Available, with specification files, from `http://www.csl.sri.com/wift-tutorial.html`.

[16] DAMM, W., JOSKO, B., AND SCHLOR, R. A net-based semantics for VHDL. In *Proc. of the European Design Automation Conference with EURO-VHDL '93* (CCH Hamburg, Germany, September 1993), pp. 514–519.

[17] DAVIS, K. C. A denotational definition of the VHDL simulation kernel. *Proc. 11th Int. Symp. on Computer Hardware Description Languages* (1993), 509–521.

[18] DELGADO KLOOS, C., AND BREUER, P. T., Eds. *Formal Semantics for VHDL*, vol. 307 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Madrid, Spain, Mar. 1995.

[19] FIORE, M. P., JUNG, A., MOGGI, E., O'HEARN, P., RIECKE, J., ROSOLINI, G., AND STARK, I. Domains and denotational semantics: History, accomplishments and open problems. Tech. Rep. CSR-96-2, University of Birmingham, School of Computer Science, Jan. 1996. Available online at `ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSR-96-02.ps.gz`.

[20] FUJIMOTO, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (October 1990), 30–53.

[21] GARLAND, S. J., AND GUTTAG, J. V. A guide to LP, the Larch Prover. Tech. rep., TR 82, DEC/SRC, December 1991.

[22] GORDON, M. Why higher order logic is a good formalism for specifying and verifying hardware. *Formal Aspects of VLSI design* (1986), 153–177.

[23] GORDON, M. J. C. HOL: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Kluwer, Boston, 1988, pp. 73–128.

[24] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.

[25] HOARE, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[26] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1988.

[27] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1993.

[28] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.

[29] KNUTH, D. E., AND BENDIX, P. B. Word problems in universal algebras. In *Proceedings of a conference held at Oxford under the auspices of the Science Research Council, Atlas Computer Laboratory, 29th August to 2nd September, 1967* (1970), J. Leech, Ed., Pergamon Press Ltd, pp. 263–297.

[30] MARTIN, D. E., MCBRAYER, T. J., AND WILSEY, P. A. WARPED: A time warp simulation kernel for analysis and application development. In *29th Hawaii International Conference on System Sciences (HICSS-29)* (January 1996), H. El-Rewini and B. D. Shriver, Eds., vol. I, pp. 383–386. "Available on the WWW at `http://www.ececs.uc.edu/~paw/warped/`.

[31] MCBRAYER, T., AND WILSEY, P. A. Process combination to increase event granularity in parallel logic simulation. In *9th International Parallel Processing Symposium* (April 1995), pp. 572–578.

[32] MÜELLER, W. Approaching the denotational semantics of behavioural VHDL descriptions. In *Proc. of the 1st Asian Pacific Conference on Hardware Description Languages, Standards & Application* (December 1993), IEEE Press, pp. 1–4.

[33] N. SHANKAR, S. OWRE, AND J. M. RUSHBY. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[34] NAUR, P. Revised to the algorithmic language ALGOL-60. *Communications of the ACM 6* (1963), 1–20.

[35] OLCOZ, S. A formal model of VHDL using colored petri nets. In *Formal Semantics for VHDL*, C.D. Kloos and P.T. Breuer, Eds., vol. 307 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Madrid, Spain, Mar. 1995, ch. 5.

[36] OLCOZ, S., AND COLOM, J. M. Toward a formal semantics of IEEE standard VHDL 1076. In *EuroDAC*. IEEE Computer Society Press, Hamburg, Germany, 1993, pp. 526–531.

[37] OLCOZ, S., AND COLOM, J. M. Toward a formal semantics of IEEE std. VHDL 1076. In *Proceedings of the European Design and Automation Conference with EURO-VHDL '93* (Hamburg, FRG, Sept. 1993), R. Camposano, Ed., IEEE Computer Society Press, pp. 526–533.

[38] OLCOZ, S., AND COLOM, J. M. A Colored Petri Net Model of VHDL. *Formal Methods in System Design 7*, 1/2 (Aug. 1995), 101–124. Kluwer Academic Publishers, ISSN: 0925-9856.

[39] OWRE, S., SHANKAR, N., AND RUSHBY, J. M. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. A new edition for PVS Version 2 is expected in late 1996.

[40] OWRE, S., SHANKAR, N., AND RUSHBY, J. M. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. A new edition for PVS Version 2 is expected in late 1996.

[41] PETERSON, G., WILLIS, J., AND WILSEY, P. Advanced intermediate representation with extensibility. `http://www.ececs.uc.edu/~paw/aire/`, January 1997.

[42] R. CHAPMAN, AND DEOK-HYUN HWANG. A process-algebraic semantics for VHDL. In *SIG-VHDL Spring '96 Working Conference* (Dresden, Germany, May 1996), W. Ecker, Ed., Shaker Verlag, pp. 157–168.

[43] READ, S., AND EDWARDS, M. A Formal Semantics of VHDL in Boyer–Moore Logic. In *2nd International Conference on Concurrent Engineering and EDA* (San Diego, CA, 1994), SCSI.

[44] S. OWRE, N. SHANKAR, AND J. M. RUSHBY. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[45] SHANKAR, N., OWRE, S., AND RUSHBY, J. M. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. A new edition for PVS Version 2 is expected in late 1996.

[46] SOMMERVILLE, I. *Software Engineering*, 4th ed. Addison-Wesley Publishing Company, 1992.

[47] SPIVEY, J. M. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, Cambridge, 1988.

[48] UMAMAGESWARAN, K. PVS files on the framework for proving equivalences of vhdl descriptions, September 1997. Available on the WWW from `http://www.ececs.uc.edu/~kodi/masters_thesis/proofs/`.

[49] VAN TASSEL, J. P. A formalisation of the VHDL simulation cycle. Technical Report 249, University of Cambridge Computer Laboratory, March 1992.

[50] VAN TASSEL, J. P. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England, July 1993.

[51] VAN TASSEL, J. P. An operational semantics for a subset of VHDL. In *Formal Semantics for VHDL*, C.D. Kloos and P.T. Breuer, Eds., vol. 307 of *The Kluwer international series in engineering and computer science.* Kluwer Academic Publishers, Madrid, Spain, Mar. 1995, ch. 3.

[52] WILLIS, J. C., WILSEY, P. A., PETERSON, G. D., HINES, J., ZAMFIRESCU, A., MARTIN, D. E., AND NEWSHUTZ, R. N. Advanced intermediate representation with extensibility (aire). In *VHDL Users' Group Fall 1996 Conference* (October 1996), pp. 33–40.

[53] WILSEY, P. A. Formal models of digital systems compatible with VHDL, 1994. (available on the WWW at `http://www.ececs.uc.edu/~paw/rassp/`).

[54] WOODCOCK, J., AND DAVIES, J. *Using Z - Specification, Refinement, and Proof.* Prentice Hall, London, 1996.

Index