Technical Report

UCAM-CL-TR-317
ISSN 1476-2986

Number 317

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Femto-VHDL: the semantics of a subset of VHDL and its embedding in the HOL proof assistant

John Peter Van Tassel

November 1993

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 1993 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

ABSTRACT

The design of digital devices now resembles traditional computer programming. Components are specified in a specialised form of programming language known as a hardware description language. Programs written in such languages are then executed to simulate the behaviour of the hardware they describe. These simulations cannot be exhaustive in most situations, so result in high, yet incomplete, confidence that the proper behaviour has been achieved.

The formal analysis of programming languages provides ways of mathematically proving properties of programs. These properties apply to behaviours resulting from all possible inputs rather just than a subset of them. The prerequisite for such an analysis is a formal understanding of the semantics of the language.

The Very High Speed Hardware Description Language (VHDL) is currently used to specify and simulate a wide range of digital devices. The language has no formal mathematical semantics as part of its definition, hence programs written in it have not been amenable to formal analysis.

The work presented here defines a structural operational semantics for a subset of VHDL. The semantics is then embedded in a mechanical proof assistant. This mechanisation allows one not only to reason about individual programs but also to express equivalences between programs. Examples which highlight the methodology used in this reasoning are provided as a series of case studies.

ACKNOWLEDGEMENTS

None of this would have been possible without the financial assistance provided by Dr. John Hines at Wright-Patterson AFB through the United States Air Force Office of Scientific Research. John also gave generously of his time whenever help was needed with anything pertaining to VHDL. Additional financial support in the form of a bursary from Smith System Engineering was also appreciated.

The members of the Cambridge Hardware Verification Group have been instrumental in the successful completion of this work. Particular thanks go to Mike Gordon for supervising and putting up with me. Mike's ability to listen to the most silly ideas and slog through my horrible grammar is an argument for his immediate canonisation. Members of the group that should also be singled out are Juanito Camilleri for initial instruction in operational semantics and helping in getting to grips with how it might be applied to VHDL and Tom Melham for providing not only a sounding board for my strange ideas, but also some of the tools that made working with this stuff in HOL relatively painless. Thanks also to Juanito Camilleri, Mike Gordon, David Hemmendinger of Union College, Beth Levy of Aerospace and Bill Young of CLI who read parts of this opus and provided comments on making it better.

Last, but by no means least, the highest accolades should go to my parents for believing in and putting up with me over the years. Their unflagging support has been a great comfort both during the research and the writing up.

CONTENTS

CHAPTER 1

INTRODUCTION

This dissertation is devoted to the definition of a structural operational semantics for a subset of the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) and its embedding in the HOL proof assistant. The research is presented in seven chapters. The current one gives some motivation for the work, some background on the language and its uses, and acts as a synopsis of the material that follows in the rest of the document. The second chapter gives an overview of past and present research that is of a complimentary nature to that described here. Chapter 3 presents the subset while Chapter 4 defines its operational semantics. The fifth chapter contains an overview of the proof assistant being used and a discussion of how the semantics is embedded in it. Chapter 6 is made up of a series of case studies that show various ways of working with the embedded semantics. Finally, Chapter 7 finishes the exposition by making some concluding remarks and posing questions for future research.

1.1 Motivation

The complexity of current real-time systems such as those contained in modern avionics systems is so great that they cannot be completely tested. In many cases, there is no direct mechanical link between the pilot and the control surfaces. Onboard computers must interrupt the inputs being made by the pilot, match them against the current condition of the aircraft and make the necessary adjustments to these surfaces – hence the name "fly by wire". While the success of a mission or the lives of passengers depend on these computers, current testing methods cannot explicitly show that the equipment will respond correctly to all possible inputs.

The silicon integrated circuits used in inflight controls and most other modern applications are becoming so complex that exhaustive simulation of their behaviour before fabrication is a practical impossibility. In the case of aircraft controllers, this means that the devices are only functionally checked over critical inputs. While

current design and engineering practices tend to eliminate problems before they reach the marketplace, bugs do get into systems. It is becoming evident that a more secure way of creating electronic devices is needed in the future.

When VHDL arrived on the hardware design scene, it was described as the solution to many existing industry problems. Chief among these was to bring some order to the collection of languages and tools then in current use. But when VHDL itself was designed, very little thought was given to its formal semantics, much less to using such a semantics to reason about individual programs. The result has been a recent upsurge in research efforts addressing precisely these problems. The rest of this document is a discussion of research into the semantics of a subset of VHDL and ways to reason about program texts written in that subset using a mechanical proof assistant.

1.2 Philosophy

Emphasis has been placed on formalising the meaning of VHDL constructs as specified in the Language Reference Manual [21]. The relationship between these constructs and the simulation engine that evaluates them has been investigated. While taking this approach may seem overly pragmatic, it not only reflects the actual simulation behaviour of VHDL, but also corresponds intuitively to the way in which VHDL users think about the language. A different approach to characterising the semantics might have destroyed this link. For purposes of the current research, it has been necessary to utilise only a subset of full VHDL. Nevertheless, that subset is sufficient to demonstrate the most salient features of that model, and to provide methodological clues as to how such a formalisation might be used in practice.

1.3 Language History

VHDL is a hardware description language (HDL) currently in use by a large segment of the design community. It is a verbose, event-driven simulation language whose semantics is defined, at least informally in the Language Reference Manual, by the way in which the various language constructs are evaluated by the simulation engine [21].

To better understand VHDL and its place in current design practice, one should consider the evolution of the language:

early-1980's: Chaos

A time period characterised by the existence of many proprietary hardware description languages and simulators. Furthermore, no one HDL is used throughout the entire design process. In many cases, ad-hoc mechanisms are developed to translate between the tools and languages in use at different stages of the design. This leads to increased confusion and provides a further avenue for the introduction of bugs.

mid-1980's: VHSIC programme

The United States Air Force, a large consumer of custom electronics obtained from a variety of vendors, is faced with the need to rationalise its procurement process. The primary concern is that all designs submitted as a part of the bidding process are written in the same non-proprietary HDL, and meet certain design and documentation standards. VHDL emerges as the language, and is made a requirement for all Air Force designs.

late-1980's: IEEE VHDL (STD 1076)

As a part of the VHSIC programme, large vendors such as IBM and Texas Instruments take an interest in VHDL. That interest, coupled with the growing need for an industry standard HDL, causes VHDL to go through the IEEE standardisation process and emerge as the language in use today.

1.4 Application Areas

Since its introduction, the use of VHDL has spread to many areas of digital design. While most of these applications deal directly with design itself, there has also been recent interest in synthesis directly from VHDL. The language in its current state is not generally amenable to this task, but it is anticipated that modifications to the standard (VHDL'92) will overcome many of these problems. A number of commercial CAD vendors now support the language, and its use has spread from being a military systems language to one used and supported by a wide variety of design houses.

1.5 Evaluation of VHDL Programs

Given that a large part of the work to be presented is centred around the formalisation of the VHDL simulation loop, an informal description of the workings

of that loop and its associated parts is in order. It should however be noted that the discussion of VHDL which follows is rudimentary, and some familiarity with the language is assumed. The first step is to introduce a few definitions:

Signal: The name given to a communication channel in VHDL. For purposes of the presentation here, a signal is a wire.

Event: A change in the value of a signal

Transaction: The value that a signal should have at a particular time in the future. A transaction may (or may not) be converted into an event at that time.

Point of Computation: The point at which a particular collection of transactions is processed.

Process: A VHDL concurrent statement equipped with a set of signal names called a sensitivity list guarding its activation.

Simulation Cycle: The evaluation of all the processes in a VHDL program text at a particular point of computation.

Simulation Loop: Moving from point of computation to point of computation executing simulation cycles.

The top-level simulation loop of VHDL may be viewed as a four step sequence, and is illustrated in Figure 1.1. An iteration of the loop starts by moving forward to the nearest interesting point of computation (i.e. one where there are transactions to process), and setting the current simulation time, expressed as a natural number, to be the physical time unit associated with that point of computation. The state of the signal values is then modified to reflect those that they are supposed to take on during the present point of computation. Note that signals must have unique values in any given state. Those signals for which the update represents a change in value are then tagged as events, and a simulation cycle is performed making use of the new information. The loop is repeated until there are no more events or transactions to process, at which stage the simulation is said to have *quiesced*.

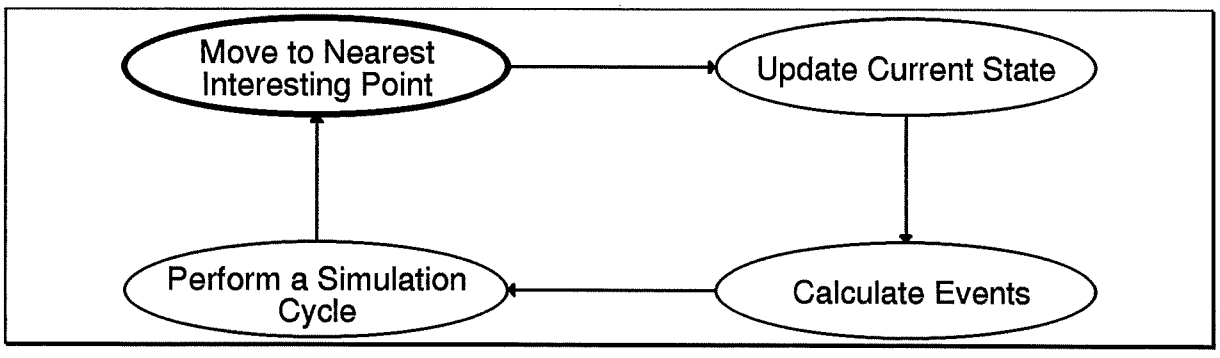


Figure 1.1: The Simulation Loop

The graphical description just presented may be summarised by the following pseudo-code description of the simulation loop:

```
while transactions remain to be processed
  1. go to the nearest point of computation with transactions to process
  2. update the state from the current transactions
  3. determine which updates represent events
  4. perform a simulation cycle based on the new state and events
end while
```

Emphasis should be placed on the use of the phrase "point of computation" in the above description. While one could often equate a point of computation with a time slice, it is frequently more than that within the context of the particular simulation model of VHDL. One could, for instance, say that the points of computation P_1 through P_n represent the time units 1 through n. Alternatively, they could represent 1 through n 0-length delays between two major time units.

The concept of δ-delay is the way in which VHDL deals with these 0-length delays. They result from 0-delay signal assignments. A static state of the world is being used during each iteration through the simulation loop, and transactions are being scheduled to occur at some future point of computation whenever a signal assignment is encountered. This scheduling applies to any transaction, whether it is to occur at a delay offset zero units from now, or at some larger one. Each time around the simulation loop (Figure 1.1) in zero time is called a δ-delay. An individual δ step does *not*, therefore, represent a quantifiable unit of time. Rather it is simply a simulator artefact that ensures consistency in the ordering of events.

As mentioned earlier, a component of the simulation loop is the performance of a "simulation cycle". The execution of such a cycle is illustrated in Figure 1.2. Each cycle begins with the activation of those processes that are sensitive to any of the current events. All the active processes are then run in parallel, each taking a copy of the static simulation environment with it. During its execution, each process may schedule transactions to be processed in the future. When all the processes have terminated, the futures that have been calculated by the individual processes are gathered together into an amalgamated view of future behaviour.

Figure 1.2: Progression of a simulation cycle

The above steps are again more concisely stated as:

- determine which processes are active based on current events
- run the active processes in parallel
- merge all processes' scheduled transactions into a collective whole

1.6 Formalisation

The above informal description of the simulation model of VHDL has been translated into an operational semantics and embedded in a mechanical proof assistant. Due to the size of the language and a desire to emphasise the workings of the simulation model of VHDL, a subset has been chosen. In following this course of action, it has not only been possible to prove theorems about individual program texts, but also to demonstrate properties of the semantics itself that increase one's confidence in it. Some of these theorems will be presented in later chapters, and were developed with the aid of the HOL proof assistant. Because of their mechanical nature the proofs of these theorems are not readily understandable, and are therefore not given in the text. Where necessary, a proof sketch will be given to argue for the correctness of the result. The scripts that were used to prove the theorems are presented in Appendix A.

Operational semantics, as the name implies, allows for the specification of and reasoning about dynamic systems. Specifically, it makes it possible for one to define an abstract representation of an interpreter for a particular language. Given the interpreter, it is not only possible to reason about individual programs, but also infer properties for groups of similar programs. In the context of a semantics for VHDL, an operational framework makes for an easier correspondence between the informal and formal descriptions of the simulation model of the language.

1.7 Automation

Once defined, the syntax and semantics of the VHDL subset is embedded in the proof assistant. In order to make use of the embedding a suite of proof tools was developed to animate the semantics via proof. In essence, this leads to a very slow simulator for the subset.

This simulator itself is not, however, a typical VHDL simulator. Certainly, one may use it to execute programs containing concrete values for signals. The difference lies in the fact that one may also use it to perform simulations with symbolic values for these same signals. In that case, the execution of the simulation

model is providing information on the behaviour of a program over all values. A discussion of the proof assistant and the embedding appears in Chapter 5.

1.8 Examples

By embedding the semantics in a proof assistant, it has been possible to explore more fully the relationships between VHDL programs. This exploration has led to a way of combining symbolic simulation with proof to give stronger assurances than is usually possible in a pure simulation environment about the equivalence of designs. While large designs are impractical to analyse at present in the described manner, the methodology still gives useful insight into the workings of the VHDL simulation engine.

The examples themselves are presented as a series of case studies in Chapter 6. They range in complexity from the analysis of a simple NAND gate to a parity-checking device. In each of the case studies, the following pattern is used:

1. Specify a high-level (i.e. algorithmic) representation of the device.
2. Derive, using the semantics, a general behaviour for the specification.
3. Specify a low-level implementation of the device.
4. Derive a general behaviour for the implementation in the same fashion as 2.
5. Prove that the two behaviours are the same.

Why perform several examples of exactly the same form? The answer lies in the points that each raises about the simulation model and the kind of results one can expect from it. The issues range from how δ-steps are resolved to compromises that must be made in order to get a meaningful result from the embedded semantics.

1.9 Accomplishments

The work in the chapters that follow represents research in establishing a formal semantics for a subset of VHDL. The semantics is useful in and of itself as an unambiguous specification of the language. Furthermore, the semantics has been embedded in a mechanical proof assistant to provide insight into how it might be used in reasoning about VHDL designs. This reasoning will be seen as a change from traditional simulation in that it moves the examination of programs from validation based on concrete inputs to verification based more general, symbolic ones.

CHAPTER 2

RELATED WORK

Efforts at applying formal methods to VHDL programming fall into two categories. The first deals strictly with increased support for simulation-time validation of programs. The second pertains to the search for and use of a tractable semantics for a reasonable subset of the language. The rest of this chapter presents a broad overview of these approaches.

2.1 Simulation-Time Validation

The late 1980's was a period of great activity in research surrounding the development of simulation-based validation tools. This particular type of system was meant to provide the designer with additional assurance that a given program was behaving in the intended way through clever use of the simulator itself. Because it followed shortly after the introduction of VHDL as a standard in the United States, the research into these methods was carried out there.

One of these investigations led to the development of the VHDL Annotation Language (VAL) at Stanford [4,5]. VAL allowed the user to decorate a program with specifications about its operation. These annotations appeared as comments in the VHDL program text, and were expanded by a pre-processor before the program was compiled and simulated. When the program was executed, the annotations triggered warning messages if the design differed in its execution from the implicit specification that they provided. This particular methodology was not only a way of getting more useful information out of a simulation, but also made it easy for clear specifications of the behaviour of various parts of a design to be embedded in program texts.

Another early project, conducted by the author, attempted to automatically generate VHDL assertions characterising the behaviour of programs [36,37]. The philosophy was much the same as in the VAL effort. The difference was that it was based on automatically ascertaining the characteristic statement of behaviour for a

particular program. These assertions could then be embedded in the text of the program for use during simulation in much the same way as VAL specifications. The notion was that once the behaviour of the canonical version of a device had been agreed upon, a statement describing its behaviour could be extracted automatically and embedded in any other program purporting to be the same device. Any exception raised during simulation would therefore indicate some divergence from the accepted norm which needed to be investigated.

Both these projects were based on the respective researcher's interpretation of the semantics of VHDL. No formal statement about the semantics of the language was available at the time beyond that given in the Language Reference Manual. Furthermore, they relied on traditional simulation practice. While this was not a bad thing in that it brought a degree of rigour to the examination of the simulation model of VHDL, these research efforts did not result in any definitive mathematical statement about the semantics of the language.

2.2 Semantics-Based Verification

More recent research has been directed at the verification, rather than validation, of VHDL programs. To accomplish their goals, all the projects involved in this area have necessarily had to build up a formal notion of the semantics of VHDL. The research efforts currently under way vary not only in their approaches to the particular problem of semantics, but also in their implementation of it. Furthermore, they are characteristic of emerging international interest in the field. One camp bases their approach on primarily denotational methods. The other is grounded in a more dynamic intuition.

2.2.1 Denotational Work

Early work in discerning a semantics for signal attributes emerged from IMAG in Grenoble [33]. The emphasis was to statically understand the simulation behaviour associated with the attributes of individual signals. The work eventually led to a prototype system for reasoning about a subset of VHDL similar to that chosen here. A drawback was that because of its insistence on a unit-time model, 0-delay signal assignments could pose problems in the framework.

Promising early work on a full denotational description of the VHDL simulation kernel is underway at the University of Cincinnati [16]. The research is again based upon a small subset of VHDL that is not dissimilar to that presented

here. The work in Cincinnati has lead to the development of valuation functions for initialisation and state transformation during a simulation. There have, however, been no proofs of properties of the semantics to emerge from the effort as of yet.

2.2.2 Operational Work

Research closer in flavour to that presented here is underway at several sites. These projects focus on an operational formalisation of what is specified in the Language Reference Manual. Even within this unanimity of spirit, the approaches differ in practice. One type of formalisation is based on extra-logical manipulation of the semantics which passes the results to a proof assistant for additional manipulation. The other does all the work in whatever logical system has been chosen.

Use of an Oracle

Research at Siemens [35] is characteristic of the first approach. The simulation model of VHDL is written in the functional language ML. The results of working with a program inside this specification of the simulator are then passed to the LAMBDA system for reasoning. LAMBDA is a proof assistant from Abstract Hardware, Ltd. that is based on the same logic as the system used here. The difference is that it has been specially tailored for reasoning about hardware. In essence, the ML program is being used as an oracle providing a statement of behaviour for the theorem-prover to use in subsequent proofs. The goal of the Siemens project has been to provide an industrial-strength tool, and the subset of the language is therefore very large. The approach chosen, while not strictly formal in that the semantics of the language is specified by an interpreter written in a functional language rather than securely inside the logic of LAMBDA, has merits in that it provides an efficient environment for working with VHDL programs. The limitation is that one must be convinced of the correctness of the external oracle before trusting any results that arise from it and the subsequent proof process.

Secure Formalisations

The other approach, namely that of modelling the semantics of VHDL inside the mechanisation of the particular logical system being used, is much closer in flavour to the work that follows in later chapters. The important difference between it and the material summarised here is that the formalisations are expressed in (and are

therefore perhaps limited by) the particular logic of the given proof environment. The research presented in this document makes use of a semantics that has been developed in an accepted semantic formalism without regard as to how it is to be mechanised. The generality of the target proof environment then allows one to seamlessly embed the semantics to reason not only about it but VHDL programs as well.

The most substantial piece of work in this field is the result of research at the Aerospace Corporation. The semantics of the simulation kernel have been implemented in the State Delta Verification System. SDVS is a framework for describing the execution of individual program steps based on the way that they affect the overall state in which the program executes [1]. The subset implemented is also large [17]. The idea is much the same as in the Siemens project (i.e. develop the meaning of a particular program based on a specification of the simulation model and then reason about it). Again, the contrast here is that the work is being carried out without recourse to an external oracle. The problem is not therefore in trusting both the oracle and the interface between two systems, but in gaining confidence in the formal specification of the VHDL simulation kernel.

Other similar research is just beginning at Computational Logic, Inc. [39]. The idea is much the same as in the Aerospace project. The system will be implemented in the Boyer-Moore logic [11] as realised in the NQTHM theorem-prover [12]. The Boyer-Moore system differs from both SDVS and the proof development environment that will be used later in that it is a much more automatic system, requiring less user intervention.

2.2.3 Other Approaches

Research into verification environments for VHDL is not limited to those methodologies given in Sections 2.2.1 and 2.2.2. Other projects make use of either custom-designed or existing formal languages for their implementations. One is underway at Royal Holloway and Bedford New College where a language called FUNNEL is being developed [34]. The language is intended as a descriptive medium for many different hardware description languages, and therefore provide a framework for reasoning about designs irrespective of the language in which they were specified. Another approach is being taken at IRISA [7] using the language SIGNAL to describe the simulation kernel of VHDL and to rationalise its timing model.

CHAPTER 3

STATIC SEMANTICS

Before embarking on a formal discussion of the way VHDL simulates, the subset of the language that will be addressed needs to be identified. Furthermore, it will be necessary to define a way of checking that particular programs written in that subset are properly constructed. The exposition that follows begins by introducing the syntax of the subset and informally discussing the kind of restrictions that have been placed upon it. The chapter concludes with a suite of definitions that formalise the restrictions about the well-formedness of VHDL programs written in the subset.

3.1 Femto-VHDL

VHDL is a language that can best be thought of as one whose execution is based upon that of concurrent blocks. The most basic of these is the `process` statement. Each block contains sequential statements that describe the algorithm that it is supposed to be implementing. To a large extent, these sequential statements are exactly those of sequential Ada[1]. Only one particular class of sequential statements, signal assignments, is essential for driving the simulation. The emphasis in the definition of the subset will therefore be to highlight these two important types of statement (processes and signal assignments), and to provide enough other syntactic machinery to ensure that the semantics fully exercises them.

The VHDL subset for which a semantics has been defined is therefore rather small. Its name is derived from the smallest user-addressable unit of time in full VHDL – the femtosecond. Femto-VHDL does, however, contain enough constructs of the full language to illustrate the important features of the simulation model. As just discussed, only one kind of VHDL concurrent statement is supported, namely `process` statements. These are further restricted to those equipped with explicit sensitivity lists of signal names, which can be thought of as guards on their

[1]Ada is a registered trademark of the U.S. Government – Ada Joint Program Office.

activation. The sequential statements addressed are the `if-then-else` conditional, as well as inertial and transport delay signal assignments. Architectural hierarchies allow the user to group concurrent statements into large blocks with specific inputs and outputs. When simulated, these hierarchies will be flattened out into a sea of concurrent processes. For purposes of the subset, the architectures here are limited to those without VHDL `generic` [21] statements. Furthermore, the following assumptions are made:

- Signals may only be Boolean-valued. This is for simplicity, and does not materially affect the exposition of the simulation model.
- Resolved (multiply driven) signals are not allowed. The implication is that constructs such as wired-and's and wired-or's are not permitted.

Why these particular assumptions? The first one is understandable in that assignment to Boolean-valued signals is the same as assignment to signals of any other value. The focus here is on the act of assignment, not on the type of the assigned-to signal. The second assumption is a bit more cryptic. In essence, it ensures that no signal may be assigned to from more than one `process` statement. If this was allowed, two active processes might try to assign to the same signal at the same time causing a clash over which value that signal should take on. Full VHDL provides a way of dealing with this eventuality through the use of user-defined resolution functions [21]. In an effort to keep the discussion of the simulation model simple, these particular types of actions have been dis-allowed. As will be seen in Chapter 4, it is possible to add resolved signals to the semantics, but an extension is required.

Given the above listing of supported statements and assumptions, one can ask what physically constitutes a Femto-VHDL program. At the outermost level, a program text again is made up of one or more architectural components. Each component is comprised of either one or more concurrent statements, or the nesting of more components. The concurrent statements in turn contain sequential statements. The following abstract syntax for Boolean expressions, sequential statements and concurrent statements in Femto-VHDL may be given:

bexp	::=	*signal* | *signal*'delayed(*n*) | *signal*'event | not *bexp* | *bexp* and *bexp* | *bexp* or *bexp* | *bexp* nand *bexp* | *bexp* nor *bexp* | *bexp* xor *bexp* | *bexp* xnor *bexp* | true | false
ss	::=	*ss* ; *ss* | null | *bexp* => *ss* | *ss* | *string* $:=^{\text{inert}}$ (*bexp*,*n*) | *signal* $:=^{\text{trans}}$ (*bexp*,*n*)
cs	::=	*cs* || *cs* | *sl* : *ss*
sl	::=	$\{s_1,\ldots,s_n\}$

The symbol *bexp* ranges over Femto-VHDL Boolean expressions. These are made up of single signal names, attributed signals ('event and 'delayed), or compound expressions using the various VHDL Boolean operators. Attributes allow the user to access properties of signals. The ones in Femto-VHDL are used for ascertaining whether or not a signal has an event on it now ('event) and for looking at the value of a signal delayed by n units of time expressed as a natural number ('delayed). Sequential statements (*ss*) may be either two statements in sequence (;), a null statement, a conditional statement guarded by some Boolean expression, an inertial delay signal assignment statement, or a transport delay signal assignment statement. Each type of signal assignment statement maps a Boolean value to a signal after some delay also expressed in terms of the naturals. Concurrent statements (*cs*) are either two such statements in parallel, or a single process statement made up of a sensitivity list and some sequential statements. Sensitivity lists (*sl*) are simply sets of signal names.

The overall structure of design hierarchies is expressed as:

design	::=	arch $string_e$ $string_a$ *ports decls cs* | dpar *design design*
decls	::=	signal $\{s_1,\ldots,s_n\}$

Local signal declarations may be made (*decls*), and do not have associated directions. Ports (*ports*) are the external interfaces for VHDL architectures. They have associated directions, and are represented in Femto-VHDL, by a triple of sets. Signals of direction `in` (inputs) are in the first part of the triple, `out` signals (outputs) are in the second part and `inout` signals (bi-directionals) are in the final part. A design is composed of either a stand-alone component or a nested hierarchy of them. It should be noted that neither || nor dpar have any corresponding construct in the concrete syntax of VHDL. They are given here simply as a means of grouping together language constructs.

VHDL components may be specified at different levels of abstraction. To this end, the concept of design *entity* and design *architecture* are available to the designer. An entity is nothing more than a specification of connections to the outside world via ports. For purposes of Femto-VHDL, an architecture groups together concurrent statements that express the behaviour of a particular component (e.g. an ALU). Two particular architectures may actually represent the same component at different levels of abstraction. In that case the architectures are really part of the same entity, as they make use of the same ports to plug into the outside world irrespective of their internal construction. The two strings $string_e$ and $string_a$ in the arch part of the

definition of *design*, represent the name of the entity and the name of the architecture respectively. It is therefore entirely possible for individual components to have the same entity name, but different architecture names.

The following tables give the relationship between the abstract syntax and the concrete syntax for various syntactic classes of Femto-VHDL. No mapping will be shown for Boolean expressions, as the abstract and concrete syntaxes are the same. For reasons of brevity the abstract syntax will be used in describing the semantics of Femto-VHDL.

	design
arch $string_e$ $string_a$ *ports decls cs*	entity $string_e$ is port *ports*; end $string_e$; architecture $string_a$ of $string_e$ is signal $s_1,\ldots,s_n$; begin *cs* end $string_a$;

	cs
$\{s_1,\ldots,s_n\}$: *ss*	process $(s_1,\ldots,s_n)$ begin *ss* end process;

	ss
$signal \overset{\text{inert}}{:=} (bexp,n)$	*signal* <= *bexp* after *n*;
$signal \overset{\text{trans}}{:=} (bexp,n)$	*signal* <= transport *bexp* after *n*;
null	null;
bexp => *ss* | *ss*	if *bexp* then *ss* else *ss* end if;

3.2 Definitional Style

When defining functions and relations associated with the semantics, recourse will be made to the following definitional style. The symbol ≡ indicates a definition. The function name, when first defined, will appear in **bold sans serif** typeface. Variables will be shown in *math italic*. Internal functions and variables will be defined using let ... (and ...) in constructions. As an example, the function foo may defined as:

```
foo x ≡
  let a = x + 1 and b = x − 1 in
  let c = a + b in
    (x, c)
```

The function takes a natural number as its argument and returns a pair of naturals. The nesting of let-expressions allows for the calculation of c. Scoping is such that a and b are not visible to one another, but are visible in the second let-expression when c is formed. If one expanded the first let-expression, the result would be:

```
foo x ≡
  let c = (x + 1) + (x − 1) in
    (x, c)
```

Use will also be made of the form if x then ... (elseif y then ...) else ... to express conditionals. Lists will be required in some of the definitions that come later, and list processing functions such as hd (head), tl (tail) and . (cons) will therefore be needed as well. Finally, it will be necessary to make use of tuples in some definitions, and the functions fst and snd will be used to respectively access the first and second elements of them.

3.3 Syntactic Well-Formedness

What constitutes a syntactically valid Femto-VHDL program? First, output signals must not have multiple drivers. This means that a signal must not be assigned to from more than one source. Next, the signals used in the program text must be only those that happen to be declared as ports or as local signal declarations. Finally, those signals must also be assigned to in an appropriate manner (i.e. signals of direction `in` must not be driven, and signals of direction `out` must not be read). The rest of the chapter involves a series of definitions leading up to the function well_formed_design which encompasses these properties. Readers not interested in the details of these definitions may safely progress to Chapter 4.

3.3.1 Multiple Drivers

To check the multiple driver condition, it is necessary to determine that a given signal name appears on the left-hand side of the signal assignments of only one process in the program text. In other words, if the set of assigned-to signal names for one process of the design has a non-empty intersection with a similarly derived

set for any other process in the design, the program text is not well-formed. At the lowest level of this check, one needs to extract a set of signal names that are assigned a value in the statements of a given process. Having done so, the requirement is then to make sure that these signals are not being assigned to in any other process.

As the first step in the check, one defines the function well_formed_Femto_ss over the structure of sequential statements:

well_formed_Femto_ss $ss \equiv$
if $(ss = ss' \,;\, ss'') \vee (ss = e \Rightarrow ss' \mid ss'')$ then
(well_formed_Femto_ss ss') $\cup$ (well_formed_Femto_ss ss'')
elseif $\left(ss = s :=^{\text{inert}} (e, dly)\right) \vee \left(ss = s :=^{\text{trans}} (e, dly)\right)$ then $\{s\}$
else $\{\}$

The function returns, as a set, the signal names that appear on the left-hand side of any assignment statement. For instance, well_formed_Femto_ss would return the set $\{a,b\}$ if given the conditional statement $e \Rightarrow a :=^{\text{inert}} (x,2) \mid b :=^{\text{trans}} (y,0)$.

Use may now be made of well_formed_Femto_ss in gathering together a collection of assigned-to signals for all the processes in a given architecture. The function well_formed_Femto_cs, again defined recursively on structure, does just that. It returns a tuple of sets. The first element of the pair represents the intersection that is of interest, while the second is the set of all assigned-to signals.

well_formed_Femto_cs $cs \equiv$
if $cs = (sl : ss)$ then $(\{\}, \text{well\_formed\_Femto\_ss } ss)$
else let $(cs' \,||\, cs'') = cs$ in
let a = snd(well_formed_Femto_cs cs')
and b = snd(well_formed_Femto_cs cs'') in
$(a \cap b, a \cup b)$

If the two processes $\{x\} : a :=^{\text{inert}} (x,2) \,||\, \{y\} : b :=^{\text{trans}} (y,0)$ were used as an example, the function would return the pair $(\{\},\{a,b\})$. However, had both signal assignments been addressing signal a, the resulting pair would have been $(\{a\},\{a\})$.

Having defined the two previous functions, all that remains to be done is ascertain the emptiness of the first element of the pair returned by well_formed_Femto_cs. The function wfF_csss is defined as a top-level interface, and makes the desired comparison:

wfF_csss $cs \equiv (\text{fst}(\text{well\_formed\_Femto\_cs } cs) = \{\})$

If the just-used concurrent statements were passed as arguments to wfF_csss, it is clear from the above results that an invocation on the processes which assign to both a and b would fulfil the multiple driver restriction, while one where both processes address the signal a would not.

In order to make use of wfF_csss for larger components, it will be necessary to flatten out architectural hierarchies into a sea of processes. To this end, the function dflat has been defined. The function takes one or more Femto-VHDL components, and returns only the processes associated with each one. In the case of multiple architectures, the processes of each are glued together in parallel.

dflat *design* ≡
  if *design* =arch *ent arch ports decls cs* then *cs*
  else let (dpar d' d'') = *design* in (dflat d') || (dflat d'')

For instance, the design dpar (arch e' a' p' d' cs') (arch e'' a'' p'' d'' cs'') would, when passed through dflat, be reduced to $cs' \,||\, cs''$.

3.3.2 Signal Relationships

The other restrictions on the syntactic well-formedness of Femto-VHDL programs hinge heavily on the interrelationships of the signals as they are found in individual components. In order to make any statement about those relationships, it will first be necessary to extract the necessary signals from a program text. Having done so, it is then possible to go on to make a definition of what is required of those signals in the overall design.

Signal Extraction

When dealing with Femto-VHDL designs, two kinds of signals are of interest. One group is the external ports. The other is the locally declared signals. The function get_sigs_decls accesses the set of locally declared signals in a stand-alone component.

get_sigs_decls (signal *sigs*) ≡ *sigs*

For instance, if the function were applied to the declarations of arch e a p (signal d) cs, it would return the set d. The ports of a design may be extracted with get_ports. The function is defined recursively over the structure of designs in an analogous manner to dflat.

get_ports *design* ≡
  if *design* =arch *ent arch ports decls cs* then
    (fst *ports*) ∪ (fst (snd *ports*)) ∪ (snd (snd *ports*))
  else let (dpar d' d'') = *design* in (get_ports d') ∪ (get_ports d'')

The whole idea of the function is to remove any reference to directionality, and return as a set only the names of the ports of a particular design.

The above functions extract the declared signals in a design. However, in order to make the kind of checks that are necessary, functions that return the signals that are actually used in the concurrent statements of a design are also required. A series of functions that extract these signals, paying particular attention to their intended direction, are now defined.

At the lowest level, the signals will need to be returned from Boolean expressions. Because these expressions are only capable of holding input signals (i.e. they are always read, but never written to), all that is required in get_sigs_bool is to gather together any signal names found in a Boolean expression, and return them as a set.

get_sigs_bool *bexp* ≡
  if (*bexp* = not e) then (get_sigs_bool e)
  elseif (*bexp* = e' and e'') ∨ (*bexp* = e' or e'') ∨ (*bexp* = e' nand e'') ∨
      (*bexp* = e' nor e'') ∨ (*bexp* = e' xor e'') ∨ (*bexp* = e' xnor e'') then
    (get_sigs_bool e') ∪ (get_sigs_bool e'')
  elseif (*bexp* = *sig*) ∨ (*bexp* = *sig*'delayed(t)) ∨ (*bexp* = *sig*'event) then {*sig*}
  else {}

Sequential statements contain both input and output signals. The only output signals are those on the left-hand side of signal assignment statements. All other signals are assumed to be inputs. The function get_sigs_ss returns a pair of sets, the first of which is the set of input signals (including those found in any Boolean expressions), while the second is the set of output signals.

get_sigs_ss *ss* ≡
  if (*ss* = ss' ; ss'') ∨ (*ss* = e => ss' | ss'') then
    let (is', os') = get_sigs_ss ss' and (is'', os'') = get_sigs_ss ss''
    and *bools* = if (*ss* = e => ss' | ss'') then get_sigs_bool e else {} in
      ($is' \cup is'' \cup bools$, $os' \cup os''$)
  elseif (*ss* = s :=$^{\text{inert}}$ (e, *dly*)) ∨ (*ss* = s :=$^{\text{trans}}$ (e, *dly*)) then (get_sigs_bool e, {s})
  else ({}, {})

Finally, it is necessary to traverse an entire sea of concurrent processes gathering the same information. The function get_sigs_cs does this, and returns a pair of sets of the same form as those created by get_sigs_ss.

get_sigs_cs $cs \equiv$
  if $cs = (sl : ss)$ then get_sigs_ss ss
  else let $(cs' \,||\, cs'') = cs$ in
    let $(is', os') =$ get_sigs_cs cs' and $(is'', os'') =$ get_sigs_cs cs'' in
      $(is' \cup is'', os' \cup os'')$

As an example of the behaviour of this suite of functions, take the Femto-VHDL processes:

$$\{x\} : a :=^{\text{inert}} (\text{not } x, 2) \,||\, \{y'\} : b :=^{\text{trans}} (y' \text{ and } y'', 0) \,||\, \{z\} : c :=^{\text{trans}} (z, 0)$$

When get_sigs_cs is run over them, the pair $(\{a,b,c\},\{x,y',y'',z\})$ is returned.

A companion function to get_sigs_cs is get_sigs_design. It extracts all the signal names from a design without regard for their associated direction by making use of get_sigs_cs.

get_sigs_design *design* $\equiv$
  if *design* =arch *ent arch ports decls cs* then
    let $(is, os) =$ get_sigs_cs *cs* in $is \cup os$
  else
    let (dpar d' d'') = *design* in (get_sigs_design d') $\cup$ (get_sigs_design d'')

In a similar fashion to the previous example, the architectures

$$\text{dpar}\left(\begin{array}{l} \text{arch } e' \; a' \; p' \; d' \left(\{x\} : a :=^{\text{inert}} (\text{not } x, 2)\right) \\ \text{dpar}\left(\begin{array}{l} \text{arch } e'' \; a'' \; p'' \; d'' \left(\{y'\} : b :=^{\text{trans}} (y' \text{ and } y'', 0)\right) \\ \text{arch } e''' \; a''' \; p''' \; d''' \left(\{z\} : c :=^{\text{trans}} (z, 0)\right) \end{array}\right) \end{array}\right)$$

would yield the single set $\{a,b,c,x,y',y'',z\}$ when put through get_sigs_design.

Well-Formedness of Design Signals

In order to ensure the well-formedness of the signals in a Femto-VHDL program, the various constraints on signals mentioned earlier must be met. To reiterate, they were:

- Only declared signals may be used in the program text.
- Directionality of assignments must be preserved.

The function well_formed_design_sigs uses the just defined functions to express these goals for the well-formedness of the signals in a Femto-VHDL design.

well_formed_design_sigs *design* ≡
  if *design* = arch *ent arch ports decls cs* then
    let *ip* = fst *ports* and *op* = fst (snd *ports*) and *op* = snd (snd *ports*)
    and $(isigcs, osigcs)$ = get_sigs_cs *cs* and *d* = get_sigs_decls *decls* in
    let $p = (ip \cup op \cup iop)$ and $sigcs = (isigcs \cup osigcs)$ in
    let $pId = (p \cap d = \{\})$ and $oIi = (op \cap isigcs = \{\})$ and $iIo = (ip \cap osigcs = \{\})$ in
      $(sigcs = p \cup d) \wedge pId \wedge oIi \wedge iIo$
  else
    let (dpar d' d'') = *design* in
    let (P', S') = well_formed_design_sigs d'
    and (P'', S'') = well_formed_design_sigs d'' in
      $(P' \wedge P'')$

In the case of stand-alone components, *ip* are the `in` ports, *op* are the `out` ports and *iop* are the `inout` ports. *isigs* represents the signals that are used as inputs in the concurrent statements of the component, whilst *osigs* are those used as outputs. *d* are the locally declared signals. The variable *p* holds all the ports, and *sigcs* contains all the signals of the concurrent statements. *pId* states that the ports and locally declared signals must be disjoint, *oIi* says that the output ports must not be used as inputs, and *iIo* ensures that input ports are not being used as outputs. Conjoining these last three with the statement that the ports of the component are the same as the signals used by the concurrent statements ensures that the all the signals of that component are well-formed.

A useful consequence of the definition of well_formed_design_sigs when reasoning about the dynamic semantics of Femto-VHDL in Chapter 4, will be the knowledge that if the signals in a design display the proper interrelationships, then the ports of the design are a subset of all the signals in that design.

Theorem 3.3.2.1:
$\forall dsgn.$ (well_formed_design_sigs $dsgn$) $\supset$ (get_ports $dsgn \subseteq$ get_sigs_design $dsgn$)

The proof was performed in the theorem prover by induction on the structure of designs. This proof, because of its mechanical nature, is not given here; but the argument is fairly straightforward. The interested reader is referred to Appendix A for the actual proof script. The base case follows from an expansion of the definition of well_formed_design_sigs. Once done, one observes that the ports and declarations of the component must be the same as the signals of its concurrent statements (derived by get_sigs_cs). This of course means that the ports are a subset of the total signals. The induction step is also a based on a straightforward expansion of definitions. Once those of well_formed_design_sigs, get_ports and get_sigs_design have been applied to composite designs one arrives at an instance of the induction hypothesis.

3.3.3 Overall Well-Formedness

It is now possible to make an overall definition about the syntactic well-formedness of any Femto-VHDL design. Simply stated, the signals of the design must exhibit the appropriate relationships, and the concurrent statements must not contain multiply driven signals. These requirements are formally defined in the function well_formed_design:

well_formed_design $design \equiv$
(well_formed_design_sigs $design$) $\wedge$ (wfF_csss (dflat $design$))

A simple fact arising from the definition of well_formed_design is:

Theorem 3.3.3.1:
$\forall design.$ well_formed_design $design \supset$ well_formed_design_sigs $design$

The intuition for the proof is to assume well_formed_design $design$, expand the definition and take the first conjunct.

CHAPTER 4

DYNAMIC SEMANTICS

The only official specification of the meaning of VHDL is the Language Reference Manual. It describes the semantics of the language in an informal manner through recourse to terms such as "elaboration" and "execution". These are words that implicitly lead to a formulation of meaning in terms of simulation. The material that follows is a formalisation of the semantics of Femto-VHDL that reflects this dynamic intuition. The discussion will begin by presenting a short overview of operational semantics, and will continue with a presentation of how information will be structured in the semantics. The definitions concerning the well-formedness and equivalence of some of those structures will then follow. The chapter will conclude with the definitions and rules that make up the semantics of the language.

4.1 Operational Semantics

The semantics of Femto-VHDL will be given in an operational manner [32]. As alluded to earlier, operational semantics it is a formalism that allows for the specification of and reasoning about dynamic systems. Since VHDL is defined even in the informal context of [21] by means of the simulation process, the idea of using this particular semantic approach is intuitively appealing. In order to define the semantics in an operational way, it will be essential to understand a special form of definition. An example taken from [20] is perhaps the easiest way of coming to grips with what is required.

4.1.1 Inductive Definitions

If one rephrases mathematical induction as a more generally applicable form of induction, the set N of natural numbers satisfies the following conditions (or rules):

- $0 \in N$
- if $x \in N$ then $x+1 \in N$

Many other sets also have these properties. Examples are the set of integers (positive and negative), the set of rational numbers and the set of real numbers. However, these properties have special significance for N; N is the *least* set which has both these properties. This means that if X is any other set with these properties then $N \subseteq X$. To prove this one uses mathematical induction to show that $n \in X$ for every natural number n. More precisely, mathematical induction is employed to show the property $P(n)$ is true for every natural number n where $P(x)$ is defined by:

$P(x)$ iff $x \in X$

The base case, $0 \in X$, is trivially true because X satisfies the first property. The fact that X satisfies the second property means that the inductive step is also true: $k \in X$ implies $k+1 \in X$. One can therefore conclude that $n \in X$ for every natural number n, i.e. $N \subseteq X$. In general many sets, relations, etc. will be defined as the least ones satisfying a set of conditions or rules. Each such definition gives rise to a form of induction, called *Rule induction*, for proving properties of the sets, relations, etc. For this reason these are called *inductive definitions*.

4.1.2 Inductive Definitions and Femto-VHDL

The definitions that will be used in the semantics of Femto-VHDL are made up of sets of rules expressed as conditionals:

if **premises** then **conclusion**

Where the premises may take on one of two forms, those associated with the relation being defined and other truths which must be known. These other truths are known as *side conditions*. When writing the rules of the semantics, the following form will be used:

$\frac{\textbf{premises}}{\textbf{conclusion}}$ **side conditions**

Any rule without premises is known as an *axiom*.

The form that each of the premises and the conclusion will take on in the rules is:

$$env \vdash a \xrightarrow{eval} b$$

which should be read, "a evaluates to b in the environment *env* using the relation $\xrightarrow{eval}$". The environment is the static state of the evaluation (i.e. the current time, the state of the signals, etc.). The turnstile ($\vdash$) separates the environment from the operation of evaluation from a into b.

4.2 Information Organisation

The information needed to make the semantics work is really very simple. The way it is presented here was conceived to correspond to the intuition of VHDL users about the structuring of information in VHDL itself. Much use will be made of set notation in what follows.

Time will be represented by the natural numbers, values by Booleans, and signal names by strings. Whenever there is an event on signal x, its name will be included in a set of events (γ). The state of signals (σ) is given as a set of name-value pairs. Future transactions (τ) is a function from time to σ, and will provide the framework for scheduling. The trace of past behaviour (θ), which is necessary for proper interpretation of attributed signals, is also a function on time, but maps it to a γ-σ pair. The environment in which an individual simulation cycle runs (ρ) holds the current time, σ, γ, and θ. The overall simulation environment (μ) augments ρ with an ordered list of times representing the delays associated with the signal assignments of the Femto-VHDL program in question.

$$\text{time} = N \qquad \text{value} = \text{Boolean} \qquad \text{name} = \text{string}$$
$$\gamma = (\text{name})\text{set} \qquad \sigma = (\text{name} \times \text{value})\text{set} \qquad \tau = \text{time} \to \sigma \qquad \theta = \text{time} \to (\gamma \times \sigma)$$
$$\rho = (\text{time} \times \sigma \times \gamma \times \theta) \qquad \mu = ((\text{time})\text{list} \times \rho)$$

The delays are added to the overall simulation environment to allow the simulation loop to move forward to the next point of computation. Recall that the loop should progress to the nearest point of computation where there are transactions to process. The way that these transactions are created is through the execution of signal assignment statements. So by gathering together all the delays associated with the signal assignments of a given program into an ordered list, we provide a way of knowing the only reachable offsets from the current time.

4.2.1 Delay Extraction

Bearing in mind the usefulness of the just mentioned delays, functions are now defined to extract them from Femto-VHDL programs. Before doing so, two functions that ensure the ordering of lists of delays are given. The first function is list recursive, and adds a new delay to the existing list of delays. It guarantees that any such addition preserves the order of the list.

```
add_one_delay dly dlys ≡
  if nil dlys then [dly]
  elseif dly = (hd dlys) then dlys
  elseif dly < (hd dlys) then (dly . dlys)
  else (hd dlys . add_one_delay delay (tl dlys))
```

The next function is also list recursive, and merges together two lists of delays. Use is made of add_one_delay to preserve the ordering. merge_delays is really doing nothing more than a sorted appending operation that deletes duplicate elements.

```
merge_delays dlys' dlys'' ≡
  if nil dlys' then dlys''
  else merge_delays (tl dlys) (add_one_delay (hd dlys') dlys'')
```

As an example, merge_delays [2,5,1][4,5,8] would yield the list [1,2,4,5,8].

Having defined both add_one_delay and merge_delays, the functions that actually extract the delays from Femto-VHDL program texts may be defined. It is assumed that any design hierarchies have been flattened out into a sea of concurrent processes. The first function extracts delays from sequential statements. As anticipated, delays are only to be found on signal assignment statements, and are added to the running list by calls to add_one_delay.

```
get_delays_ss ss l ≡
  if (ss = ss' ; ss'') ∨ (ss = e => ss' | ss'') then get_delays_ss ss' (get_delays_ss ss'' l)
  elseif (ss = s :=inert (e,dly)) ∨ (ss = s :=trans (e,dly)) then add_one_delay dly l
  else l
```

The second function extracts delays from the collection of parallel processes. Any delays that are found in individual processes through the use of get_delays_ss, are merged with those found on any other processes via calls to merge_delays.

get_delays_cs $cs \equiv$
if $cs = (sl : ss)$ then get_delays_ss ss []
else let $(cs' \,||\, cs'') = cs$ in merge_delays (get_delays_cs cs') (get_delays_cs cs'')

An example of the use of this suite of functions may be had by taking some example processes used in Chapter 3,

$$\{x\} : a :=^{\text{inert}} (\text{not } x, 2) \,||\, \{y'\} : b :=^{\text{trans}} (y' \text{ and } y'', 0) \,||\, \{z\} : c :=^{\text{trans}} (z, 0)$$

and running get_delays_cs over them to return the list $[0,2]$.

4.2.2 Well-Formedness

A well-formed σ has two essential parts. The first being that there are no signal-value pairs that have the same signal name as their first element (i.e. signals have unique values). The second criterion is that if an architecture simulates with a particular σ in its environment, then all the signals in the architecture must have an associated value in that σ. The definition of well_formed_σ encapsulates these properties:

well_formed_σ σ *design* $\equiv$
let *sigs* = image fst σ in
card σ = card *sigs* $\wedge$
$\forall sig.(sig \in (\text{get\_sigs\_design } design)) \supset \exists val.\,(sig, val) \in \sigma$

The function uses card to calculate the cardinality of a finite set (i.e. card $\{a,b\} = 2$ and card $\{\} = 0$), and image to extract the set of signal names from the given σ (i.e. image fst $((x,a),(y,b),(y,c)) = \{x,y\}$). The first conjunct simply states that the σ in question must have the same number of elements as the set of signal names associated with that σ. The second conjunct stipulates that any signal that is in the set of signal names extracted from the design, must have a value associated with it in σ.

When dealing with the well-formedness of traces of past behaviour (θ), two constraints must be met. Both will apply at any time rather than at specific offsets associated with a particular program. The reason is twofold. First, it makes little intuitive sense to have a past which is only applicable at certain time points. Second, a 'delayed signal attribute may attempt to sample the past at a time offset which is not one of the signal assignment delays. The semantics, particularly the definition of

the top-level simulation loop, will ensure that traces of past behaviour are constructed to contain intervals rather than time points.

The function well_formed_θ is defined to encapsulate the two requirements on the proper construction of individual θ. These constraints apply, as in well_formed_σ, with respect to the design that a particular θ is associated with. The first of the two constraints is that any signal found as an event in the trace must be one of the signals of the design in question. The second requires that for any time point mentioned, the σ found there must be well-formed.

well_formed_θ θ *design* ≡
$\forall t\ sig.\left(sig \in (\mathsf{fst}\ (\theta\ t)) \supset sig \in (\mathsf{get\_sigs\_design}\ design)\right) \wedge$
$\left(\mathsf{well\_formed\_}\sigma\ (\mathsf{snd}\ (\theta\ t))\ design\right)$

It will be necessary in later reasoning to know about membership in a trace of past activity based on the well-formedness of that history:

Theorem 4.2.2.1:
$\forall \theta\ dsgn.$
well_formed_θ θ *dsgn* ⊃ $\forall t\ s.s \in$ get_sigs_design *dsgn* ⊃ $\exists v.\ (s,v) \in \mathsf{snd}(\theta\ t)$

If one assumes well_formed_θ θ *dsgn*, expand its definition (as well as that of well_formed_σ), reduce the resulting internal let expressions, and take the second conjunct, the required result is obtained after discharging the initial assumption and generalising both θ and *dsgn*.

4.2.3 Equivalence

Equivalence for γ, σ, τ and θ is not a straight equality. Rather it is predicated on the user supplying some signals of interest. The reasoning behind this is perhaps best illustrated by example. Assume that one wished to compare the simulation γ associated with architecture A with that for architecture B. Both A and B have the same external ports, but differ in their internally declared signals. Since any comparison of A with B is only meaningful on externally visible signals, it is essential that the internal signals be factored out of each γ before making the equality test on the two sets. So, the relation $\overset{\gamma}{=}$ is defined in the following way:

$$\gamma' \underset{sigs}{\overset{\gamma}{=}} \gamma'' \equiv \left(\{s : s \in \gamma' \wedge s \in sigs\} = \{s : s \in \gamma'' \wedge s \in sigs\}\right)$$

It is an equivalence relation, and obviously exhibits the required properties of reflexivity, symmetry and transitivity. A similar relationship, may be defined for σ:

$$\sigma' \underset{sigs}{\overset{\sigma}{=}} \sigma'' \equiv \left(\{(s,v) : s \in \sigma' \wedge s \in sigs\} = \{(s,v) : s \in \sigma'' \wedge s \in sigs\}\right)$$

Both τ and θ are a bit more involved when defining equivalence. This is due to the fact that they are functions of time. Therefore, the simple approach taken above must be augmented to check that the sets are the same for all times. For τ, the relation $\overset{\tau}{=}$ may is defined as:

$$\tau' \underset{sigs}{\overset{\tau}{=}} \tau'' \equiv \forall t. \left(\{(s,v) : s \in (\tau'\ t) \wedge s \in sigs\} = \{(s,v) : s \in (\tau''\ t) \wedge s \in sigs\}\right)$$

The definition of a relation on θ is much the same. Note that the definition of $\overset{\theta}{=}$ takes both the trace of γ and the trace of σ into account.

$$\theta' \underset{sigs}{\overset{\theta}{=}} \theta'' \equiv \forall t. \left(\{s : s \in (\mathsf{fst}\ (\theta'\ t)) \wedge s \in sigs\} = \{s : s \in (\mathsf{fst}\ (\theta''\ t)) \wedge s \in sigs\}\right) \wedge$$
$$\left(\{(s,v) : s \in (\mathsf{snd}\ (\theta'\ t)) \wedge s \in sigs\} = \{(s,v) : s \in (\mathsf{snd}\ (\theta''\ t)) \wedge s \in sigs\}\right)$$

Later on in the presentation of the semantics it will become necessary to reason about membership of signal value pairs in equivalent τ's. The following complementary theorems will aid in that reasoning. They merely state that if two τ's are know to be equivalent on a set of ports, then given any signal named in those ports which is in one of τ's the same signal-value pair must appear the other one as well.

Theorem 4.2.3.1:

$$\forall \tau'\ \tau''\ t\ sig\ val\ ports.$$
$$\left(\left(\tau' \underset{ports}{\overset{\tau}{=}} \tau''\right) \wedge (sig \in ports) \wedge (sig,val) \in (\tau'\ t)\right) \supset (sig,val) \in (\tau''\ t)$$

Theorem 4.2.3.2:

$$\forall \tau'\ \tau''\ t\ sig\ val\ ports.$$
$$\left(\left(\tau' \underset{ports}{\overset{\tau}{=}} \tau''\right) \wedge (sig \in ports) \wedge (sig,val) \in (\tau''\ t)\right) \supset (sig,val) \in (\tau'\ t)$$

The definition of $\stackrel{\tau}{=}$ makes it clear that any signal named in *ports* will have the same signal-value pair in both τ' and τ'' at any chosen time. Each of the desired implications is a consequence of this.

4.3 Rules of the Semantics

The rules that define the operational semantics of the Femto-VHDL subset fall under five broad headings – rules for Boolean expressions, rules for sequential statements, rules for concurrent statements (the simulation cycle), rules for initialisation and rules for the top-level simulation loop. The presentation of these rules now follows, beginning with those defining the evaluation of Boolean expressions. Their format is based on the presentation discussed in Section 4.1.2.

4.3.1 Rules for Boolean Expressions

The rules that define the semantics of Boolean expressions are numerous, but are conceptually very simple. They are, in essence, describing an evaluator for Boolean expressions. A key component of that evaluation will be the determination of the value of actual signals – a task accomplished by the function valcalc:

$$\textbf{valcalc}\ sig\ \sigma \equiv \textsf{choice}\ \{val{:}(sig, val) \in \sigma\}$$

The function calculates the value of a signal *sig* in a state σ, by making use of the function choice which selects an arbitrary element from a non-empty set. The use of choice is possible in this context because the set from which the choice is made will always be a singleton set. This is due to the fact that signals have unique values in any σ as long as they are well-formed. For example, valcalc $x\ \{(y,a),(x,b)\} = b$.

In the rules, the symbol $\xrightarrow{bool}$ denotes the evaluation relation for Femto-VHDL Boolean expressions. The type of $\xrightarrow{bool}$ is $bexp \rightarrow Boolean \rightarrow Boolean$, and is given in the upper left-hand corner of the box displaying the rules. This type information will be a recurrent pattern in the presentation of the other rule sets. Axioms are provided for the evaluation of individual signals, the values of the constants true and false, the calculation of the delayed value of a signal and the determination of whether or not there is an event occurring on a signal. The other inference rules define the relationship between Femto-VHDL Boolean operators, and the logical Boolean expressions that they evaluate to. For instance, rule b_6 states that if some Femto-VHDL Boolean expression e evaluates to the logical value x, then the

not of that expression evaluates to the logical negation of x. The symbols T and $\perp$ below stand for "true" and "false" respectively. Furthermore, $\oplus$ is the exclusive or operation.

$bexp \rightarrow Boolean \rightarrow Boolean$

$$\{b_1\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig \xrightarrow{bool} \text{valcalc } sig\ \sigma}$$

$$\{b_2\}\ \frac{}{\rho \vdash \text{true} \xrightarrow{bool} T} \qquad \{b_3\}\ \frac{}{\rho \vdash \text{false} \xrightarrow{bool} \perp}$$

$$\{b_4\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig\text{'delayed}(t) \xrightarrow{bool} \text{valcalc } sig\ (\text{snd}(\theta(now-t)))}$$

$$\{b_5\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig\text{'event} \xrightarrow{bool} sig \in \gamma}$$

$$\{b_6\}\ \frac{\rho \vdash e \xrightarrow{bool} x}{\rho \vdash (\text{not } e) \xrightarrow{bool} \neg x}$$

$$\{b_7\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ and } e'') \xrightarrow{bool} (x \wedge y)} \qquad \{b_8\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ or } e'') \xrightarrow{bool} (x \vee y)}$$

$$\{b_9\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ nand } e'') \xrightarrow{bool} \neg(x \wedge y)} \qquad \{b_{10}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ nor } e'') \xrightarrow{bool} \neg(x \vee y)}$$

$$\{b_{11}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ xor } e'') \xrightarrow{bool} (x \oplus y)} \qquad \{b_{12}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ xnor } e'') \xrightarrow{bool} \neg(x \oplus y)}$$

The only non-trivial rule is perhaps b_4 which defines the calculation of a delayed value for a particular signal *sig*. The result arises from looking at the state of the signal values $now - t$ units ago and extracting the value of the signal in question via valcalc. The state of interest is found by examining the trace of past history represented by θ at the desired time, and extracting the second element of the pair.

4.3.2 Rules for Sequential Statements

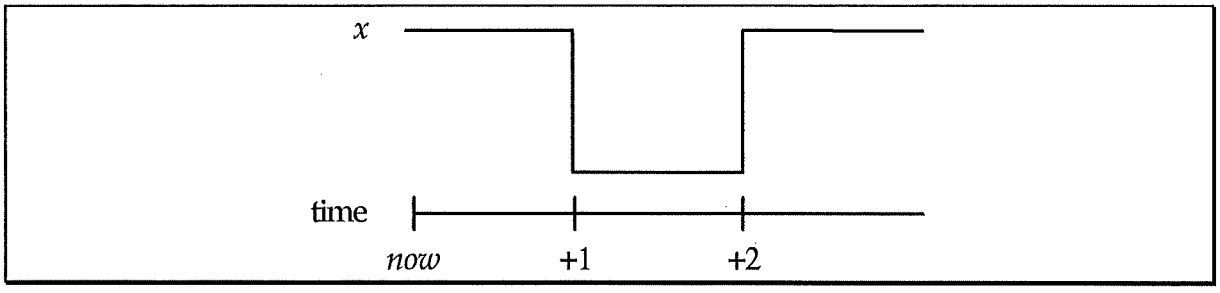
The formalisation of the sequential statements of Femto-VHDL is for the most part as obvious as that for Boolean expressions. The rules, in addition to the other sequential constructs in the subset, encapsulate the statements which drive the simulation forward – inertial and transport delay signal assignment. Whenever a signal assignment statement of either form is encountered, it becomes necessary to "post", or schedule, a transaction to take place at some future point of computation

to effect that assignment. These transactions are in turn used by the overall simulation loop to determine where the next point of computation lies.

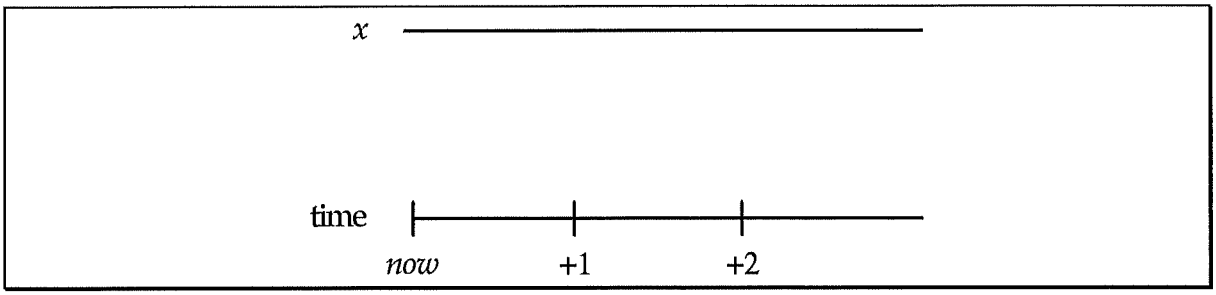
During signal assignment, it is essential that the pre-emptive scheduling required by VHDL take place [21]. This type of scheduling is a particular artifact of the VHDL simulation model that may delete certain previously posted transactions, and is best illustrated by example. Assume that signal x is scheduled to take on a value at time t. Also assume that an attempt is made to post another value for the same signal at time t'. If t' is less than or equal to t, then the first value scheduled at time t (along with any other value previously posted at or after time t') is deleted from the future transactions, and the new transaction for time t' is posted. The inverse situation (t' greater than t) is not part of pre-emptive scheduling, but rather relates to inertial delay assignments in particular.

Transport delay, barring pre-emptive scheduling, is analogous to saying that whatever value gets assigned to a signal eventually appears. Inertial delay makes use of exactly the same mechanism as transport delay, but in a constrained way. In order for a particular transaction to be posted, inertial delay further requires that the inputs to the assignment be stable during the entirety of the assignment interval, and thereby removes any "spikes" on the input.

As an example of the difference between the two kinds of delay, consider the following example. Assume that the signal x has a high value. Furthermore, assume that the signal is scheduled to take on a low value at time $now+1$. If one then attempts to post a high for x at time $now+2$ using transport delay, the waveform for x would look like:



If, however, the same scheduling were to take place with inertial delay, the waveform would take on the following form:



The difference results from the fact that x is not stable during the long assignment interval, and is disqualified from assignment at time $now+1$ because the transaction scheduled there represents a spike in the interval between times now and $now+2$.

Transport delay will expressed by the function trans_post:

trans_post *sig val* τ $t \equiv$
let stripped $t' = (\tau\, t') - \{(x,y):(x,y) \in (\tau\, t') \wedge (x = sig) \wedge (t' \geq t)\}$ in
$\lambda t'$.if $(t' = t)$ then $((sig, val)$ insert (stripped $t')$)
else (stripped t')

The function makes use of insert to insert a new element in to a set (i.e. 1 insert $\{3,4\}$ becomes $\{1,3,4\}$). The current transactions (τ), the signal for which the assignment is to be made (*sig*), the value that the signal is supposed to take on (*val*) and the time at which that assignment is to take place (t) are employed to create a new version of future transactions by returning a function from time to σ. If the time supplied to this function is the one at which the signal assignment is to take place, a set consisting of a "stripped" version of the projected σ with the given signal-value pair inserted into it is returned. Otherwise, all that is returned is the stripped version of the projected state. The stripping process is performed by a function local to trans_post called stripped, which in fact does the pre-emption part of scheduling a new transaction. stripped, when given a time t', will return a version of the starting τ in which all pending transactions on signal *sig* have been removed when t' is greater than or equal to the time at which the current scheduling is to take place.

Inertial delay signal assignment statements (i.e. those with the kind of set-up and hold constraints mentioned earlier) make use of a function that is rather different from trans_post to perform the posting of transactions. The difference lies in the stability check and its consequences on the act of scheduling. With transport delay, a transaction will always be posted. With inertial delay, this is not the case as a non-stable waveform will cause transactions to be unscheduled.

The main concern before doing the assignment is therefore to ensure the stability of the input waveform. To do so, pending transactions during the assignment interval will have to be examined. At the lowest level, a check will have to be made on planned σ to see that the signal value that has been projected in that σ is the same as the one trying to be scheduled.

check σ *sig val* $\equiv$
  let $sigval = \{(x,y):(x,y) \in \sigma \wedge (x = sig)\}$ in
    if $(sigval = \{\})$ then T else $(sigval = \{(sig, val)\})$

The function also flags as acceptable those σ that have no projected value for the given signal.

The above check will have to be made on a specific range in the future transactions. Those that are of interest lie between whatever the current simulation time is and that time plus the delay associated with the assignment statement in question.

is_stable *dly* τ *now sig val* $\equiv$
  if $(dly = 0)$ then T
  else $(\text{check}\ (\tau\ (now + dly))\ sig\ val) \wedge (\text{is\_stable}\ (dly - 1)\ \tau\ now\ sig\ val)$

Signals are assumed to be trivially stable at the current time. For all other times in the delay, one recurses back check'ing each time point.

If there is an instability, it will be necessary to purge any spikes that are found in the assignment interval. The function purge, similar to is_stable, accomplishes the task:

purge *dly* τ *now sig val* $\equiv$
  if $(dly = 0)$ then τ
  else let $\tau' = \lambda t.$if $(t = now + dly)$ then
                $(\tau(now + dly)) - \{(x,y):(x,y) \in \tau(now + dly) \wedge x = sig \wedge y \neq val\}$
              else $\tau\ t$ in
      purge $(dly - 1)\ \tau'$ *now sig val*

Armed with the definitions of is_stable and purge it is now possible to give the function which performs the scheduling of inertial delay assignments. Intuitively, as long as the stability constraint holds, the assignment is nothing more than what was done for transport delay. If the signal is not stable during the interval, then the spikes should be purged from the interval before the assignment can take place.

inert_post *sig val curval* τ *now dly* ≡
  if (is_stable *dly* τ *now sig curval*) then trans_post *sig val* τ (*now* + *dly*)
  else trans_post *sig val* (purge *dly* τ *now sig val curval*) (*now* + *dly*)

The arguments to inert_post should be read as (from left to right) the signal in question, the value being assigned, the current value of the signal, the projected transactions, the current time and the delay associated with the assignment.

The rules defining the evaluation of Femto-VHDL sequential statements are now given. The relation makes use of the one defined for Boolean expressions in many places. The $\langle x, \tau \rangle$ notation is used for grouping together a given statement with the current transactions (e.g. $\langle \mathsf{null}, \tau \rangle$). An axiom defines that a null statement should do nothing to the transactions, passing them along unchanged. Sequencing of Femto-VHDL statements and the evaluation of conditionals proceed in the expected way. It is, however, in defining the evaluation of the two kinds of assignment statements that the driving force of the simulation engine is addressed. The result of performing an assignment statement is the modification of future transactions through inert_post and trans_post.

$ss \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{\mathsf{ss}_1\}\ \frac{}{\rho \vdash \langle \mathsf{null}, \tau \rangle \xrightarrow{seq} \tau}$$

$$\{\mathsf{ss}_2\}\ \frac{\rho \vdash \langle ss', \tau \rangle \xrightarrow{seq} \tau' \quad \rho \vdash \langle ss'', \tau' \rangle \xrightarrow{seq} \tau''}{\rho \vdash \langle ss'; ss'', \tau \rangle \xrightarrow{seq} \tau''}$$

$$\{\mathsf{ss}_3\}\ \frac{\rho \vdash e \xrightarrow{bool} \mathsf{T} \quad \rho \vdash \langle ss', \tau \rangle \xrightarrow{seq} \tau'}{\rho \vdash \langle e \Rightarrow ss' \mid ss'', \tau \rangle \xrightarrow{seq} \tau'}$$

$$\{\mathsf{ss}_4\}\ \frac{\rho \vdash e \xrightarrow{bool} \bot \quad \rho \vdash \langle ss'', \tau \rangle \xrightarrow{seq} \tau'}{\rho \vdash \langle e \Rightarrow ss' \mid ss'', \tau \rangle \xrightarrow{seq} \tau'}$$

$$\{\mathsf{ss}_5\}\ \frac{(t, \sigma, \gamma, \theta) \vdash e \xrightarrow{bool} x}{(t, \sigma, \gamma, \theta) \vdash \left\langle sig :=^{\mathsf{inert}} (e, dly), \tau \right\rangle \xrightarrow{seq} \mathsf{inert\_post}\ sig\ x\ (\mathsf{valcalc}\ sig\ \sigma)\ \tau\ t\ dly}$$

$$\{\mathsf{ss}_6\}\ \frac{(t, \sigma, \gamma, \theta) \vdash e \xrightarrow{bool} x}{(t, \sigma, \gamma, \theta) \vdash \left\langle sig :=^{\mathsf{trans}} (e, dly), \tau \right\rangle \xrightarrow{seq} \mathsf{trans\_post}\ sig\ x\ \tau\ (t + dly)}$$

4.3.3 Rules for Concurrent Statements

The rules that define the evaluation of Femto-VHDL concurrent statements are really those that describe a single iteration of a simulation cycle. As explained in Chapter 1, a simulation cycle is based on the notion of processes running in parallel, and returning a composite view of future behaviour after they have all finished executing. To effect this amalgamation, it should only be necessary to perform a simple set union at all times in two returned τ's on the σ's found there. It now becomes evident why the multiple driver restriction has been placed on Femto-VHDL. If such a union was being done for two particular τ's at some time t, and both σ's found there contained a projected value on the same signal, the consistency constraint of unique values for signals would be violated. In a fuller semantics for the VHDL simulation engine, it would be necessary to replace the set union with a more complex function that took into account user-defined resolution functions.

The kind of behaviour required in joining together transactions could be given by the function Zip with the property that:

$$\forall \tau' \tau'' t.$$
$$\text{let } x = \text{image fst } \{(x,y):(x,y) \in (\tau' t)\} \text{ and } y = \text{image fst } \{(x,y):(x,y) \in (\tau'' t)\} \text{ in}$$
$$(\text{disjoint } x\ y) \Rightarrow ((\tau' \textbf{ Zip } \tau'') \equiv \lambda t.(\tau' t) \cup (\tau'' t))$$

Zip is different from previous definitions that have been made in that it is partially specified. The only statement that has been made is what should occur when the two sets are disjoint. The behaviour is undefined when they are not. One would expect the simulation of a well-formed Femto-VHDL program to ensure that those conditions are met. Recall that the multiple driver restriction ensures that only one process may assign to a particular signal, which should cause the transactions at any particular time to be disjoint during simulation. It is therefore possible to re-cast the operation as a more familiar and simple definition:

$$\tau' \textbf{ zip } \tau'' \equiv \lambda t.(\tau' t) \cup (\tau'' t)$$

In fact, the following obvious theorem about the relationship between Zip and zip holds:

Theorem 4.3.3.1: (Equality of Zip and zip)
$\forall \tau' \tau'' t.$
let x = image fst $\{(x,y):(x,y) \in (\tau' t)\}$ and y = image fst $\{(x,y):(x,y) \in (\tau'' t)\}$ in
(disjoint x y) $\Rightarrow ((\tau'$ Zip $\tau'') = (\tau'$ zip $\tau''))$

Armed with the reassurance provided by this fact, use may be made of the simpler zip in the rules which are to come. This is because the two definitions are interchangeable as long as the pre-conditions on Zip hold.

Some other useful theorems for manipulating expressions containing zip include commutativity, associativity and reduction when one of the arguments is a τ that always points to an empty σ. As zip is based on set union, they mirror properties associated with it.

Theorem 4.3.3.2: $\forall \tau' \tau''. (\tau'$ zip $\tau'') = (\tau''$ zip $\tau')$

Theorem 4.3.3.3: $\forall \tau \tau' \tau''. (\tau$ zip $(\tau'$ zip $\tau'')) = ((\tau$ zip $\tau')$ zip $\tau'')$

Theorem 4.3.3.4: $\left(\forall \tau. \left((\lambda t. \{\})\text{ zip }\tau\right) = \tau\right) \wedge \left(\forall \tau. \left(\tau\text{ zip }(\lambda t. \{\})\right) = \tau\right)$

Given the definition of zip, a first attempt at the rules that define the execution of a simulation cycle may be attempted. The first rule states that if the sensitivity list of a particular process does not intersect with the set of current events, the transactions remain unchanged. The second rule is the inverse of the first. That is to say that if the intersection of sensitivity list and events is non-empty, then the resulting transactions arise from the execution of the sequential statements of the process. The final rule says that if starting from the same set of transactions, two concurrent statements produce their own version of future transactions, then their parallel execution results in the zip'ing together of those transactions.

cs → *transactions* → *transactions* → *Boolean*

$$\{cs_1\}\quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl:ss,\tau\rangle \xrightarrow{cyc} \tau} \quad (sl \cap \gamma) = \{\}$$

$$\{cs_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss,\tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl:ss,\tau\rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{cs_3\}\quad \frac{\rho \vdash \langle cs',\tau\rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'',\tau\rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' || cs'',\tau\rangle \xrightarrow{cyc} \tau' \text{ zip } \tau''}$$

The problem with the above rules is that it is possible for a perfectly well-formed program to generate conflicting transactions, thereby not allowing Theorem 4.3.3.1 to be used to get back to Zip. Take for instance the simple case of a program made up of two processes A and B where the starting τ contains a transaction scheduled for signal x. Assume A does not activate, and returns the original transactions unchanged. Also assume that B does activate, and pre-empts the transaction on x. The result of zip'ing together the old and new transactions is a conflict on signal x. According to our intuitive understanding of a simulation cycle, there should not have been a conflict resulting from the execution of these processes.

The seemingly obvious solution to the problem is to have inactive processes return empty transactions ($\lambda t.\{\}$). While this would take care of most situations, the case of a process activating but not scheduling any new transactions (i.e. passing on the old ones unchanged through null statements) engenders exactly the same problem. Clearly, some augmentation of zip is required.

Intuitively, any original transactions need to be stripped out of the returned τ's before they are zip'ed together. Furthermore, any signals for which there are scheduled transactions in the new τ's need to have any reference to them removed from the original transactions. To that end, the function clean_zip is defined as:

clean_zip *orig* τ' τ'' ≡
  let strip_τ' = $(\tau'\ t) - (orig\ t)$
  and strip_τ'' = $(\tau''\ t) - (orig\ t)$ in
  let τ = strip_τ' zip strip_τ'' in
  let $new\_orig\ t = (orig\ t) - \{(x,y): (x \in (\text{image fst } (\tau\ t))) \wedge ((x,y) \in (orig\ t))\}$ in
    τ zip *new_orig*

A pair of theorems about clean_zip now follow. The first one deals with the simple case of the original transactions being empty ($\lambda t.\{\}$), and shows that such a case degenerates to a simple zip of the remaining τ's.

Theorem 4.3.3.5:
$\forall \tau'\ \tau''.\ (\text{clean\_zip}\ (\lambda t.\{\})\ \tau'\ \tau'') = (\tau'\ \text{zip}\ \tau'')$

The argument is based on the fact that any set s less the empty set is s. Therefore, strip_τ' and strip_τ'' in the definition of clean_zip reduce to τ' and τ'' respectively. As a result, τ becomes a zip of τ' and τ''. It is also known that the empty set less any other set is always the empty set. As a consequence, *new_orig* reduces to

($\lambda t.\{\}$). Since any τ zip'ed with empty transactions is that τ, the desired conclusion follows trivially.

The second theorem expresses the commutativity of two τ's when the original transactions remain unchanged.

Theorem 4.3.3.6:
$\forall orig\ \tau'\ \tau''.$ clean_zip $orig\ \tau'\ \tau''$ = clean_zip $orig\ \tau''\ \tau'$

Once one recalls that zip is a commutative operation, the result follows trivially from the definition of clean_zip.

Armed with the definition of clean_zip, the rules that define a simulation cycle may be given as:

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{cs_1\}\quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl{:}ss,\tau\rangle \xrightarrow{cyc} \lambda t.\{\}} \quad (sl \cap \gamma) = \{\}$$

$$\{cs_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss,\tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl{:}ss,\tau\rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{cs_3\}\quad \frac{\rho \vdash \langle cs',\tau\rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'',\tau\rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' \| cs'',\tau\rangle \xrightarrow{cyc} \tau'\ \text{clean\_zip}\ \tau\ \tau'\ \tau''}$$

Note that inactive processes return empty transactions as in the intermediate solution above. Also, the call to zip in the rule defining the execution of concurrent processes has been replaced by a call to clean_zip. These modifications guarantee that as long as the program is well-formed, there can be no clash of signal names in the transactions resulting from the parallel execution of two processes.

4.3.4 Rules for Initialisation

Initialisation in VHDL is characterised by a more general version of the simulation cycle. All processes are activated irrespective of sensitivity lists, and run once [21]. In terms of rules, it is possible to remove the rule about non-active processes from the rules for concurrent statements, as well as the side condition from the second rule. This gives rise to the much simpler relation $\xrightarrow{init}$.

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{\text{init}_1\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss,\tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl:ss,\tau\rangle \xrightarrow{init} \tau'}$$

$$\{\text{init}_2\}\quad \frac{\rho \vdash \langle cs',\tau\rangle \xrightarrow{init} \tau' \quad \rho \vdash \langle cs'',\tau\rangle \xrightarrow{init} \tau''}{\rho \vdash \langle cs' \| cs'',\tau\rangle \xrightarrow{init} \tau' \ \text{clean\_zip}\ \tau\ \tau'\ \tau''}$$

4.3.5 Rules for the Simulation Loop

A key component of the simulation loop is the ability to move from the current point of computation to the next interesting one. Recall that the first component of the simulation environment μ is an ordered list of times that represent the delays in a given Femto-VHDL program text. So, to determine the next point of computation, it is necessary to run through the list until the first non-empty set of transactions is reached. The function next_time is defined recursively over the structure of finite lists to accomplish this task:

next_time *dlys now* $\tau \equiv$
  if (nil *dlys*) then *now*
  elseif $\neg\left(\left(\tau\ (now + (\text{hd } dlys))\right) = \{\}\right)$ then $(now + (\text{hd } dlys))$
  else next_time (tl *dlys*) *now* τ

Another important part of the simulation loop is the determination of whether or not a particular simulation has quiesced. The job of finding this out is analogous to discovering the next point of computation. If a non-empty set of transactions is found while iterating through the list of delays, the simulation has not quiesced. Furthermore, after exhausting all possible delays it is essential to check the emptiness of the current set of events. Obviously, a non-empty γ also means that the simulation has not quiesced.

quiesce *dlys now* $\tau \equiv$
  if (nil *dlys*) then $(\gamma = \{\})$
  elseif $\left(\left(\tau\ (now + (\text{hd } dlys))\right) \neq \{\}\right)$ then $\perp$
  else quiesce (tl *dlys*) *now* τ

If the simulation has not quiesced, it is necessary to merge the signal changes dictated by the current transactions into the static state of signal values. This is done by first extracting the names of the signals for which a transaction has occurred and

using them as a filter. A new set of signal-value pairs is then created from the new transactions and the old state less those pairs which have a signal name in common with the new pairs.

next_ $\sigma\ \sigma'\ \sigma'' \equiv$
  let *sigs* = image fst σ' in $\{(s,v):(s,v) \in \sigma' \vee ((s,v) \in \sigma'' \wedge s \notin sigs)\}$

For instance, next_ $\sigma\ \{(a,\mathsf{T})\}\ \{(a,\bot),(b,\bot)\}$ reduces to $\{(a,\mathsf{T}),(b,\bot)\}$.

A similar approach is used to generate a new set of events. The function next_ γ creates a set of signal names from those pairs with a common signal name in the new and old σ's whose values differ.

next_ $\gamma\ \sigma'\ \sigma'' \equiv$
  $\{signl:\exists x\ y.\ (signl,x) \in \sigma' \wedge (signl,y) \in \sigma'' \wedge (x \neq y)\}$.

Using the same inputs as the previous example, next_ $\gamma\ \{(a,\mathsf{T})\}\ \{(a,\bot),(b,\bot)\}$ becomes $\{a\}$.

The final auxiliary function required by the simulation loop merges a set of events and a set of signal-value pairs into the running trace of past activity. The function add_to_ θ takes the components to be added, the current θ, the beginning of the past time interval and the endpoint of the past time interval (i.e. the point of computation that has just been entered), and returns a new θ. If the beginning time and the endpoint are the same (meaning that the simulation has quiesced, or a δ-step has been made), then the function returned is predicated on the time input to it being the same as or greater than the beginning time, and allows the just-added information to be accessed. Otherwise, the function returned creates an interval of times for which the new trace information may be observed.

add_to_ $\theta\ \sigma\ \gamma\ \theta$ *start finish* $\equiv$
  λt.if $(start = finish)$ then (if $(t \geq start)$ then (γ,σ) else $(\theta\ t)$)
    else (if $(start \leq t \geq finish)$ then (γ,σ) else $(\theta\ t)$)

There is no cause for concern in the use of $\geq$ to delineate the upper bound of a time interval. If the simulation has quiesced, then the final values of the signals should be expected to hold for any time after the end of the simulation. Conversely, if the simulation has not quiesced, then the non-decreasing nature of simulation time will ensure that the right information is accessible as the nested conditional is built. As

an example, the conditional if $(t \geq finish)$ then X else if $(start \leq t \geq finish)$ then Y else Z is the same as if $(t \geq finish)$ then X else if $(start \leq t < finish)$ then Y else Z

A useful theorem about the reduction of nested calls to add_to_θ with identical time arguments is:

Theorem 4.3.5.3:

$\forall \sigma' \gamma' \sigma'' \gamma'' \tau\ start\ finish.$

add_to_θ $\sigma' \gamma'$ (add_to_θ $\sigma'' \gamma'' \tau\ start\ finish$) $start\ finish =$

add_to_θ $\sigma' \gamma' \tau\ start\ finish$

The result follows from the definition of add_to_θ. The nested calls create similarly nested conditionals where the innermost one is hidden by the outermost one. Obviously, this simplifies down to an identical conditional to that of the stand-alone invocation of add_to_θ.

The simulation loop itself is formulated in two rules. The first deals with quiescence, and ensures that the running θ is updated by the final σ and γ before returning a trace of signal values to the user (i.e. the second component of θ at any given time). The second rule states that if the processes in the Femto-VHDL program produce some transactions based on both the current environment and some starting transactions from which the current ones have been deleted, and that if the simulation loop may be performed on these results to give a final behaviour (using the functions defined above), then the processes yield this same behaviour in the starting environment.

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{sim_1\}\quad \frac{}{(dlys,t,\sigma,\gamma,\theta) \vdash \langle cs,\tau\rangle \xrightarrow{sim} \lambda t'.\,\mathsf{snd}(\mathsf{add\_to\_}\theta\ \sigma\ \gamma\ \theta\ t\ t\ t')}\ \mathsf{quiesce}\ dlys\ t\ \tau\ \gamma$$

$$\{sim_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle cs,\tau''\rangle \xrightarrow{cyc} \tau' \qquad (dlys,t',\sigma',\gamma',\theta') \vdash \langle cs,\tau'\rangle \xrightarrow{sim} beh}{(dlys,t,\sigma,\gamma,\theta) \vdash \langle cs,\tau\rangle \xrightarrow{sim} beh}$$

$\tau'' = (\lambda t'.\,\mathsf{if}\ t' = t\ \mathsf{then}\ \{\}\ \mathsf{else}\ (\tau\ t'))$

$t' = \mathsf{next\_time}\ dlys\ t\ \tau'$

$\sigma' = \mathsf{next\_}\sigma\ (\tau'\ t')\ \sigma$

$\gamma' = \mathsf{next\_}\gamma\ (\tau'\ t')\ \sigma$

$\theta' = \mathsf{add\_to\_}\theta\ \sigma\ \gamma\ \theta\ t\ t'$

Having specified the top-level simulation loop, it is now possible to make a definition characterising a complete simulation:

simulate *design* ≡
  (well_formed_design *design*) ∧
  let *procs* = dflat *design* in
  (let σ = image (λ*sig*. (*sig*, ⊥)) (get_sigs_design *design*) in
   let γ = { } and τ = (λ*t*. { })
   and θ = (λ*t*. if (*t* = 0) then ({ }, σ) else ({ }, { })) in
      (0, σ, γ, θ) ⊢ ⟨*procs*, τ⟩ —*init*→ τ′) ∧
  (let *dlys* = get_delays_cs *procs* and *now* = next_time τ′ in
   let γ′ = next_γ (τ′ *now*) γ and σ′ = next_σ (τ′ *now*) σ
   and θ′ = add_to_θ σ γ θ 0 *now*
   and τ″ = (λ*t*. if (*t* = 0) then { } else (τ′ *t*)) in
      (*dlys*, *now*, σ′, γ′, θ′) ⊢ ⟨*procs*, τ″⟩ —*sim*→ *beh*)

The Language Reference Manual stipulates that all signals must be stable at the beginning of the initialisation process. To that end, the initial events are an empty set. The starting transactions are also empty. Since signal values are always defined in the past [21], the initial trace is set up to ensure that any signal in the program which makes use of the delayed attribute will access the starting value of the signal.

Strictly speaking, the definition just presented is not quite complete. Ideally, one needs to have a facility that permits some signals begin the simulation with a value other than the default of low. The definition of simulate may be changed to reflect the desired functionality by adding a set of signal-value pairs called *uservals* to its parameter list and revising the way in which the original σ gets set up.

simulate *design uservals* ≡
  (well_formed_design *design*) ∧
  let *procs* = dflat *design* in
  (let *default*_σ = image (λ*sig*. (*sig*, ⊥)) (get_sigs_design *design*) in
   let σ = {(*s*, *v*): ((*s*, *v*) ∈ *default*_σ ∧ *s* ∉ (image fst *uservals*)) ∨ (*s*, *v*) ∈ *uservals*} in
      ...

It really does not matter if the user-supplied set of values contains signals other than those in the program being simulated. They will be added to the initial state, and will persist throughout the simulation. But, they will not affect its evaluation as they are never accessed by any of the statements in the program.

4.4 Equivalence

One of the most powerful aspects of a formal semantics is its generality. VHDL is presented as a hardware description language permitting many levels of abstraction in the design process. Fundamental to this concept is the notion that one component of a design may be replaced by another equivalent component without affecting the overall design. In effect, this is an affirmation that the equivalence of two programs is a congruence.

What does it mean for two components to be equivalent? The simple answer is that their behaviours are "the same". For purposes of the work presented here, the simulation of a program must quiesce before it can be reasoned about. Once it has done so, the behaviour resulting from rule sim_1 in the definition of $\xrightarrow{sim}$ may be analysed. This means that the equivalence of two components is a comparison of the behaviours generated by each based on the relation $\overset{\tau}{=}$.

The property in question needs several assumptions to hold before it can be proven. For the most part these center around the way in which certain parts of the simulation environment for the overall design are constructed. For instance, the starting state of the signals must be a union of the σ associated with one of the sub-components and the starting σ for the rest of the program. This overall starting state of the signals must furthermore be $\overset{\sigma}{=}$-equivalent on all the signals of the composite program. A like construction with union is also required of the trace of past behaviour. The behaviours of the two subcomponents used in the replacement are required to be $\overset{\tau}{=}$-equivalent on their own ports. Naturally, these sub-programs must correspond to the same entity and have identical ports. Otherwise, it would be impossible to replace one by the other.

Well-formedness assumptions need only apply to the rest of the design into which the sub-components are being inserted. The reasoning behind this is fairly obvious. The sub-components have already been executed, and have a known behaviour. The program into which the substitution is being made is however not known. But, in order for it to simulate properly it must be well-formed both syntactically and with respect to its trace of past behaviour (which implies that it was well-formed on any simulation σ that was ever used).

Theorem 4.4.1:

let A = arch *ent* $arch_A$ *ports* $decls_A$ cs_A and B = arch *ent* $arch_B$ *ports* $decls_B$ cs_B
and $\sigma_{AC} = \sigma_A \cup \sigma_C$ and $\sigma_{BC} = \sigma_B \cup \sigma_C$
and $\theta_{AC}\ t = \left(\left(\mathsf{fst}\left(\theta_A\ t\right)\right) \cup \left(\mathsf{fst}\left(\theta_C\ t\right)\right), \left(\mathsf{snd}\left(\theta_A\ t\right)\right) \cup \left(\mathsf{snd}\left(\theta_C\ t\right)\right)\right)$
and $\theta_{BC}\ t = \left(\left(\mathsf{fst}\left(\theta_B\ t\right)\right) \cup \left(\mathsf{fst}\left(\theta_C\ t\right)\right), \left(\mathsf{snd}\left(\theta_B\ t\right)\right) \cup \left(\mathsf{snd}\left(\theta_C\ t\right)\right)\right)$ in
let $ports'$ = get_ports A and $ports''$ = get_ports C in
let $prts$ = $ports' \cup ports''$ in

$$\begin{aligned}
&\left(\left(dlys_A, now, \sigma_A, \gamma_A, \theta_A\right) \vdash \left\langle \mathsf{dflat}\ A, \tau_A \right\rangle \xrightarrow{sim} beh_A \wedge \mathsf{quiesce}\ dlys_A\ now\ \tau_A\ \gamma_A \wedge \right.\\
&\left(dlys_B, now, \sigma_B, \gamma_B, \theta_B\right) \vdash \left\langle \mathsf{dflat}\ B, \tau_B \right\rangle \xrightarrow{sim} beh_B \wedge \mathsf{quiesce}\ dlys_B\ now\ \tau_B\ \gamma_B \wedge\\
&\left(dlys_{AC}, now, \sigma_{AC}, \gamma_{AC}, \theta_{AC}\right) \vdash \left\langle \mathsf{dflat}\left(\mathsf{dpar}\ A\ C\right), \tau_{AC} \right\rangle \xrightarrow{sim} beh_{AC} \wedge\\
&\left(dlys_{BC}, now, \sigma_{BC}, \gamma_{BC}, \theta_{BC}\right) \vdash \left\langle \mathsf{dflat}\left(\mathsf{dpar}\ B\ C\right), \tau_{BC} \right\rangle \xrightarrow{sim} beh_{BC} \wedge\\
&\mathsf{quiesce}\ dlys_{AC}\ now\ \tau_{AC}\ \gamma_{AC} \wedge \mathsf{quiesce}\ dlys_{BC}\ now\ \tau_{BC}\ \gamma_{BC} \wedge\\
&\mathsf{well_formed_design}\ C \wedge \mathsf{well_formed_}\theta\ \theta_C\ C \wedge\\
&\left.\left(\sigma_{AC} \overset{\sigma}{=}_{prts} \sigma_{BC}\right) \wedge \left(beh_A \overset{\tau}{=}_{ports'} beh_B\right)\right) \supset \left(beh_{AC} \overset{\tau}{=}_{prts} beh_{BC}\right)
\end{aligned}$$

The gist of the proof is to show that for any time t and for any signal *sig* in the set *prts*, the value associated with that signal is the same in both $\left(beh_{AC}\ t\right)$ and $\left(beh_{BC}\ t\right)$. In doing so, the following cases arise:

1. $sig \in ports'$ and $\left(sig, val\right) \in \left(beh_{AC}\ t\right)$, show $\left(sig, val\right) \in \left(beh_{BC}\ t\right)$
2. $sig \in ports'$ and $\left(sig, val\right) \in \left(beh_{BC}\ t\right)$, show $\left(sig, val\right) \in \left(beh_{AC}\ t\right)$
3. $sig \in ports''$ and $\left(sig, val\right) \in \left(beh_{AC}\ t\right)$, show $\left(sig, val\right) \in \left(beh_{BC}\ t\right)$
4. $sig \in ports''$ and $\left(sig, val\right) \in \left(beh_{BC}\ t\right)$, show $\left(sig, val\right) \in \left(beh_{AC}\ t\right)$

Because of the symmetry of $\overset{\tau}{=}$, this reduces down to only 1 and 3.

Recall that when a simulation quiesces, the resulting behaviour is a function from time to the state component σ of the trace θ used in the simulation environment. Each of the θ's here is a composite of the trace associated with the components of interest (A and B) and the overall design C. So, each of the beh_{XC} expands to $\left(\mathsf{snd}\left(\theta_X\ t\right)\right) \cup \left(\mathsf{snd}\left(\theta_C\ t\right)\right)$ where X is either A or B and the traces have also been augmented by the final state of the signals.

For those signals in $ports'$, Theorems 4.2.3.1 and 4.2.3.2 allow one to conclude the appropriate membership information. For the other signals (those explicitly associated with C in $ports''$) the argument devolves to a case analysis based on time. At time *now* the property holds because the starting state of the signals have been assumed to be equivalent. At all other times, the well-formedness of θ_C with respect to C and Theorem 4.2.2.1 lead to the knowledge that any signal mentioned in C has

a value at any given time point in $\mathsf{snd}(\theta_C\ t)$. The well-formedness of C leads with Theorems 3.3.3.1 and 3.3.2.1 to the fact that *ports''* is a subset of all the signals in C. So any port of C will always have a value in $\mathsf{snd}(\theta_C\ t)$. Furthermore, that value is the same irrespective of whether C is paired with A or with B.

CHAPTER 5

MECHANISING FEMTO-VHDL

The semantics of Femto-VHDL as presented in Chapters 3 and 4 may be embedded in a mechanical proof assistant to provide ways of reasoning about programs. The proof environment that will be used is first presented. The discussion will then turn to how the mechanisation of Femto-VHDL is accomplished in this system.

5.1 HOL

The Cambridge Higher-Order Logic (HOL) system was chosen, because of its generality, as the mechanical proof development system for the embedding Femto-VHDL. Much of what follows is a paraphrase of material found in [19] and [18]. The intention is not to give an in-depth view of the system, but to provide the reader with sufficient understanding to comprehend the material that is to be presented later.

5.1.1 ML

The HOL system is built upon an implementation of the language ML, or "Meta Language". ML is an interactive language. At top-level one can evaluate expressions and perform declarations. A complete description of the syntax and semantics of ML is given in [19].

In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon). The declaration `let` $x = e$ evaluates e and binds the resulting value to x. To bind the variables $x_1 \ldots x_n$ simultaneously to the values $e_1 \ldots e_n$ one can perform either the declaration `let` $x_1 = e_1$ `and...and` $x_n = e_n$ or `let` $x_1,\ldots,x_n = e_1,\ldots,e_n$. These two declarations are equivalent. A declaration d can be made local to the evaluation of an expression e by evaluating the expression d `in` e.

5.1.2 The Logic

This section introduces the logic used by the HOL system. The system supports *higher order logic*, which is a version of predicate calculus with three main extensions:

- Variables can range over functions and predicates.
- The logic is *typed*.
- There is no separate syntactic category of *formulae*.

Much of the material here is taken from [19] and [18], and is presented in an informal way. The interested reader is referred to [19] for a formal set-theoretic semantics of the logic.

It is assumed the reader is familiar with predicate logic. The table below summarises the notation used.

Kind of term	*HOL notation*	*Standard notation*	*Description*
Truth, Falsity	`T, F`	$\mathsf{T}, \bot$	*true, false*
Negation	`~`t	$\neg t$	*not* t
Disjunction	t_1 `\/` t_2	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	t_1 `/\` t_2	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	t_1 `==>` t_2	$t_1 \supset t_2$	t_1 *implies* t_2
Equality	t_1 `=` t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	`!`x `.` t	$\forall x.t$	*for all* x: t
$\exists$-quantification	`?`x `.` t	$\exists x.t$	*for some* x: t
ε-term	`@`x `.` t	$\varepsilon x.t$	*an* x *such that*: t
Conditional	`(`t `=>` t_1 `|` t_2`)`	$(t \Rightarrow t_1, t_2)$	*if* t *then* t_1 *else* t_2

5.1.3 Terms

Terms of the HOL logic are represented in ML by a type called `term`. They are normally input between quotation marks. As an example, the expression `"x /\ y ==> z"` evaluates to a term representing $x \wedge y \supset z$. Terms may be manipulated by various built-in functions to either extract subterms or combine them into larger terms.

Terms are quite similar to ML expressions and this can at first be confusing. Indeed, terms of the logic have types similar to ML expressions. For example, `"(1,2)"` is an ML expression with ML type `term`. The HOL type of this term is `num`×`num`. By contrast, the ML expression `("1","2")` has type `term`×`term`.

Functions have types of the form $\sigma_1 \rightarrow \sigma_2$, where σ_1 and σ_2 are the types of the domain and range of the function, respectively. The types of constants are declared in *theories*. An application $t_1\ t_2$ is badly typed if t_1 is not a function or if it is a function, but t_2 is not in its range. Lambda-terms, or λ-terms, denote functions. The symbol '\' is used as an ASCII approximation to λ. Thus '`\x. t`' should be read as '$\lambda x.\, t$'. For example, "`\x. x+1`" is a term that denotes the function $x \mapsto x+1$.

The HOL quotation parser also accepts `let`-terms superficially similar to those in ML. `let`-terms are actually abbreviations for ordinary terms which are specially supported by the parser and pretty printer. The constant `LET` is defined by the λ-term `(\f x. f x)`, and is used to encode `let`-terms in the logic. The system parser repeatedly applies the transformations:

"`let` $f\ v_1 \ldots v_n = t_1$ `in` t_2"	$\rightarrow$	"`LET(\`$f.t_2$`)(\`$v_1 \ldots v_n.t_1$`)`"
"`let` $(v_1,\ldots,v_n) = t_1$ `in` t_2"	$\rightarrow$	"`LET(\`$(v_1,\ldots,v_n).t_2$`)`t_1"
"`let` $v_1 = t_1$ `and` $\ldots$ `and` $v_n = t_n$ `in` t"	$\rightarrow$	"`LET(...(LET(LET(\`$v_1 \ldots v_n.t$`)`t_1`)`$t_2 \ldots$`)`t_n"

5.1.4 Theories

The result of a session with the HOL system is an object called a *theory*. This object is closely related to what a logician would call a theory, but there are some differences arising from the needs of mechanical proof. A HOL theory, like a logician's theory, contains sets of types, constants, definitions and axioms. In addition, however, a HOL theory contains an explicit list of theorems that have been proved from the axioms and definitions. Logicians normally do not need to distinguish theorems that have actually been proved from those that could be proved, hence they do not normally consider sets of proven theorems as part of a theory; rather, they take the theorems of a theory to be the (often infinite) set of all consequences of the axioms and definitions. Another difference between logicians' theories and HOL theories is that, for logicians, theories are relatively static objects, but in HOL they can be thought of as potentially extendible. For example, the HOL system provides tools for adding to theories and combining theories. A typical interaction with HOL consists in combining some existing theories, making some definitions, proving some theorems and then saving the resulting new theory.

The purpose of the HOL system is to provide tools to enable well-formed theories to be constructed. All the theorems of such theories are logical consequences of the definitions and axioms of the theory. The HOL system ensures

that only well-formed theories can be constructed by allowing theorems to be created by *formal proof* only.

5.1.5 Definitions

In HOL, *definitions* are a special kind of axiom that are guaranteed to be consistent. The commonest (but not only) form of a definition is:

$$f\ x_1 \ldots x_n = t$$

where f is declared to be a new constant satisfying this equation (and t is a term whose free variables are included in the set $x_1 \ldots x_n$). Such definitions cannot be recursive because, for example:

$$f\ x = (f\ x) + 1$$

would imply 0=1 (subtract $f\ x$ from both sides) and is therefore inconsistent.

The use of axioms carries considerable danger in general because it is very easy to assert inconsistent axioms. It is thus safer to use only definitions. At first sight this might appear impossible, but in fact all of ordinary mathematics can be developed from logic by definition alone. A theory containing only definitions is called a *definitional theory*. Many useful definitional theories are built into the HOL system, or available as libraries. Examples include theories of numbers (both natural numbers and integers), sets, bags, finite trees, group theory, properties of fixed points and more. The semantics of Femto-VHDL as embedded in HOL also represents a suite of definitional theories.

The theory of numbers built into HOL is a definitional theory that defines numbers logically. Peano's postulates are proved from the definitions of the type `num` and the constants `0` and `SUC`. It follows from Peano's postulates that certain kinds of recursion equations are equivalent to non-recursive definitions. There is a built-in theory of primitive recursion that supports this, together with tools for automatically transforming recursion equations into definitions.

5.1.6 Proof

For a logician, a formal proof is a sequence, each of whose elements is either an *axiom* or follows from earlier members of the sequence by a *rule of inference*. A theorem is the last element of a proof.

Theorems are represented in HOL by values of type `thm`. The only way to create theorems is by generating a proof. In HOL, this consists in applying ML functions representing *rules of inference* to axioms or previously generated theorems. The sequence of such applications directly corresponds to a logician's proof.

There are five axioms of the HOL logic and eight primitive inference rules. The axioms are bound to ML names. For example, the Law of Excluded Middle is bound to the ML name `BOOL_CASES_AX`, and looks like `|- ∀t. (t = T) ∨ (t = F)`.

Theorems are printed with a preceding turnstile `|-` as illustrated above. Rules of inference are ML functions that return values of type `thm`. An example of a rule of inference is *specialisation* (or ∀-elimination). In standard 'natural deduction' notation this is:

$$\frac{\Gamma \vdash \forall x.\, t}{\Gamma \vdash t[t'/x]}$$

- $t[t'/x]$ denotes the result of substituting t' for free occurrences of x in t, with the restriction that no free variables in t' become bound after substitution.

This rule is represented in ML by a function `SPEC`, which takes as arguments a term `"a"` and a theorem `|- ∀x. t[x]` and returns the theorem `|- t[a]`, the result of substituting a for x in $t\,[x]$.
A proof using the axiom and inference rule could be constructed as follows:

1. $\vdash \forall t.\, t = \mathsf{T} \vee t = \bot$	[Axiom `BOOL_CASES_AX`]
2. $\vdash (1 = 2) = \mathsf{T} \vee (1 = 2) = \bot$	[Specialising line 1 using `SPEC` to '1 = 2']

This session consists of a proof of two steps: using an axiom and applying the rule `SPEC`.

The general form of a theorem is $t_1,\ldots,t_n \vdash t$, where $t_1,\ldots,t_n$ are Boolean terms called the *assumptions* and t is a Boolean term called the *conclusion*. Such a theorem asserts that if its assumptions are true then so is its conclusion. Its truth conditions are thus the same as those for the single term $(t_1 \wedge \ldots \wedge t_n) \supset t$. Theorems with no assumptions are printed out in the form `|-` t.

The five axioms and eight primitive inference rules of the HOL logic are described in detail [19]. Every value of type `thm` in the HOL system can be obtained by repeatedly applying primitive inference rules to axioms. When the HOL system is built, the eight primitive rules of inference are defined and the five axioms are bound to their ML names, all other predefined theorems are proved using rules of inference as the system is made.

A proof in the HOL system is constructed by repeatedly applying inference rules to axioms or to previously proved theorems. Since proofs may consist of millions of steps, it is necessary to provide tools to make proof construction easier for the user. The kind of proof performed in the earlier example is known as *forward* proof, and tends to be used in the development of these tools. The idea of a forward direction arises from the act of inferring new facts from known ones.

A particular kind of proof tool known as a *conversion* is a rule that maps a term to a theorem expressing the equality of that term to some other term. An example is the rule for β-conversion:

$$(\lambda x.t_1)\,t_2 \;\mapsto\; \vdash (\lambda x.t_1)\,t_2 = t_1[t_2/x]$$

Theorems of this sort are used in HOL in a variety of contexts to justify the replacement of particular terms by semantically equivalent terms.

The ML type of conversions is `conv` (or `term -> thm`). For example, `BETA_CONV` is an ML function of type `conv` (i.e. a conversion) that expresses β-conversion in HOL. It produces the appropriate equational theorem on β-redexes and fails elsewhere.

The other kind of proof in HOL is known as *backward* or *goal-directed* proof. Here, the user states the fact that needs to be proven as the main goal, and goes about breaking it up into smaller sub-goals that are easier to prove. The system keeps track of these sub-goals, and uses the reasoning that proved them to construct a proof of the original goal using forward proof.

5.1.7 Libraries and Tools

The HOL system provides a variety of libraries that may be combined to make proof development easier. These libraries are either application-specific proof tools, a collection of theorems encapsulating a particular mathematical domain, or a combination of the two. All libraries may be loaded dynamically into a running copy of the HOL system. For purposes of the material presented in this document, use has been made of the following libraries:

- `arith`: Suite of proof procedures for linear arithmetic [8].
- `ind_defs`: Tool for making inductive definitions [25].
- `string`: Logical types for ASCII character codes and strings [27].
- `sets`: Theory of finite and infinite sets [26].

The system also provides many tools that make the task of proof easier. Most of the ones necessary for our purposes have been described above. The only one not covered has been the type definition tool [28]. The tool allows the user to define recursive types in the logic. Heavy use will be made of it in expressing the syntax of Femto-VHDL, as it will allow the BNF grammar that describes it to be directly embedded in the logic. The result will be that one is able to have Femto-VHDL text represented as HOL terms. Furthermore, one will also be able to define recursive functions over their syntax.

5.2 Femto-VHDL in HOL

The mechanisation of Femto-VHDL in HOL was to a large extent a transcription of the definitions made in Chapters 3 and 4 into the system. Once that task was accomplished, proof procedures in the form of conversions were written to animate the semantics. What follows is a general discussion of the way in which the embedding was realised and the top-level conversions that were devised.

5.2.1 Embedding

The embedding of hardware description languages in HOL has recently been an active area of research [9, 10, 38]. Throughout it all, two particular ways of performing this embedding have emerged:

- Represent the abstract syntax of the language in question by terms, then define within the logic semantic functions that assign meanings to terms.
- Only define semantic operators in the logic and arrange that the user interface parse input from language syntax directly to semantic structures, and also print semantic representations in language syntax.

The first style is known as "deep embedding", the second as "shallow embedding". These labels were first applied to embedded semantics in [9]. A deep embedding approach has been chosen for Femto-VHDL and mirrors the presentation of static and dynamic semantics presented in Chapters 3 and 4.

Syntax

In order to make a deep embedding, the syntax of Femto-VHDL first has to be expressed as a suite of recursive type definitions in the logic [28] leading to a type of

terms called DESIGN. As an example, the following shows the HOL definition of this type:

```
let DESIGN = define_type `DESIGN`
 (`DESIGN = ARCH string string `^
                `((string)set#(string)set#(string)set) DECLS CS `^
        `| DPAR DESIGN DESIGN`)
```

The definition closely resembles the BNF specification of the syntax of designs that was made in Chapter 3. The function which defines the type takes two strings as arguments. The first is the name under which the definition will be stored in the theory. The second is a grammar giving the structure of the desired type. The symbol ^ is the ML string concatenation operator, and pairs of ` delineate those strings. The definition makes use of other types constructed in an identical manner. One of them is for local declarations (DECLS), the other for concurrent statements (CS).

Semantics

The definitions which make up the semantics of Femto-VHDL are just as straightforward as those which embed its syntax. Three different functions were used in this process. The first makes non-recursive definitions, the second makes recursive ones and the third allows inductive ones to be made. Examples of each are given below.

The simplest form of definition may be illustrated by using it to set up the equivalence relation for events ($\overset{\gamma}{=}$). Recall its definition from Chapter 4:

$$\gamma' \underset{sigs}{\overset{\gamma}{=}} \gamma'' \equiv \left(\{s : s \in \gamma' \wedge s \in sigs\} = \{s : s \in \gamma'' \wedge s \in sigs\}\right)$$

When put into HOL, the following definition is made:

```
let EQ_GAMMA = new_definition (`EQ_GAMMA`,
    "EQ_GAMMA (gam':events) (gam'':events) sigs =
       {x|(x IN gam')/\(x IN sigs)}={x|(x IN gam'')/\(x IN sigs)}
```

The defining function takes a pair as its argument. The first element is a string giving the name that the definition will be stored under in the theory. The second is a term expressing the desired functionality. Free (unbound) variables in the term will cause new_definition to fail. The resulting theorem, which is in the same form as the term given to new_definition, may subsequently be used to rewrite

instances of the definition into the actions that are required. All the other simple (i.e. non-recursive or inductive) definitions presented in Chapters 3 and 4 were made in an identical manner.

Recursive definitions require that the function `new_recursive_definition` be used to express them. An example of its use can be seen with the realisation of dflat in HOL. The function was given in Chapter 3 as:

dflat *design* ≡
 if *design* =arch *ent arch ports decls cs* then *cs*
 else let (dpar *d′ d″*) = *design* in (dflat *d′*) || (dflat *d″*)

and may be embedded in HOL by:

```
let DFLAT = new_recursive_definition false DESIGN `DFLAT`
    "(DFLAT (ARCH ent arch ports decls cs) = cs) /\
     (DFLAT (DPAR d' d'') = PAR (DFLAT d') (DFLAT d''))"
```

The arguments to the function are a theorem describing the recursion scheme, a flag stating if the constant being defined (in this case `DFLAT`) is prefix or infix, the name the definition will be stored under in the theory, and a term expressing the desired function as a conjunction of recursion equations establishing its behaviour. The recursion theorem is a by-product of defining the type being recursed over [28]. In this case it is `DESIGN`, which was just defined as a part of the syntax of Femto-VHDL. Like `new_definition`, `new_recursive_definition` returns as a theorem a conjunction that looks exactly like the term used in the definition. Each part of the conjunction may then be used independently to rewrite instances of `DFLAT`.

The rules for the semantics make use of the inductive definition mechanism provided in HOL [25]. The rules were transcribed from those of Chapter 4 into the syntax required by the package. Recall the rules for concurrent statements:

cs → *transactions* → *transactions* → *Boolean*

$$\{\mathsf{cs}_1\}\quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl:ss,\tau\rangle \xrightarrow{cyc} \lambda t.\{\}} \quad (sl \cap \gamma) = \{\}$$

$$\{\mathsf{cs}_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss,\tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl:ss,\tau\rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{\mathsf{cs}_3\}\quad \frac{\rho \vdash \langle cs',\tau\rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'',\tau\rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' \| cs'',\tau\rangle \xrightarrow{cyc} \mathsf{clean\_zip}\ \tau\ \tau'\ \tau''}$$

When given to the defining function, they appear as:

```
let (csrules,csind) =
    let CS="—cyc→:cycenv->CS->trans->trans->bool" in
    new_inductive_definition false `—cyc→_DEF`
    ("^CS ρ Stmnt τ τ'",[])
    [[
      %-------------------------------% "(sl ∩ γ_of_ρ ρ))={}"],
      "^CS ρ (PROCESS sl SS) τ (λt.{})";
     [    "—seq→ ρ SS τ τ'";
      %---------------------------% "~((sl ∩ (γ_of_ρ ρ))={})"],
      "^CS ρ (PROCESS sl SS) τ τ'";
     [    "^CS ρ S0 τ τ'" ; "^CS ρ S1 τ τ''"
      %-----------------------------------------% ],
      "^CS ρ (PAR S0 S1) τ (CLEAN_ZIP τ τ' τ'')"]
```

The function γ_of_ρ extracts the γ portion of the environment ρ. The actual constants used for —cyc→ and —seq→ are `SimCycle` and `SeqStmnt` respectively. The arguments to `new_inductive_definition` are much the same as other similar functions in the HOL system. The first argument is a flag to tell the system whether or not the constant being defined is infix or prefix. The second argument is the name associated with the definition in the theory. The third argument is a 'pattern' that supplies information which is needed because the function may be used to define classes of inductively defined relations, rather than just single instances of these relations [25]. The final argument is a list of term list-term pairs, each element of which describes one rule. The term list contains the hypotheses and side conditions. An empty list is used for axioms. The term is the conclusion of the rule. In HOL, any text between pairs of % is interpreted as a comment. So, each of the lines separating hypotheses from conclusion are really nothing more than comments inserted for readability. The result of making an inductive definition in HOL is a suite of theorems comprising both rewriting rules for each rule of the semantics and a rule induction scheme.

5.2.2 Animation

Why bother with the animation of Femto-VHDL inside HOL at all? First, it reinforces one's confidence that the semantics is actually expressing the intended intuition [13]. The deficiency in understanding which led to the replacement of zip by clean_zip in the rules for concurrent statements was exposed in just this way. Second, it is the only robust way of deriving the behaviour of specific programs so that properties of

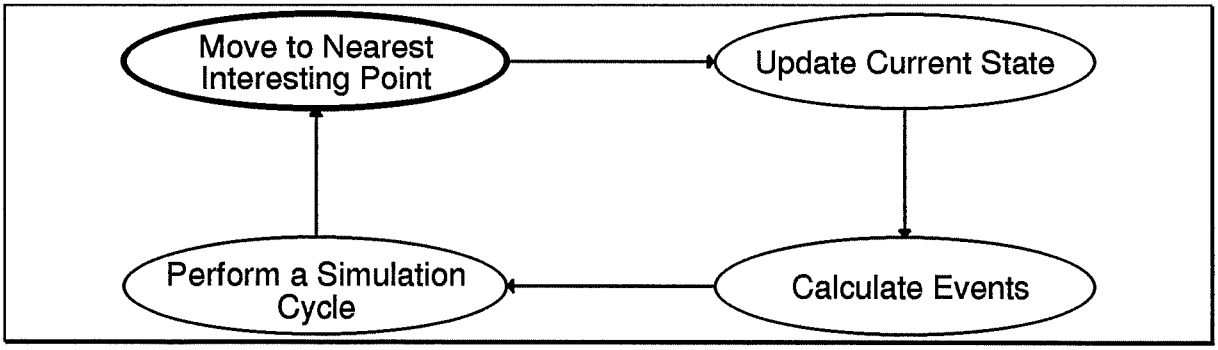
those behaviours or their equivalence to other behaviours may be discerned by proof.

In order to make use of the semantics to derive these behaviours for individual programs, it becomes necessary to write special-purpose conversions to animate the definitions. To a large extent, the process is one of transforming one theorem into another by expanding the definitions that it comprises in an intelligent way. To that end, almost every definition made in the semantics has had a HOL conversion written for it. These conversions are often rewriting engines to replace a definition with its body. Frequently however, it is possible to do some simplification of the result to get a clearer statement (simplification of arithmetic expressions, normalisation, etc.). In these cases the conversions must also do context-dependant reasoning. Most of the time, this leads to faster execution time as it cuts down on the size of the terms being manipulated.

In the course of working with the semantics of individual Femto-VHDL programs, two particular conversions will be frequently used. The first is one that animates one iteration of the simulation loop, and is aptly-enough named `ONCE_AROUND`. The other, `finish_GAMMA`, helps trigger an event when there is insufficient contextual information about signal values. None of the ML code for either of these conversions will be given. Instead the algorithm and rationale behind each will be presented. Examples of their use follow in Chapter 6.

The Conversion ONCE_AROUND

Recall the diagram for the top-level simulation loop:



The purpose of `ONCE_AROUND` is to co-ordinate these steps for individual Femto-VHDL programs by using the rules and definitions of the semantics. Each of the actions above represents a suite of additional proof procedures which animate the related definitions.

The three parts of the cycle, "move to nearest interesting point", "update current state" and "calculate events", are animated by `simp_state_CONV`. The conversion first traverses the list of delays trying to determine the next point of computation. Having ascertained that time, the posted transactions for that time are extracted. These transactions are then compared to the existing signal values to give a new set of signal-value pairs and a new set of events via `NEXT_SIGMA_CONV` and `NEXT_GAMMA_CONV` respectively. Once the new environment has been derived in this fashion, the final part of the diagram ("perform a simulation cycle") is entered into. Here `RUNCYCLE_CONV` is the top level for another set of related conversions which execute the simulation cycle.

Also recall the steps that make up a simulation cycle:

- determine which processes are active based on current events
- run the active processes in parallel
- merge all processes' scheduled transactions into a collective whole

`RUNCYCLE_CONV` first traverses the sea of processes to wake up and run any of them which are sensitive to the just-derived events. This is accomplished by making use of the rules for the simulation cycle as defined in Chapter 4 and embedded in the HOL system via `new_inductive_definition`. Once a process has been activated, the statements inside it are animated by use of `RUNSEQ_CONV` and `RUNBOOL_CONV`, which are responsible for the execution of the rules for sequential statements and Boolean expressions respectively. The behaviour resulting from the activation of any two processes in parallel is resolved from calls to clean_zip to zip by `CLEAN_ZIP_CONV`.

This account of the sequence of conversions hides the fact that simplification of the results arising from many of the steps is also being carried out. A simple example is that calls to clean_zip are reduced to the appropriate sequence of zip's. These calls to zip are also further simplified by the collapsing out of any empty transactions ($\lambda t.\{\}$). Once normalised, the resulting transactions are more concise and therefore make the work of `simp_state_CONV` much easier.

The Conversion `finish_GAMMA`

It is sometimes the case that when `NEXT_GAMMA_CONV` is attempting to ascertain the next set of events, there is not enough information about the state of the signals to ascertain whether or not a new value for particular a signal represents a change from the old one. This situation frequently occurs when the user has supplied

variable values for signals and not expressed their interrelationships. A conditional split must then be introduced into the behaviour of a program to allow for both possibilities (i.e. where one branch assumes an event and the other does not). The user may of course choose at this stage to specialise the variables being used to ensure either one of these possible behaviours. If not, `finish_GAMMA` introduces the appropriate split. Examples of the use of `finish_GAMMA` in context are provided in Chapter 6.

The use of `finish_GAMMA` emphasises one particularly troublesome feature of the animation of not only Femto-VHDL programs, but also of full VHDL. If care is not taken in the specification of initial values of signals, it is quite possible to engender this kind of conditional split for every signal almost every time around the simulation loop. The result is naturally an explosion in the size of the terms being examined. It also puts one in the situation of doing exhaustive search over all possible values for the signals, which is exactly the kind of reasoning that is trying to be avoided.

From the point of view of automation, the case splits engendered by `finish_GAMMA` are one of the reasons behind not attempting to construct a completely automatic tool. Such choice points may be of interest in individual circumstances where one particular course of action needs to be taken. If, however, one was not interested in human intervention, it would be simple to write a naïve conversion to repetitively go around the simulation loop simply by composing `ONCE_AROUND` and `finish_GAMMA`. The problem is, of course, that the simulation may not quiesce. In those situations, the new conversion would loop infinitely.

CHAPTER 6

CASE STUDIES

Several case studies that make use of the formalisation that has been developed are now presented. While the studies do not involve terribly complex programs, they are sufficient for giving a flavour of the kind of reasoning that may be performed. The exposition begins with the simple example of a NAND gate, and progresses through to a more complex device.

As alluded to in Chapter 1, the studies will follow a general pattern:

1. Specify a high-level (i.e. algorithmic) representation of the device.
2. Derive, using the semantics, a general behaviour for the specification.
3. Specify a low-level implementation of the device.
4. Derive a general behaviour for the implementation in the same fashion as 2.
5. Prove that the two behaviours are the same.

The framework for each example will be the same, but different features of the semantics are illustrated by them. The first one concerns itself with demonstrating 0-delay signal assignments. The second deals not only with the equivalence of two components, but also with their provable difference to similar ones. The third example shows how one contends with initialisation and quiescence restrictions. The last one illustrates the use of a state-holding device.

6.1 A NAND Gate

The initial example is not only of use in introducing the way in which one works with the embedded semantics, but is also a demonstration of how 0-delay signal assignments are dealt with. In performing this example, the equivalence of two different versions of a NAND gate will be demonstrated. The first can be considered a "specification" of the behaviour of the gate, and makes use of the VHDL primitive Boolean operator `nand`. The second is an "implementation" that chains together an AND gate and an inverter. The specification will be used to illustrate the

methodology of using the embedded semantics. The implementation will then employ of the same methodology in a similar derivation. The actual HOL session from which this study is extracted appears as Appendix B.

The two designs in question are of course instances of the same entity. The Femto-VHDL for them is:

```
                    entity nandgte is
                      port (A,B : in boolean;
                            C : out boolean);
                    end nandgte;

architecture spec of nandgte is   architecture impl of nandgte is
begin                               signal TMP : boolean;
  process (A,B)                   begin
  begin                             process (A,B)
    c <= A nand B after 1 ns;       begin
  end process;                        TMP <= A and B after 1 ns;
end spec;                           end process;

                                    process (TMP)
                                    begin
                                      C <= not TMP after 0 ns;
                                    end process;
                                  end impl;
```

Inertial delay is used here and in the other studies as it encompasses both the pre-emptive scheduling of transport delay as well as the set-up and hold constraints of inertial delay proper. The architecture for the implementation differs from the specification in that the monolithic NAND operation has been replaced by two parallel processes. A local signal has also been introduced. The first process assigns to this signal, which in turn assigns the result to the output.

In the interests of brevity, the rest of this study as well as those which follow it will revert to the use of abstract syntax for displaying Femto-VHDL text. The translations of the two programs just given are:

$$\mathsf{dflat}\left(\mathsf{arch\ nandgte\ spec}\left(\{A,B\},\{C\},\{\}\right)\left(\mathsf{signal}\ \{\}\right)\left(\{A,B\} : C :=^{\mathsf{inert}} \left(A\ \mathsf{nand}\ B,1\right)\right)\right)$$

$$\mathsf{dflat}\left(\begin{array}{l}\mathsf{arch\ nandgte\ impl}\left(\{A,B\},\{C\},\{\}\right)\left(\mathsf{signal}\ \{TMP\}\right)\\ \qquad\left(\left(\{A,B\} : TMP :=^{\mathsf{inert}} \left(A\ \mathsf{and}\ B,1\right)\right)\|\left(\{TMP\} : C :=^{\mathsf{inert}} \left(\mathsf{not}\ TMP,0\right)\right)\right)\end{array}\right)$$

for the specification and implementation respectively. The above pattern of typefaces will also be followed throughout. Specifically, names of Femto-VHDL

entities, architectures and signals will be given in roman font. Constants relating to the abstract syntax will be given in sans serif. Variables will be shown in *math italic*.

6.1.1 Behaviour of the Specification

Due to the deep embedding approach, it is possible to set up a term in the HOL system that characterises the semantics of the specification in the following way:

```
"let Π = dflat(arch nandgte spec({A,B},{C},{})(signal {})({A,B} : C :=inert (A nand B,1))) in
 let μ = (get_delays_cs Π, now, {(A,a),(B,b),(C,c)}, {A}, λt.({},{})) in
     μ ⊢ ⟨Π, λt.{}⟩ —sim→ beh"
```

Those parts of the term in `typewriter` font represent ML text. Note also the use of `let`-expressions in the term. Despite the fact that use is being made of abstract syntax, the terms of the embedded system can become *very* large. Term size management is therefore essential, and `let`-expressions provide the necessary organisation by minimising the amount of duplication within any given term.

The process of deriving a behaviour for the above term is accomplished by animating it with the rules of the semantics. Essentially, this is a symbolic simulation – "symbolic" in the sense that the values associated with each signal are logical variables, as in the state component of μ above. The conversions described in Chapter 5 are employed in this simulation, and are used to transform the term from a statement about the semantics of the simulation of a particular program into one of its behaviour over time. The result of the process will be a theorem about the simulation and its semantic equality to the behaviour represented by the variable *beh*. The actual derivation of this general behaviour could be a completely automatic process, but is broken down here into its separate components to aid in understanding the individual steps.

Conceptually, the derivation is nothing more than the execution of the program by the simulation engine as it is specified by the rules of the semantics. The first move therefore is to use `ONCE_AROUND` to perform a simulation cycle to determine where the next point of computation lies, move to it and figure out what the new environment there should be. These actions are accomplished by the following simple call to `ONCE_AROUND` where `tm` is bound to the above term:

```
let t1 = ONCE AROUND tm
```

The result is that the starting statement is converted into an equality between it and the simulation behaviour at the next point of computation, $now+1$. The new state of the signals is $\{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a \wedge b))\}$. The problem is that NEXT_GAMMA_CONV could not ascertain what the new events (if any) should be. Clearly the values of A and B have not changed. The question is whether or not the new value of C, $\neg(a \wedge b)$, is a change from its old one, c. There is obviously not enough contextual information about the relationship between these two values to make a judgement.

As a result, the conversion finish_GAMMA must now be used to clear up the ambiguity by inserting the appropriate case split into the derived behaviour. Intuitively, the existence of such an event really should not matter. The signal C is not hooked up to another device, and cannot cause a ripple-through effect. The execution of the program should therefore be the same irrespective of the presence or non-presence of an event on C at time $now+1$. The following application inserts the necessary conditional:

```
let t2 = RIGHT CONV RULE finish GAMMA t1
```

Recall the discussion of conversions in Chapter 5, and the fact that they result in a theorem about the logical equivalence of one term with respect to another. RIGHT_CONV_RULE is a function which applies a conversion to the right hand side of such an equality. In the context of the current example, it is applying finish_GAMMA to the just-derived simulation behaviour at time $now+1$.

Knowledge of the simulation loop leads to the understanding that the introduction of an event on C will eventually filter through the system and not generate any associated activity. Since the signal values are now represented by relationships of the initial inputs, finish_GAMMA will not be needed in subsequent iterations of the simulation loop. The job of arriving at a quiescent statement of behaviour may be done simply by repeated applications of ONCE_AROUND.

A quiescent simulation leaves a theorem that is almost in the desired form. It is a statement of equality between the initial simulation specification and the behaviour of that simulation. The behaviour does however contain two parts that need to be dealt with. The first is the conditional branch introduced by finish_GAMMA. The behaviour after time $now+1$ is the same irrespective of the value of C. This allows the branch to be simplified down to one instance of that behaviour. The other simplification is to replace calls to add_to_θ which had accumulated in the simulation θ and had subsequently become the behaviour of the

program at quiescence with the conditionals that they represent. The result of the simplification is:

$$
\begin{array}{l}
\vdash \forall now\ a\ b\ c\ beh. \\
\quad \texttt{let}\ \Pi = \mathsf{dflat}\left(\mathsf{arch\ nandgte\ spec}\left(\{A,B\},\{C\},\{\}\right)\ldots\left(\{A,B\} : C \overset{\mathsf{inert}}{:=} (A\ \mathsf{nand}\ B,1)\right)\right)\ \texttt{in} \\
\quad \texttt{let}\ \mu = \left(\mathsf{get\_delays\_cs}\ \Pi, now, \{(A,a),(B,b),(C,c)\},\{A\},\lambda t.(\{\},\{\})\right)\ \texttt{in} \\
\qquad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh = \\
\left(\begin{array}{l} beh = \lambda t.\ \texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(A,a),(B,b),(C,\neg(a \wedge b))\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(A,a),(B,b),(C,c)\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right)
\end{array}
$$

It is important to note that the use of add_to_θ has transformed the discrete nature of time used by the simulator into continuous ranges more amenable to human reasoning. Strictly speaking, $t \geq now$ could be replaced by $t = now$ as there are no intervening units of time between *now* and $now+1$. Additionally, the `if-then` construction has replaced the HOL conditional here to aid legibility. The ellipsis (...) indicates where the local signal declarations should go. There are none, and they have been omitted for purposes of conciseness.

6.1.2 Behaviour of the Implementation

A similar sequence of steps may be carried out in the derivation of a behaviour for the implementation of the NAND gate. The first is to again start with a term which describes the architecture in question, along with its starting environment.

$$
\begin{array}{l}
\texttt{"let}\ \Pi = \mathsf{dflat}\left(\begin{array}{l}\mathsf{arch\ nandgte\ impl}\left(\{A,B\},\{C\},\{\}\right)\left(\mathsf{signal}\ \{TMP\}\right) \\ \qquad \left(\left(\{A,B\} : TMP \overset{\mathsf{inert}}{:=} (A\ \mathsf{and}\ B,1)\right) || \left(\{TMP\} : C \overset{\mathsf{inert}}{:=} (\mathsf{not}\ TMP,0)\right)\right)\end{array}\right)\ \texttt{in} \\
\quad \texttt{let}\ \mu = \left(\mathsf{get\_delays\_cs}\ \Pi, now, \{(A,a),(B,b),(C,c),(TMP,tmp)\},\{A\},\lambda t.(\{\},\{\})\right)\ \texttt{in} \\
\qquad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh\text{"}
\end{array}
$$

As seen earlier, the architecture has become a bit more complex. The interest here is to see how the δ-step introduced by the 0-delay signal assignment gets treated during the derivation.

The same steps employed in arriving at a behaviour for the specification are also used in the case of the implementation. The difference is what happens at time

$now+1$. After the process performing the `and` has executed, the state of the signals is $\{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c),(\mathrm{TMP},a\wedge b)\}$. The same ambiguity about events seen in the specification also occurs here. This time, the question is about the value of TMP. The split introduced by `finish_GAMMA` presents the user with two execution paths. The first gives an event on TMP. The second does not. Clearly, an event is needed so that the second process may execute to assign a result to the output. Specialising the variable *tmp* to $\neg(a\wedge b)$ ensures that only the event branch is taken.

More iterations of `ONCE_AROUND` are now required to achieve a quiescent state. Because the signal relationships have been crystallised, there is no further need to use `finish_GAMMA`. All of the subsequent iterations of the simulation loop occur in 0-time. Since these steps are treated in an identical fashion to their unit-delay counterparts, there should be no surprise at the result:

$$
\begin{array}{l}
\vdash \forall now\ a\ b\ c\ beh.\\
\quad \texttt{let}\ \Pi = \mathsf{dflat}\,(\mathsf{arch\ nandgte\ impl}\,(\{\mathrm{A},\mathrm{B}\},\{\mathrm{C}\},\{\})\,(\mathsf{signal}\,\{\mathrm{TMP}\})\ldots)\ \texttt{in}\\
\quad \texttt{let}\ \mu = (\mathsf{get\_delays\_cs}\ \Pi, now, \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c),(\mathrm{TMP},\neg(a\wedge b))\},\ldots)\ \texttt{in}\\
\qquad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh =\\
\quad \left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a\wedge b)),(\mathrm{TMP},(a\wedge b))\}\\ \qquad\quad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c),(\mathrm{TMP},\neg(a\wedge b))\}\\ \qquad\quad \texttt{else}\ \{\} \end{array}\right)
\end{array}
$$

The repeated iterations around the simulation loop at time $now+1$ left nested calls to add_to_θ with identical time arguments in the trace of past behaviour. These were reduced via Theorem 4.3.5.3 when the simulation θ was transformed into the behaviour at quiescence through the use of rule sim_1. The remaining instance of add_to_θ at time $now+1$ resulted in the first branch of the conditional seen in the derived behaviour. The simulator has therefore provided a way of abstracting from multiple δ-steps into the world of macro time simply through its operation.

The signal TMP also has important implications. Recall that the value associated with the signal in the original term has changed from *tmp* to $\neg(a\wedge b)$ as a result of its specification during the derivation. So, without actually knowing the initialisation conditions for the signal, the steps in the derivation allowed them to be determined. This was an instance of human intervention after an application of

finish_GAMMA, and is a reason why this particular kind of verification can never be a completely automatic method.

6.1.3 Equivalence

Having arrived at a behaviour for both versions of the NAND device, it is now possible to determine their equivalence. Recall that the way in which all the equivalence relations were defined in Chapter 4 imposed an obligation on the user to nominate signals of interest. In the present circumstances, the interest is in the behaviour of the device on the ports - in effect abstracting away internally visible signals. Given the definition of equivalence on transactions $\overset{\tau}{=}$, it is easy to prove the following theorem:

$$
\begin{array}{l}
\vdash \forall now\ a\ b\ c\ beh. \\
\left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a \wedge b))\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c)\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right) \\
\qquad\qquad \overset{\tau}{=}_{\{\mathrm{A},\mathrm{B},\mathrm{C}\}} \\
\left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a \wedge b)),(\mathrm{TMP},(a \wedge b))\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c),(\mathrm{TMP},\neg(a \wedge b))\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right)
\end{array}
$$

Essentially, all the definition of $\overset{\tau}{=}$ does is factor out any reference to the signal TMP, leaving two identical functions. The proof in the HOL system is actually performed via case analysis on time.

6.2 DeMorgan Property

The next example demonstrates a VHDL version of the DeMorgan property (i.e. that the negation of the conjunction of two Boolean terms is the same as the disjunction of the negation of each of the terms, and vice versa). The motivation behind the example is to not only show that it is possible to prove that different architectures implement various parts of the property, but also that some of their behaviours provably differ.

The distinction between specification and implementation is slightly blurred in this study. In each part of the analysis, a comparison of two uniprocess designs will

take place. Each of them may be seen as the specification for the other, and vice versa. To that end, no hierarchical distinction will be made between the two programs being examined in each part of the study.

6.2.1 Negation of a Conjunction vs. a Disjunction of Negations

In an identical manner to the previous study, the analysis begins with terms in the HOL system which specify the simulation of the desired programs. For purposes of the current analysis, the following two are used:

$$\texttt{"let } \Pi = \mathsf{dflat}\Big(\mathsf{arch\ dm\ na}\ (\{A,B\},\{C\},\{\})\ldots\Big(\{A,B\} : C \overset{\mathsf{Inert}}{:=} (\mathsf{not}\ (A\ \mathsf{and}\ B),1)\Big)\Big)\ \texttt{in}$$
$$\texttt{let } \mu = \big(\mathsf{get\_delays\_cs}\ \Pi, now, \{(A,a),(B,b),(C,c)\},\{A\},\lambda t.(\{\},\{\})\big)\ \texttt{in}$$
$$\mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh\texttt{"}$$

$$\texttt{"let } \Pi = \mathsf{dflat}\Big(\mathsf{arch\ dm\ on}\ (\{A,B\},\{C\},\{\})\ldots\Big(\{A,B\} : C \overset{\mathsf{Inert}}{:=} (\mathsf{not}\ A\ \mathsf{or}\ \mathsf{not}\ B,1)\Big)\Big)\ \texttt{in}$$
$$\texttt{let } \mu = \big(\mathsf{get\_delays\_cs}\ \Pi, now, \{(A,a),(B,b),(C,c)\},\{A\},\lambda t.(\{\},\{\})\big)\ \texttt{in}$$
$$\mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh\texttt{"}$$

The first term represents the simulation of the negation of a conjunction, while the second deals with a disjunction of negations. As there are no local signals in the program, their declarations have been omitted for conciseness. Note that as in the previous study both environments are the same, and that the signals in each program begin the simulation with arbitrary values.

Behaviour

Each of these terms may be simulated in an identical manner to the specification of the NAND gate. Afterwards, the following theorems giving the behaviour of each program are arise:

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\texttt{let}\ \Pi = \mathsf{dflat}\left(\mathsf{arch\ dm\ on}\ (\{A,B\},\{C\},\{\})\ \ldots\right)\ \texttt{in}\ \ldots =$$
$$\left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(A,a),(B,b),(C,\neg a \vee \neg b)\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(A,a),(B,b),(C,c)\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right)$$

Equivalence

After simulation, the equivalence of the behaviours for the two programs may be ascertained. The only signals of interest are again the input and output ports A, B and C (these also happen to be the only signals in either program). The proof is trivial, and involves comparing the various branches of the conditional statement in each behaviour. The only place where there is not an immediately discernible equality is in the comparison of $\neg(a \wedge b)$ and $(\neg a \vee \neg b)$ at all times greater than or equal to $now+1$. This turns out not surprisingly to be an instance of the DeMorgan property. The actual theorem proved in the HOL system is the following:

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(A,a),(B,b),(C,\neg(a \wedge b))\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(A,a),(B,b),(C,c)\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right)$$
$$\overset{\tau}{=}_{\{A,B,C\}}$$
$$\left(\begin{array}{l} beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(A,a),(B,b),(C,\neg a \vee \neg b)\} \\ \qquad\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(A,a),(B,b),(C,c)\} \\ \qquad\qquad \texttt{else}\ \{\} \end{array}\right)$$

In fact, since the signals are the same in both programs, and no internal declarations are being used, one could replace the use of $\overset{\tau}{=}$ with a simple equality to make a stronger statement.

6.2.2 Negation of a Disjunction vs. a Conjunction of Negations

The second half of the property, namely that the negation of a disjunction is the same as a conjunction of negations follows in an identical manner. The process again begins with the two HOL terms setting up the simulations:

$$\text{"let } \Pi = \text{dflat}\Big(\text{arch dm no } (\{A,B\},\{C\},\{\}) \ldots \Big(\{A,B\} : C \overset{\text{inert}}{:=} (\text{not } (A \text{ or } B),1)\Big)\Big) \text{ in}$$
$$\text{let } \mu = \big(\text{get\_delays\_cs } \Pi, now, \{(A,a),(B,b),(C,c)\},\{A\},\lambda t.(\{\},\{\})\big) \text{ in}$$
$$\mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh\text{"}$$

$$\text{"let } \Pi = \text{dflat}\Big(\text{arch dm an } (\{A,B\},\{C\},\{\}) \ldots \Big(\{A,B\} : C \overset{\text{inert}}{:=} (\text{not A and not B},1)\Big)\Big) \text{ in}$$
$$\text{let } \mu = \big(\text{get\_delays\_cs } \Pi, now, \{(A,a),(B,b),(C,c)\},\{A\},\lambda t.(\{\},\{\})\big) \text{ in}$$
$$\mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh\text{"}$$

Behaviour

Symbolic simulation of these terms leads to the following two theorems in the HOL system. The results are much the same as in Section 6.2.1.

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\text{let } \Pi = \text{dflat}\big(\text{arch dm no } (\{A,B\},\{C\},\{\}) \ldots\big) \text{ in } \ldots =$$
$$\left(beh = \lambda t.\text{if } (t \geq now+1) \text{ then } \{(A,a),(B,b),(C,\neg(a \vee b))\}\right.$$
$$\text{elseif } (t \geq now) \text{ then } \{(A,a),(B,b),(C,c)\}$$
$$\left.\text{else } \{\}\right)$$

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\text{let } \Pi = \text{dflat}\big(\text{arch dm an } (\{A,B\},\{C\},\{\}) \ldots\big) \text{ in } \ldots =$$
$$\left(beh = \lambda t.\text{if } (t \geq now+1) \text{ then } \{(A,a),(B,b),(C,\neg a \wedge \neg b)\}\right.$$
$$\text{elseif } (t \geq now) \text{ then } \{(A,a),(B,b),(C,c)\}$$
$$\left.\text{else } \{\}\right)$$

The difference between these simulations and the earlier ones is in the value of signal C for all times greater than or equal to $now+1$.

Equivalence

As in the previous version, the simulations may be shown to be $\overset{\tau}{=}$-equivalent:

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\left(beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a \vee b))\}\right.$$
$$\left.\texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c)\}\right.$$
$$\left.\texttt{else}\ \{\}\right)$$
$$\overset{\tau}{=}_{\{\mathrm{A},\mathrm{B},\mathrm{C}\}}$$
$$\left(beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg a \wedge \neg b)\}\right.$$
$$\left.\texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c)\}\right.$$
$$\left.\texttt{else}\ \{\}\right)$$

6.2.3 Non-equivalence

It is also possible to show that the behaviours derived in Section 6.2.1 are *not* equivalent to those just given in Section 6.2.2. In each case the place where differences arise is in the value of signal C at times greater than or equal to $now+1$. As an example, consider the comparison of the negation of a conjunction and the conjunction of negations. If the derived behaviours for each are examined, the following theorem about their non-equivalence may be proved ($\overset{\tau}{\neq}$ should be understood to mean "not equivalent according to the relation $\overset{\tau}{=}$").

$$\vdash \forall now\ a\ b\ c\ beh.$$
$$\left(beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg(a \wedge b))\}\right.$$
$$\left.\texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c)\}\right.$$
$$\left.\texttt{else}\ \{\}\right)$$
$$\overset{\tau}{\neq}_{\{\mathrm{A},\mathrm{B},\mathrm{C}\}}$$
$$\left(beh = \lambda t.\texttt{if}\ (t \geq now+1)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},\neg a \wedge \neg b)\}\right.$$
$$\left.\texttt{elseif}\ (t \geq now)\ \texttt{then}\ \{(\mathrm{A},a),(\mathrm{B},b),(\mathrm{C},c)\}\right.$$
$$\left.\texttt{else}\ \{\}\right)$$

Similar results may be obtained for any permutation of comparisons of the behaviours in Sections 6.2.1 and 6.2.2. This correspondence to the expected behaviour of the DeMorgan property further increases one's confidence in the simulation semantics that Femto-VHDL has been given and the way that it is implemented in the HOL system.

6.3 Parity Checker

The above examples were sufficient to introduce basic concepts and techniques. A larger example is however required to demonstrate how concrete initialisation and quiescence restrictions are dealt with, as well as some subtleties of the semantics. A standard (albeit somewhat overworked) example from the HOL literature is a parity checker. The device was introduced in [18]. 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 first requirement gives rise to a change from the kind of analysis that was performed in the previous examples. With the introduction of a specific kind of behaviour at time 0, one is forced to examine not only the general behaviour of the device, but also what it does when initialised.

6.3.1 Specification

The derivation begins, as in the earlier examples, with a program specifying the behaviour of the device in terms of a high-level VHDL design. The obvious Femto-VHDL program would be:

$$\text{arch parity high } (\{\text{inp}\},\{\},\{\text{outp}\})\,(\text{signal}\,\{\})$$

$$\{\text{inp}\}: \left(\begin{array}{l} (now = 0) \Rightarrow \text{outp} :\overset{\text{inert}}{=} (\text{true}, 0) \mid \\ \text{inp} \Rightarrow \text{outp} :\overset{\text{inert}}{=} (\text{not outp'delayed}(1), 0) \mid \\ \text{null} \end{array} \right)$$

In another change from previous examples, the ports include one of VHDL direction `inout`. This is because the signal in question, outp, is both read from and written to as part of a feedback loop. The conditional statement may be read directly from the informal specification (i.e. make sure the output is high at time zero as well as when an even number of high inputs have been counted). The inclusion of 1 as the argument to 'delayed assumes that the input signal being sampled has a period of 1 unit.

The above architecture represents an idealised version of what is desired. In Femto-VHDL, the user is not able to access the global clock (which is possible in full VHDL). The version that will be used in the case study makes use of the initialisation sequence of a simulation to accomplish the same task. The monolithic process above must now become two parallel processes.

```
arch parity high ({inp},{},{outp}) (signal {ini})
    {ini} : ( ini :=inert (true,0) )  ||

    {inp,ini} : ( (ini'event) => outp :=inert (true,0) |
                  (inp and not (ini'event)) => outp :=inert (not outp'delayed(1),0) |
                  null )
```

Here the signal ini is used as a trigger to assign outp the value true at time 0. Recall the discussion of the initialisation process in Chapter 4 at the end of Section 4.3.5. Initialisation involves the waking up of all the processes in the program and running them once before handing control over to the main simulation loop. During initialisation of the current program, there are by definition no events on any of the signals, and both inp and ini have a default low value. Therefore, the only way that ini can have an event is as a result of the initialisation sequence waking up the first process and causing a high value to be immediately scheduled for it. The conditional in the second process will then assign a high value to outp one δ-step later once the simulation loop has been entered. The second process does nothing when initially activated due to the values of the signals all being low and the absence of any events. After the initialisation sequence is over ini is no longer a factor, and only the second process can ever execute.

Initialisation

In order to obtain the initialisation behaviour, the above program is given to the function simulate with an empty set of initial values. In doing so, it becomes necessary to perform two kinds of simulation. The first is different from previous examples in that it involves the initialisation sequence. The second is more familiar, and is about the execution of the simulation loop. The difference is that previous simulations began with empty future transactions. Here the results of initialisation will be used as the starting point.

The initialisation part is animated by RUNINIT_CONV, whose behaviour is similar to that of RUNCYCLE_CONV described in Chapter 5. It merely executes the rules of the initialisation sequence from Chapter 4, Section 4.3.4. The result of initialisation is the derived transaction:

```
inert_post ini ⊥ (λt. {}) 0 0
```

When used as input to the simulation loop, this transaction filters through the program in a δ-step to give the following HOL theorem (Π denotes the program):

```
|– simulate Π {} = ∀beh. beh = λt. if (t ≥ 0) then {(outp,T),(inp,⊥),(ini,T)} else {}
```

which is really:

```
|– simulate Π {} = ∀beh. beh = λt. {(outp,T),(inp,⊥),(ini,T)}
```

and shows that the program behaves in the expected way at time 0.

General Behaviour

The process of deriving a general behaviour for the specification is begun in an analogous manner to the studies in Sections 6.1 and 6.2. The term used in the HOL system to start the symbolic simulation is:

```
"let Π = dflat (arch parity high ({inp},{},{outp}) (signal {ini}) ...)
 and σ = {(inp,inp),(outp,outp),(ini,T)} in
 let μ = (get_delays_cs Π,now,σ,{inp},λt.if (t ≤ now − 1) then ({},σ) else {}) in
     μ ⊢ ⟨Π,λt.{}⟩ —sim→ beh"
```

The ellipsis shows where the concurrent statements in the program should appear. The simulation environment has been augmented by a trace θ giving the past values of the signals. It ensures that the 'delayed signal attribute used in the program works properly. The events in this θ have been left empty, but a set of signal-value pairs has been given and tied to the initial values of the signals. The signal ini has been given a high value as it is assumed that the simulation behaviour of interest occurs after the completion of the initialisation phase. The environment μ is also related to the initialisation behaviour. If *now* was 1, *inp* was low and *outp* was high, the

starting state of the signals would correspond to the just-derived initialisation behaviour.

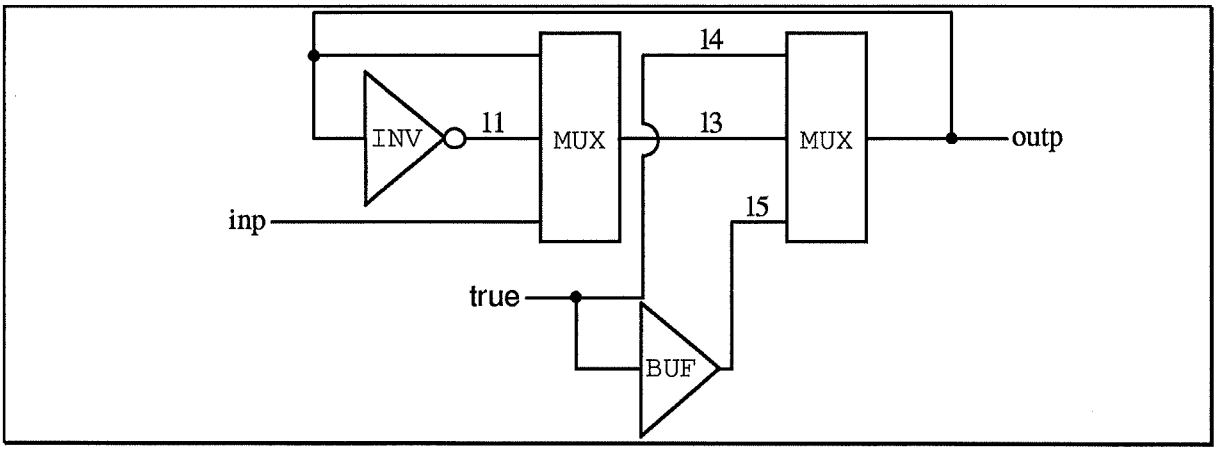
The computation of the general behaviour proceeds in a like manner to the earlier examples. At the end of the process, the following HOL theorem emerges:

$$
\begin{array}{l}
\vdash \forall now\ inp\ outp\ beh. \\
\quad \texttt{let}\ \Pi = \mathsf{dflat}\left(\mathsf{arch}\ \mathsf{parity}\ \mathsf{high}\ (\{\mathrm{inp}\},\{\},\{\mathrm{outp}\})\ (\mathsf{signal}\ \{\mathrm{ini}\})\ldots\right) \\
\quad \texttt{and}\ \sigma = \{(\mathrm{inp},inp),(\mathrm{outp},outp),(\mathrm{ini},\mathsf{T})\}\ \texttt{in} \\
\quad \texttt{let}\ \mu = \ldots\ \texttt{in} \\
\qquad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh = \\
\left(beh = \lambda t. \begin{pmatrix} \texttt{if}\ (inp \wedge (t \geq now))\ \texttt{then}\ \{(\mathrm{inp},inp),(\mathrm{outp},\neg outp),(\mathrm{ini},\mathsf{T})\} \\ \texttt{else}\ \{(\mathrm{inp},inp),(\mathrm{outp},outp),(\mathrm{ini},\mathsf{T})\} \end{pmatrix} \right)
\end{array}
$$

The result is again as expected. The output immediately inverts when the input has a high value. Otherwise, the signal values do not change.

6.3.2 Implementation

The implementation of the device is made up of four components. It includes delay elements, as well as an external feedback loop. The original design was given in [18], and has been slightly simplified here for clarity. The schematic for the device is:



The delay through each multiplexor component is zero, while that through both the buffer and the inverter is 1 unit. The signal 14 is initialised to a high value, and all other signals start low. This ensures that the device will output a high value between power-up and the first unit of simulation time, thus meeting the original specification through the second mux. After time 1, this mux becomes a pass-

through, and the work is done by the first mux. When the input inp is high, the inverted value of the output (delayed by 1 unit) is selected. Otherwise, the output remains unchanged.

In the translation of these components into VHDL, it must be remembered that the "D" in VHDL stands for "description". One is modelling a physical device, not providing an *actualité*, and should therefore beware of situations where simulation may give rise to unexpected behaviour in an apparently obvious situation. The problem with the parity checker is that unless the first multiplexor is equipped with an edge-triggered selector, it is possible to leave the input high and create a unit-delay oscillator. As a result, one must either make sure that the input to the parity checker finishes with a low value, or build into it the edge-triggered behaviour just alluded to. This is because an oscillator, of course, never quieseces, and the basis for reasoning here is constrained to only those devices which quiesce. The realisation of the above schematic must therefore take this potential for oscillation into account.

The components used in the parity checker are not at all complicated. The simplest of them is the buffer:

$$\text{arch simple regstr } (\{inp\},\{\},\{outp\})\,(\text{signal}\,\{\})\left(\{inp\} : outp :\overset{\text{inert}}{=} (inp,1)\right)$$

Note that the signal names are variables. This is because the component has not been given an instantiation in a particular design. The functionality is as expected – the output is assigned the value of the input delayed by one unit. The inverter does much the same thing, with the exception that the delayed signal is inverted before the assignment takes place:

$$\text{arch simple inv } (\{inp\},\{\},\{outp\})\,(\text{signal}\,\{\})\left(\{inp\} : outp :\overset{\text{inert}}{=} (\text{not } inp,1)\right)$$

A multiplexor might reasonably be expected to take on the following form:

$$\text{arch simple mux } (\{sel,in_1,in_2\},\{\},\{outp\})\,(\text{signal}\,\{\})$$
$$\left(\{sel,in_1,in_2\} : \left((sel) => outp :\overset{\text{inert}}{=} (in_1,0) \mid outp :\overset{\text{inert}}{=} (in_2,0)\right)\right)$$

This is perfectly suitable for the second mux in the schematic (i.e. the one that eventually becomes a pass-through device). It is, however, unacceptable if one is not particularly concerned with the values being used as input to the parity checker. As mentioned above, it is entirely possible to create an oscillator with this particular

form of mux. The version that will be used in the design is actually a form of clocked register:

$$\text{arch clkd reg}\left(\{sel, in_1, in_2\},\{\},\{outp\}\right)(\text{signal}\{\})\left(\{sel\}:\left((sel) => outp :=^{\text{Inert}} (in_1, 0) \mid outp :=^{\text{Inert}} (in_2, 0)\right)\right)$$

The component ensures that in_1 is only chosen when a rising edge is detected on the selector. This is not a true multiplexor, and could not be used in place of the previous version as the second mux in the system.

The whole design may be put together in the following fashion. The ellipsis again indicates where the concurrent statements associated with each architecture should go.

$$\begin{array}{l}\text{arch impl parity}\left(\{\text{inp}\},\{\},\{\text{outp}\}\right)(\text{signal}\{\}) \\ \quad \text{dflat}\left(\text{dpar}\left(\text{dpar}\left(\begin{array}{l}\text{arch simple regstr}\left(\{14\},\{\},\{15\}\right)(\text{signal}\{\})\ldots \\ \text{arch simple inv}\left(\{\text{outp}\},\{\},\{11\}\right)(\text{signal}\{\})\ldots\end{array}\right)\right.\right. \\ \qquad \left.\left.\text{dpar}\left(\begin{array}{l}\text{arch simple mux}\left(\{\text{inp},11,\text{outp}\},\{\},\{13\}\right)(\text{signal}\{\})\ldots \\ \text{arch clkd reg}\left(\{15,13,14\},\{\},\{\text{outp}\}\right)(\text{signal}\{\})\ldots\end{array}\right)\right)\right)\end{array}$$

If one is not concerned about the oscillation problem. The instantiation of the components would appear as:

$$\begin{array}{l}\text{arch impl parity}\left(\{\text{inp}\},\{\},\{\text{outp}\}\right)(\text{signal}\{\}) \\ \quad \text{dflat}\left(\text{dpar}\left(\text{dpar}\left(\begin{array}{l}\text{arch simple regstr}\left(\{14\},\{\},\{15\}\right)(\text{signal}\{\})\ldots \\ \text{arch simple inv}\left(\{\text{outp}\},\{\},\{11\}\right)(\text{signal}\{\})\ldots\end{array}\right)\right.\right. \\ \qquad \left.\left.\text{dpar}\left(\begin{array}{l}\text{arch simple mux}\left(\{\text{inp},11,\text{outp}\},\{\},\{13\}\right)(\text{signal}\{\})\ldots \\ \text{arch simple mux}\left(\{15,13,14\},\{\},\{\text{outp}\}\right)(\text{signal}\{\})\ldots\end{array}\right)\right)\right)\end{array}$$

Again, there are two distinct phases of simulation that must be undertaken with the implementation-level version of the parity checker – initial and general behaviour. The complimentary correspondence between these two forms of behaviour that was illustrated when working with the specification must also be shown here.

Initialisation

The initialisation phase of the analysis of the implementation may now proceed. The procedure is much the same as with the specification of the device. Recall that the term representing the architecture was passed to the function simulate. The same is true here, with the exception that a singleton set containing the initial value of the signal 14 (high) is supplied as the second argument.

Once the initialisation conversion has been executed, the resulting projected behaviour (τ) is as follows. The calls to inert_post seen in the analysis of the specification have been expanded and simplified for readability:

$$\lambda t.\,\texttt{if}\ (t = 0)\ \texttt{then}\ \{(\text{outp}, \mathsf{T}), (13, \bot)\}$$
$$\quad \texttt{elseif}\ (t = 1)\ \texttt{then}\ \{(11, \mathsf{T}), (15, \mathsf{T})\}$$
$$\quad \texttt{else}\ \{\}$$

The rest of the simulation may now be completed in an analogous manner to that of the specification. The resulting theorem in the HOL system appears as:

$$\vdash \text{simulate}\ \Pi\ \{(14, \mathsf{T})\} =$$
$$\quad \forall beh.\ beh = \lambda t.\,\texttt{if}\ (t \geq 1)\ \texttt{then}\ \{(\text{inp}, \bot), (\text{outp}, \mathsf{T}), (11, \bot), (13, \mathsf{T}), (14, \mathsf{T}), (15, \mathsf{T})\}$$
$$\qquad\qquad \texttt{elseif}\ (t \geq 0)\ \texttt{then}\ \{(\text{inp}, \bot), (\text{outp}, \mathsf{T}), (11, \bot), (13, \mathsf{T}), (14, \mathsf{T}), (15, \bot)\}$$
$$\qquad\qquad \texttt{else}\ \{\}$$

An analysis of this result shows that it meets the behavioural constraints for time 0 set down in the natural language specification. The output at time 0 is high – a value which is maintained throughout. Furthermore, no reference is made to any intervening δ-steps. One should also note that the value of the selector for the second multiplexor (15) changes from low to high after 1 unit of time has passed.

General Behaviour

Having derived the initialisation behaviour, one must now come up with an overall understanding of behaviour for the implementation. This presages a return to an environment that is replete with variables rather than concrete values. The starting term in the HOL system appears as:

"let Π = dflat (arch impl parity ({inp},{outp},{}) (signal {}) ...)
and σ = {(inp, *inp*),(outp, *outp*),(11, ¬*outp*),(13, *outp*),(14, T),(15, T)} in
let μ = (get_delays_cs Π, *now*, σ, {inp}, $\lambda t.(\{\},\{\})$) in
$\mu \vdash \langle \Pi, \lambda t.\{\} \rangle \xrightarrow{sim} beh$"

Why were these particular initial values for the signals chosen? Clearly there need to be two independent values of interest – the input signal value and the output signal value. If this course is chosen, then 11 will be the inverse of whatever the output is. Since 13 is being directly piped into the output, they both have the same value. Furthermore, it is assumed that time *now* occurs after initialisation has been completed. Both 14 and 15 therefore hold high values. One also notices that if the concrete value of outp was high and that of inp was low, the starting state of the signals corresponds to the initialisation behaviour at times greater than or equal to 1. The trace of past activity has been left empty as there are no instances of the 'delayed attribute that need to be catered for.

The derivation of the behaviour proceeds in a similar manner to earlier examples. The result is a bit more complex. The behaviour is however easily discernible from the nested conditionals.

$\vdash \forall now\ inp\ outp\ beh.$
let ... in $\mu \vdash \langle \Pi, \lambda t.\{\} \rangle \xrightarrow{sim} beh =$
$beh =$
$\lambda t.$ if *inp* then
if $(t \geq now + 1)$ then
{(inp, *inp*),(outp, ¬*outp*),(11, *outp*),(13, ¬*outp*),(14, T),(15, T)}
elseif $(t \geq now)$ then
{(inp, *inp*),(outp, ¬*outp*),(11, ¬*outp*),(13, ¬*outp*),(14, T),(15, T)}
else {(inp, *inp*),(outp, *outp*),(11, ¬*outp*),(13, *outp*),(14, T),(15, T)}
else {(inp, *inp*),(outp, *outp*),(11, ¬*outp*),(13, *outp*),(14, T),(15, T)}

6.3.3 Equivalence

Two different equivalence checks will have to be made – one for initialisation, the other for subsequent iterations of the simulation loop. If one takes the initialisation phase first, it is necessary to recall both the behaviour of the specification:

$\lambda t.$ {(inp, ⊥),(outp, T),(ini, T)}

as well as that of the implementation:

$$\begin{array}{l}\lambda t.\,\texttt{if}\ (t \geq 1)\ \texttt{then}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \quad \texttt{elseif}\ (t \geq 0)\ \texttt{then}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\bot)\}\\ \quad \texttt{else}\ \{\}\end{array}$$

at initialisation. The behaviour of the implementation may further be reduced to:

$$\begin{array}{l}\lambda t.\,\texttt{if}\ (t \geq 1)\ \texttt{then}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \quad \texttt{else}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\bot)\}\end{array}$$

for the simple reason that all the possibilities have been covered by the first two branches of the conditional. The equivalence follows as a HOL theorem from the definition of $\overset{\tau}{=}$ on the signals inp and outp after a case analysis of the behaviour of the implementation.

$$\begin{array}{l}\vdash \lambda t.\{(\text{inp},\bot),(\text{outp},\mathsf{T})\}\\ \qquad\qquad \overset{\tau}{=}\\ \qquad\qquad {\scriptstyle\{\text{inp},\text{outp}\}}\\ \quad \lambda t.\,\texttt{if}\ (t \geq 1)\ \texttt{then}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \qquad \texttt{else}\ \{(\text{inp},\bot),(\text{outp},\mathsf{T}),(11,\bot),(13,\mathsf{T}),(14,\mathsf{T}),(15,\bot)\}\end{array}$$

The second check that needs to be made is for subsequent iterations of the simulation loop. The behaviour of the specification in this instance is:

$$\begin{array}{l}\lambda t.\,\texttt{if}\ (\mathit{inp} \wedge (t \geq \mathit{now}))\ \texttt{then}\ \{(\text{inp},\mathit{inp}),(\text{outp},\neg\mathit{outp}),(\text{ini},\mathsf{T})\}\\ \quad \texttt{else}\ \{(\text{inp},\mathit{inp}),(\text{outp},\mathit{outp}),(\text{ini},\mathsf{T})\}\end{array}$$

while that derived for the implementation was:

$$\begin{array}{l}\lambda t.\,\texttt{if}\ \mathit{inp}\ \texttt{then}\\ \qquad \texttt{if}\ (t \geq \mathit{now}+1)\ \texttt{then}\\ \qquad\quad \{(\text{inp},\mathit{inp}),(\text{outp},\neg\mathit{outp}),(11,\mathit{outp}),(13,\neg\mathit{outp}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \qquad \texttt{elseif}\ (t \geq \mathit{now})\ \texttt{then}\\ \qquad\quad \{(\text{inp},\mathit{inp}),(\text{outp},\neg\mathit{outp}),(11,\neg\mathit{outp}),(13,\neg\mathit{outp}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \qquad \texttt{else}\ \{(\text{inp},\mathit{inp}),(\text{outp},\mathit{outp}),(11,\neg\mathit{outp}),(13,\mathit{outp}),(14,\mathsf{T}),(15,\mathsf{T})\}\\ \quad \texttt{else}\ \{(\text{inp},\mathit{inp}),(\text{outp},\mathit{outp}),(11,\neg\mathit{outp}),(13,\mathit{outp}),(14,\mathsf{T}),(15,\mathsf{T})\}\end{array}$$

If one abstracts away the internal signals and re-interprets time intervals, the following function on time is really what is being dealt with during the equivalence check:

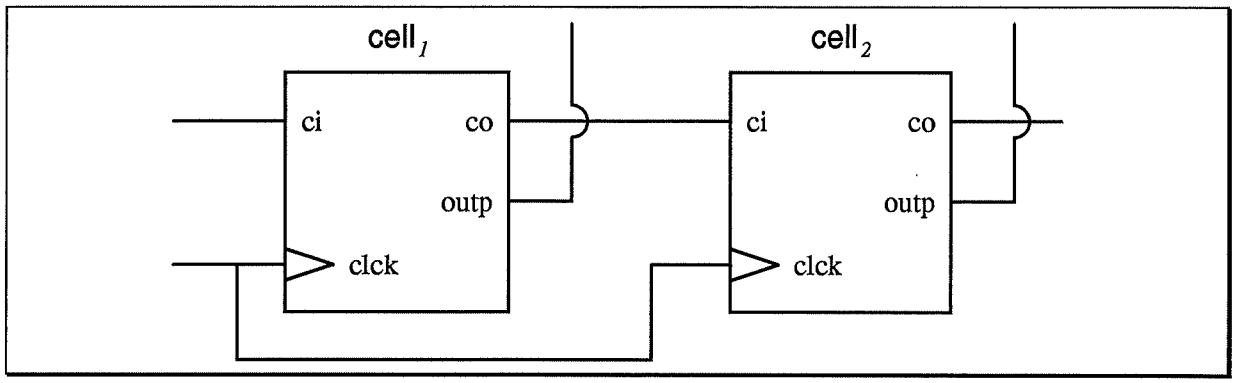
$$
\begin{array}{l}
\lambda t.\texttt{if } inp \texttt{ then} \\
\quad \texttt{if } (t \geq now) \texttt{ then } \{(\text{inp}, inp), (\text{outp}, \neg outp)\} \\
\quad \texttt{else } \{(\text{inp}, inp), (\text{outp}, outp)\} \\
\texttt{else } \{(\text{inp}, inp), (\text{outp}, outp)\}
\end{array}
$$

A trivial case analysis during the proof of the equivalence yields the expected theorem in the HOL system:

$$
\begin{array}{l}
\vdash \forall inp\ outp\ now. \\
\quad \lambda t.\texttt{if } (inp \wedge (t \geq now)) \texttt{ then } \{(\text{inp}, inp), (\text{outp}, \neg outp), (\text{ini}, \mathsf{T})\} \\
\qquad \texttt{else } \{(\text{inp}, inp), (\text{outp}, outp), (\text{ini}, \mathsf{T})\} \\
\qquad\qquad\qquad\qquad \overset{\tau}{=}_{\{\text{inp,outp}\}} \\
\quad \lambda t.\texttt{if } inp \texttt{ then} \\
\qquad \texttt{if } (t \geq now + 1) \texttt{ then} \\
\qquad\quad \{(\text{inp}, inp), (\text{outp}, \neg outp), (11, outp), (13, \neg outp), (14, \mathsf{T}), (15, \mathsf{T})\} \\
\qquad \texttt{elseif } (t \geq now) \texttt{ then} \\
\qquad\quad \{(\text{inp}, inp), (\text{outp}, \neg outp), (11, \neg outp), (13, \neg outp), (14, \mathsf{T}), (15, \mathsf{T})\} \\
\qquad \texttt{else } \{(\text{inp}, inp), (\text{outp}, outp), (11, \neg outp), (13, outp), (14, \mathsf{T}), (15, \mathsf{T})\} \\
\quad\ \texttt{else } \{(\text{inp}, inp), (\text{outp}, outp), (11, \neg outp), (13, outp), (14, \mathsf{T}), (15, \mathsf{T})\}
\end{array}
$$

6.4 A Counter Cell

The parity checker introduced a non-standard state-holding device to ensure quiescence of the design. The study presented here deals with the more traditional notion of state that is exemplified by one cell of a binary counter. The following diagram shows how two such cells may be put together in series make up a two-bit counter.



Each cell has four signals – a clock input clck, a carry input ci, a carry output co and an output outp showing the current state of the count. The clock synchronises the counter cells, and the overall device should be active on the rising edge of the clock. Furthermore, a cell should only count when its carry input is high. The carry output is intended to become high when an individual cell counts up to a high value.

6.4.1 Specification

The Femto-VHDL specification for one cell may easily be extracted from the above natural language description:

$$\text{arch cell high } (\{\text{clck},\text{ci}\},\{\text{outp}\},\{\text{co}\})\,(\text{signal}\,\{\,\})\left(\begin{array}{l}\{\text{ci},\text{outp}\} : \left(\text{co} :\overset{\text{inert}}{=} (\text{ci and outp}, 0)\right) \;||\\ \{\text{clck}\} : \left((\text{clck and ci}) => \text{outp} :\overset{\text{inert}}{=} (\text{not outp}, 1) \mid \text{null}\right)\end{array}\right)$$

The input ports are made up of the clock clck and the carry input ci, the output of the device outp is a bi-directional port and the carry output co is a pure output port. The clocked process inverts the output after 1 unit when the carry input is high and a rising edge is detected. The carry output is calculated in the first process, and will only become high when the carry input and the output of the counter are both high.

The derivation of a general behaviour for this specification of the cell begins in an analogous manner to the previous examples. The term to be used is as expected:

$$\begin{array}{l}\text{"let } \Pi = \text{dflat}\,(\text{arch cell high } (\{\text{clck},\text{ci}\},\{\text{outp}\},\{\text{co}\})\,(\text{signal}\,\{\,\})\ldots)\\ \quad \text{and } \sigma = \{(\text{clck}, clck),(\text{outp}, outp),(\text{ci}, ci),(\text{co}, co)\} \text{ in}\\ \quad \text{let } \mu = (\text{get\_delays\_cs } \Pi, now, \sigma, \{\text{clck}\}, \lambda t.(\{\,\},\{\,\}))\text{ in}\\ \qquad \mu \vdash \langle \Pi, \lambda t.\{\,\}\rangle \xrightarrow{sim} beh\text{"}\end{array}$$

The initial σ makes no special assumptions about the interrelationships of the various signals, and the starting event is on the clock.

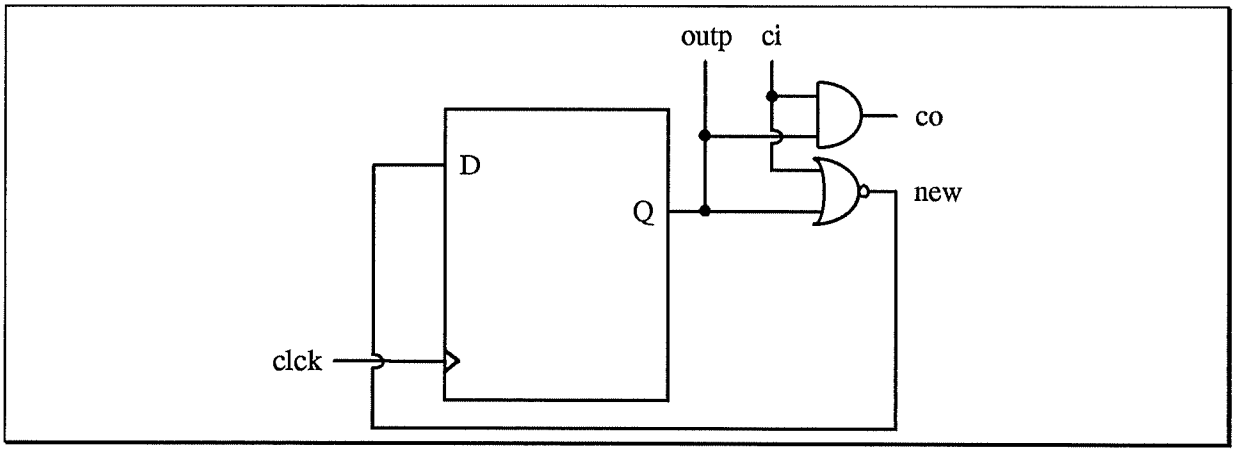
The HOL theorem that emerges after the symbolic simulation demonstrates the expected behaviour for the cell.

$$
\begin{array}{l}
\vdash \forall now\ clck\ outp\ ci\ co. \\
\quad \texttt{let}\ \Pi = \mathsf{dflat}\left(\mathsf{arch}\ \text{cell high}\ (\{\text{clck},\text{ci}\},\{\text{outp}\},\{\text{co}\})\ (\mathsf{signal}\ \{\})\ldots\right) \\
\quad \texttt{and}\ \sigma = \{(\text{clck},clck),(\text{outp},outp),(\text{ci},ci),(\text{co},co)\}\ \texttt{in} \\
\quad \texttt{let}\ \mu = \ldots\ \texttt{in} \\
\qquad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{sim} beh = \\
\left(beh = \lambda t. \left(\begin{array}{l}
\texttt{if}\ ((t \geq now+1) \wedge clck \wedge ci)\ \texttt{then} \\
\quad \{(\text{clck},clck),(\text{outp},\neg outp),(\text{ci},ci),(\text{co},(ci \wedge \neg outp))\} \\
\texttt{elseif}\ (t \geq now)\ \texttt{then} \\
\quad \{(\text{clck},clck),(\text{outp},outp),(\text{ci},ci),(\text{co},co)\} \\
else\ \{\}
\end{array} \right) \right)
\end{array}
$$

Whenever both the clock and the carry input are high, the output is inverted after one unit of time. Otherwise, nothing changes.

6.4.2 Implementation

The previous architecture presented an algorithmic view of the behaviour of the cell. The following diagram shows how it might be realised at the gate level.



The central component is a D-type flip-flop, and it is assumed to have a one unit transmission time from the input D to the output Q whenever the clock has a rising edge. The output of the cell is taken directly from the output of the flip-flop. It is

then conjoined with the carry input to produce the carry output. The output and carry input are also passed through an exclusive-or gate to create the next value for the counter. The next time the clock rises, this is the value that will be transmitted to the output. The delay through each of the combinational gates is zero.

The Femto-VHDL for the implementation of the cell may be read directly from the above schematic:

$$\begin{array}{l}\textsf{arch cell low}\,(\{\text{clck},\text{ci}\},\{\text{outp}\},\{\text{co}\})\,(\textsf{signal}\,\{\text{new}\}) \\ \quad \left(\begin{array}{l}\{\text{ci},\text{outp}\} : \left(\text{co} :\overset{\text{inert}}{=} (\text{ci and outp}, 0)\right) \;||\; \{\text{ci},\text{outp}\} : \left(\text{new} :\overset{\text{inert}}{=} (\text{ci xor outp}, 0)\right) \;|| \\ \{\text{clck}\} : \left((\text{clck}) \Rightarrow \text{outp} :\overset{\text{inert}}{=} (\text{new}, 1) \mid \textsf{null}\right)\end{array}\right)\end{array}$$

The first two processes are simple variations of the combinational gates used in previous examples. The third process represents the flip-flop. Whenever the clock is active and high (a rising edge), the input is conveyed to the output after 1 unit. The state of the output remains unchanged in all other situations. The input to the flip-flop is the internal signal new. Its output is the current count result outp.

The term used to begin the symbolic simulation is almost identical to that used for the specification of the cell.

$$\begin{array}{l}\texttt{"let}\ \Pi = \textsf{dflat}\,(\textsf{arch cell low}\,(\{\text{clck},\text{ci}\},\{\text{outp}\},\{\text{co}\})\,(\textsf{signal}\,\{\text{new}\})\ldots) \\ \texttt{and}\ \sigma = \left\{(\text{clck}, \mathit{clck}), (\text{outp}, \mathit{outp}), (\text{ci}, \mathit{ci}), (\text{co}, \mathit{co}), \left(\text{new}, ((\mathit{ci} \vee \mathit{outp}) \wedge (\neg \mathit{ci} \vee \neg \mathit{outp}))\right)\right\}\ \texttt{in} \\ \texttt{let}\ \mu = \left(\textsf{get\_delays\_cs}\ \Pi, \mathit{now}, \sigma, \{\text{clck}\}, \lambda t.(\{\},\{\})\right)\ \texttt{in} \\ \quad \mu \vdash \langle \Pi, \lambda t.\{\}\rangle \xrightarrow{\mathit{sim}} \mathit{beh}\text{"}\end{array}$$

The only addition is the starting information for the internal signal new. The general behaviour of the parity checker implementation showed that the initial values of such signals need to based on those of externally visible ones. Otherwise, there is no common basis for comparison between two versions of the same device. Here new is given an initial value in terms of the exclusive-or of the output and carry input of the cell.

The behaviour of the implementation follows in a similar fashion to the previous examples:

$$
\begin{array}{l}
\vdash \forall now\ clck\ outp\ ci\ co. \\
\quad \texttt{let}\ \Pi = \ldots \texttt{in}\ \mu \vdash \langle \Pi, \lambda t.\{\} \rangle \xrightarrow{sim} beh = \\
\quad beh = \lambda t.\ \texttt{if}\ ((t \geq now+1) \wedge clck \wedge ci)\ \texttt{then} \\
\qquad \left\{ \begin{array}{l} (\text{clck}, clck), (\text{outp}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp))), \\ (\text{ci}, ci), (\text{co}, (ci \wedge \neg outp)), (\text{new}, outp) \end{array} \right\} \\
\qquad \texttt{elseif}\ (t \geq now)\ \texttt{then} \\
\qquad \left\{ \begin{array}{l} (\text{clck}, clck), (\text{outp}, outp), (\text{ci}, ci), (\text{co}, co), \\ (\text{new}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp))) \end{array} \right\} \\
\qquad \texttt{else}\ \{\}
\end{array}
$$

Since the behaviour of the device contains longer Boolean expressions than those of earlier examples, it is not intuitively clear that the right result has been obtained. One can gain some confidence in the derived behaviour by specialising the variables which represent the values of the signals. The interesting behaviour only occurs when the clock is high. So by specialising the variable *clck* to high, the conditional is reduced to:

$$
\begin{array}{l}
\lambda t.\ \texttt{if}\ ((t \geq now+1) \wedge ci)\ \texttt{then} \\
\quad \left\{ \begin{array}{l} (\text{clck}, \text{T}), (\text{outp}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp))), \\ (\text{ci}, ci), (\text{co}, (ci \wedge \neg outp)), (\text{new}, outp) \end{array} \right\} \\
\quad \texttt{elseif}\ (t \geq now)\ \texttt{then} \\
\quad \left\{ \begin{array}{l} (\text{clck}, \text{T}), (\text{outp}, outp), (\text{ci}, ci), (\text{co}, co), \\ (\text{new}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp))) \end{array} \right\} \\
\quad \texttt{else}\ \{\}
\end{array}
$$

If one further assumes that the carry input is high, the behaviour simplifies to:

$$
\begin{array}{l}
\lambda t.\ \texttt{if}\ (t \geq now+1)\ \texttt{then} \\
\quad \{(\text{clck}, \text{T}), (\text{outp}, \neg outp), (\text{ci}, \text{T}), (\text{co}, \neg outp), (\text{new}, outp)\} \\
\quad \texttt{elseif}\ (t \geq now)\ \texttt{then} \\
\quad \{(\text{clck}, \text{T}), (\text{outp}, outp), (\text{ci}, \text{T}), (\text{co}, co), (\text{new}, \neg outp)\} \\
\quad \texttt{else}\ \{\}
\end{array}
$$

One now observes that the result of the cell counts after 1 unit to the inverse of whatever the initial output was. Furthermore, new has had its own value inverted to remember the previous output of the cell. The carry output also becomes the inverse of the original output.

6.4.3 Equivalence

Given the above reassurance that the implementation is apparently behaving in the right way, the equivalence proof between behaviours for the two versions of the cell may be completed. The proof in the HOL system devolves into a case analysis on time, and results in a theorem of the usual form:

$$
\begin{array}{l}
\vdash \forall clck\ outp\ ci\ co\ now. \\
\quad \lambda t. \left(\begin{array}{l}
\texttt{if } ((t \geq now+1) \wedge clck \wedge ci) \texttt{ then} \\
\quad \{(\text{clck}, clck), (\text{outp}, \neg outp), (\text{ci}, ci), (\text{co}, (ci \wedge \neg outp))\} \\
\texttt{elseif } (t \geq now) \texttt{ then} \\
\quad \{(\text{clck}, clck), (\text{outp}, outp), (\text{ci}, ci), (\text{co}, co)\} \\
else\ \{\}
\end{array} \right) \\
\qquad\qquad \overset{\tau}{=} \\
\qquad\qquad {\scriptstyle \{\text{clck,outp,ci,o}\}} \\
\quad \lambda t. \texttt{if } ((t \geq now+1) \wedge clck \wedge ci) \texttt{ then} \\
\qquad \left\{ \begin{array}{l}
(\text{clck}, clck), (\text{outp}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp))), \\
(\text{ci}, ci), (\text{co}, (ci \wedge \neg outp)), (\text{new}, outp)
\end{array} \right\} \\
\quad \texttt{elseif } (t \geq now) \texttt{ then} \\
\qquad \left\{ \begin{array}{l}
(\text{clck}, clck), (\text{outp}, outp), (\text{ci}, ci), (\text{co}, co), \\
(\text{new}, ((ci \vee outp) \wedge (\neg ci \vee \neg outp)))
\end{array} \right\} \\
\quad \texttt{else } \{\}
\end{array}
$$

CHAPTER 7

CONCLUSIONS

In the previous chapters, an exposition of the semantics of Femto-VHDL, a subset of full VHDL, was presented. The semantics was defined in an operational manner that closely adheres to the informal description of the simulation model of full VHDL. Furthermore, it has been embedded in the HOL system in order to automate reasoning about programs written in the subset.

7.1 The Semantics

In contrast to the approaches highlighted in Chapter 2, the semantics given in Chapters 3 and 4 was derived independently of the proof assistant in which it has been embedded. As a result, it is not bound by constraints of the object system, and the most suitable formalism for describing such a dynamic system as VHDL was able to be used.

Because of the precision provided by using the operational framework, the characterisation presented could be used as part of the specification for the design of simulators. In a full semantics for the language, it would not merely constitute a part of such a specification, but encompass it entirely. Since the semantics has been written to take into account the implied concurrency of VHDL processes, it need not simply specify sequential simulators, but also high-performance, parallel ones.

Since a uniform approach was taken in characterising what is meant by an individual simulation cycle, there has been no need to treat 0-delay signal assignments any differently from unit-delay ones. As was seen in the examples of Chapter 6, the simulation kernel itself provides the abstraction mechanism for resolving multiple 0-delay steps into an overall view of "instantaneous" assignments. The key to understanding the way VHDL works with time was to do away with it entirely. In its stead, the concept of a point of computation was used.

One important property about the semantics has also been proved. Theorem 4.4.1 reinforces one of the most basic tenets of any hardware description language.

It shows that, given the appropriate assumptions about quiescence and well-formedness, sub-components of a larger design may be replaced by equivalent ones.

7.2 The Methodology

The methodology used in working with the embedded semantics in Chapter 6 is analogous to those used in the projects described in Chapter 2. Specifically, it has been possible to symbolically simulate Femto-VHDL programs via proof to gain a general understanding of their behaviour. Once obtained, such behaviours may be compared, again through formal proof, for equivalence or non-equivalence to other similarly derived ones.

There are three problems with the methodology. The first is that the user must carefully specify the initial signal interrelationships in order to keep the analysis from degenerating into an exhaustive simulation. The second is that there a great deal of user interaction involved in producing any kind of result. Finally, analysis is limited to only those Femto-VHDL program which quiesce.

7.3 The Future

Formal analysis of the VHDL simulation model has been a fruitful exercise. The research presented here has however been constrained to only a subset of the language and a restricted form of reasoning. Furthermore, analysis of individual programs, despite automation of the semantics in a proof assistant, is also a very user-intensive process.

As noted earlier, a formal semantics of full VHDL is a useful object in and of itself, as it would provide a clear specification of the language for future implementors. Indeed, it has been used directly in the development of a symbolic simulator in the HOL system. It also has use as an unambiguous reference for language practitioners. Since VHDL is used by such a wide cross-section of the CAD community, such a specification of the full language would be of immediate relevance and use. Scope for additional research in the development of such a semantics could be found in crystallising the operation of user-defined the resolution functions. Recall that these been simplified out of the work presented here.

The methodology for reasoning about individual Femto-VHDL programs that has been presented could in theory be extended to cover the full language. The inherent problems associated with excessive user intervention will, however, persist

unless addressed. Additional questions need to be asked about the division of labour in the whole effort surrounding formal analysis of VHDL programs. Should, for instance, the approach taken by Siemens in providing an efficient oracle be taken? That way the symbolic simulator would be left to do what it does best, and the proof system is left unencumbered by a task for which it may not be suited. Certainly given a formal semantics for the language, such a system is not inconceivable. Further research must also be done into ways in which non-quiescent programs may be compared.

The way forward in the formal analysis of VHDL needs to stem from a firm understanding of the semantics of the full language. The work that has been presented here is a first step towards that goal. Further research, because of the real-world nature of its implications to the design of complex systems, represents a true test for the maturity of the formal methods approach. The benefits of verification over validation have been expressed for quite some time. Furthermore, formal systems and methodologies abound. A formal semantics and practical associated tools for full VHDL presents a unique and realisable challenge for the future.

BIBLIOGRAPHY

[1] The Aerospace Corp., *SDVS 11 User's Manual*, Engineering and Technology Group, 30 September 1992.

[2] R. Airiau, J-M Bergé, V. Olive and J. Rouillard, *VHDL: du Langage à la Modélisation*, Presses Polytechniques et Universitaires Romandes (1990).

[3] J. Armstrong, *Chip Level Modelling in VHDL*, Prentice Hall (1988).

[4] L.M. Augustin, D.C. Luckham, B.A. Gennart and A.C. Stanculescu, *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers (1991).

[5] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham and A.C. Stanculescu, 'An Overview of VAL', Technical Report CSL-TR-88-367, Stanford University, Stanford, California (October 1988).

[6] H. Barringer, G. Gough and B. Monahan, 'Operational Semantics for Hardware Description Languages', in proceedings *1992 Wokshop for the ESPRIT BRA CHARME*, (also University of Manchester Department of Computer Science Technical Report UMCS-91-2-2 (February 1991)).

[7] M. Belhadj, R. McConnell and P. Le Guernic, 'A Synchronous Framework for VHDL Attributes', IRISA Internal Report (January 1993).

[8] R. Boulton, 'The HOL Arith Library', HOL System Documentation, University of Cambridge Computer Laboratory (1992).

[9] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert and J. Van Tassel, 'Experience with Embedding Hardware Description Languages in HOL', in proceedings: *IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Designs: Theory, Practice and Experience*, edited by V. Stavridou, T. F. Melham and R. T. Boute, North-Holland (1992).

[10] R. Boulton, M. Gordon, J. Herbert and J. Van Tassel, 'The HOL Verification of ELLA Designs', in proceedings: *1991 International Workshop on Formal Methods in VLSI Design*, edited by P.A. Subrahmanyam (also University of Cambridge Technical Report 199, August 1990), Springer-Verlag (1991).

[11] R. S. Boyer and J S. Moore, *A Computational Logic*, Academic Press (1979).

[12] R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press (1988)

[13] J. Camilleri, 'Symbolic Compilation and Execution of Programs by Proof: A Case Study in HOL', University of Cambridge Computer Laboratory Technical Report 240, Cambridge, England (December 1991).

[14] S. Carlson, *Introduction to HDL-Based Design Using VHDL*, Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043.

[15] D. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers (1989).

[16] K. Davis, 'A Denotational Definition of the VHDL Simulation Kernel', in proceedings *IFIP WG 10.2 Twelfth International Symposium on Computer Hardware Description Languages and their Applications*, (to appear 1993).

[17] I.V. Filippenko, 'VHDL Verification in the State Delta Verification System (SDVS)', in proceedings: *1991 International Workshop on Formal Methods in VLSI Design*, edited by P.A. Subrahmanyam, Springer-Verlag (1991).

[18] M.J.C. Gordon, 'HOL: A Proof Generating System for Higher-Order Logic', University of Cambridge Computer Laboratory Technical Report 103 (1987), revised version in *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer (1987).

[19] M.J.C. Gordon and T.F. Melham (editors), *Introduction to HOL: A Theorem-Proving Environment for Higher Order Logic*, Cambridge University Press (1993).

[20] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction Using Structured Operational Semantics*, John Wiley and Sons (1990).

[21] Institute of Electrical and Electronics Engineers, *IEEE Standard VHDL Language Reference Manual*, IEEE Press, New York (1988).

[22] J.C. Laprie, B. Courtois, M.C. Gaudel and D. Powell, *Sûreté de Fonctionnement des Systèmes Informatiques*, BORDAS, Paris (1989).

[23] R. Lipsett, C. Shaefer and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers (1989).

[24] M.M. Mano, *Digital Design*, Prentice-Hall (1984).

[25] T.F. Melham, 'A Package for Inductive Relation Definitions in HOL', in proceedings: *1991 International Workshop on the HOL Theorem Proving System and its Applications*, IEEE Computer Society Press (1991).

[26] T.F. Melham, 'The HOL Sets Library', HOL System Documentation, University of Cambridge Computer Laboratory (1992).

[27] T.F. Melham, 'The HOL String Library', HOL System Documentation, University of Cambridge Computer Laboratory (1992).

[28] T.F. Melham, 'Automating Recursive Type Definitions in Higher-Order Logic', in proceedings: *Current Trends in Hardware Verification and Automated Deduction*, edited by G. Birtwistle and P.A. Subrahmanyam, Springer-Verlag (1988).

[29] T.F. Melham, 'Abstraction Mechanisms for Hardware Verification', University of Cambridge Computer Laboratory Technical Report 106, Cambridge, England (May 1987).

[30] F.G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Inc., (1981).

[31] D.L. Perry, *VHDL*, McGraw-Hill (1991).

[32] G. Plotkin, 'A Structural Approach to Operational Semantics', Technical Report DAIMI FN-19, Computer Science Dept., Århus University (September 1981).

[33] A. Salem and D. Borrione, 'Formal Reasoning About Signal Attributes in VHDL', in proceedings: *VHDL Forum for CAD in Europe*, Spring 1991 meeting.

[34] V. Stavridou, J.A. Goguen, A. Stevens, S.M. Eker, S.N. Aloneftis and K.M. Hobley, 'FUNNEL and 2OBJ: Towards an Integrated Hardware Design Environment', in proceedings: *IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Designs: Theory, Practice and Experience*, edited by V. Stavridou, T. F. Melham and R. T. Boute, North-Holland, (1992).

[35] G. Umbreit, 'Providing a VHDL Interface for Proof Systems', in proceedings: *European Design Automation Conference* (1992).

[36] J.P. Van Tassel, *The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools*, MSc Thesis, Dept. of Computer Science and Engineering, Wright State University, Dayton, Ohio (1989).

[37] J.P. Van Tassel and D. Hemmendinger, 'Towards Formal Verification of VHDL Specifications', in proceedings: *Applied Formal Methods for Correct VLSI Design*, edited by L. Claesen, Elsevier Science Publishers (1990).

[38] J.P. Van Tassel, 'A Formalisation of the VHDL Simulation Cycle', University of Cambridge Computer Laboratory Technical Report 249, Cambridge, England (March 1992).

[39] W. Young, unpublished technical notes, Computational Logic, Inc., 1993.

APPENDIX A

HOL PROOFS

This appendix gives the HOL scripts used to prove the theorems shown in Chapters 3 and 4.

A.1 Theorem 3.3.2.1

```
let well_formed_DESIGN_sigs_lemma =
    prove_thm(`well_formed_DESIGN_sigs_lemma`,
    "!D. (FST(well_formed_DESIGN_sigs D)) ==>
         !sig. (sig IN (get_PORTS D)) ==>
               (sig IN (get_sigs_DESIGN D))",
    let pair = GEN_ALL (SYM (SPEC_ALL PAIR)) in
    let th1 =
        let x = INST_TYPE [(":(name)set",":*");(":(name)set",":**")]
                          pair
        in
            SPEC "get_sigs_CS C" x
    and (th2,th3) =
        let x = INST_TYPE [(":bool",":*");(":(name)set",":**")]
                          pair
        in
           (SPEC "well_formed_DESIGN_sigs D" x,
            SPEC "well_formed_DESIGN_sigs D'" x)
    in
    INDUCT_THEN DESIGN_Induct ASSUME_TAC THEN
    REWRITE_TAC [get_PORTS;get_sigs_DESIGN;IN_UNION;
                 well_formed_DESIGN_sigs] THENL
    [CONV_TAC (DEPTH_CONV let_CONV) THEN REWRITE_TAC [] THEN
     PURE_ONCE_REWRITE_TAC [th1] THEN
     CONV_TAC (DEPTH_CONV let_CONV) THEN REWRITE_TAC [] THEN
     REPEAT GEN_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC [] THEN
     REWRITE_TAC [IN_UNION] THEN
     REPEAT STRIP_TAC THEN ASM_REWRITE_TAC []
    ;PURE_ONCE_REWRITE_TAC [th2;th3] THEN
     CONV_TAC (ONCE_DEPTH_CONV let_CONV) THEN
     REWRITE_TAC[] THEN STRIP_TAC THEN RES_TAC THEN
     REPEAT STRIP_TAC THEN RES_TAC THEN ASM_REWRITE_TAC[]]);;
```

A.2 Theorem 3.3.3.1

```
let lemma4a =
    let th1 = ASSUME "well_formed_DESIGN design" in
    let th2 = REWRITE_RULE [well_formed_DESIGN] th1 in
    let th3 = CONJUNCT1 th2 in
        GEN_ALL (DISCH_ALL th3)
```

A.3 Theorem 4.2.3.1

```
let th1 = PROVE(
    "!A B ports sig val t.
      ((EQ_TAU A B ports) /\ (sig IN ports) /\ (sig,val) IN (A t)) ==>
      (sig,val) IN (B t)",
    REPEAT GEN_TAC THEN
    REWRITE_TAC [definition `Femto-config` `EQ_TAU`] THEN
    CONV_TAC (ONCE_DEPTH_CONV let_CONV) THEN
    REWRITE_TAC [EXTENSION] THEN
    CONV_TAC (ONCE_DEPTH_CONV SET_SPEC_CONV) THEN
    CONV_TAC (TOP_DEPTH_CONV LEFT_AND_FORALL_CONV) THEN
    CONV_TAC (TOP_DEPTH_CONV LEFT_IMP_FORALL_CONV) THEN
    EXISTS_TAC "t:time" THEN EXISTS_TAC "(sig:name,val:value)" THEN
    STRIP_TAC THEN
    FIRST_ASSUM
     \th. let (a,_)=EQ_IMP_RULE th in
          let th1=CONV_RULE (TOP_DEPTH_CONV LEFT_IMP_EXISTS_CONV) a in
          let th2=SPECL ["sig:name";"val:value"] th1 in
              ASSUME_TAC (REWRITE_RULE [] th2) THEN
    RES_TAC THEN ASM_REWRITE_TAC[]);;
```

A.4 Theorem 4.2.3.2

```
let th2 =
    let a=ONCE_REWRITE_RULE[theorem `Femto-config` `EQ_TAU_SYM`]th1 in
    let b=SPECL ["B:transactions";"A:transactions"] a in
        GENL ["A:transactions";"B:transactions"] b;;
```

A.5 Theorem 4.3.3.1

```
let Zip_ZIP_EQ =
    let ZIP = definition `cs-rules` `ZIP`
    and Zip = definition `cs-rules` `Zip` in
    let th = REWRITE_RULE [SYM (SPEC_ALL ZIP)] Zip in
        CONV_RULE (ONCE_DEPTH_CONV let_CONV) th;;
```

A.6 Theorem 4.3.3.2

```
let ZIP_COMM = PROVE(
    "!A B. (A ZIP B) = (B ZIP A)",
    REWRITE_TAC [ZIP] THEN
    CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV) THEN
    REPEAT GEN_TAC THEN BETA_TAC THEN
    REWRITE_TAC [EXTENSION] THEN GEN_TAC THEN
    EQ_TAC THEN STRIP_TAC THEN ONCE_REWRITE_TAC [UNION_COMM] THEN
    ASM_REWRITE_TAC[]);;
```

A.7 Theorem 4.3.3.3

```
let ZIP_ASSOC = PROVE(
    "!Tau Tau' Tau''.
      (Tau ZIP (Tau' ZIP Tau'')) = ((Tau ZIP Tau') ZIP Tau'')",
    REPEAT GEN_TAC THEN
    REWRITE_TAC [definition `cs-rules` `ZIP`] THEN
    BETA_TAC THEN
    REWRITE_TAC[theorem `sets` `UNION_ASSOC`]);;
```

A.8 Theorem 4.3.3.4

```
let ZIP_EMPTY = PROVE(
    "(!tau. (\t.{}) ZIP tau = tau) /\ (!tau. tau ZIP (\t.{}) = tau)",
    REWRITE_TAC [definition `cs-rules` `ZIP`;UNION_EMPTY;ETA_AX]);;
```

A.9 Theorem 4.3.3.5

```
let CLEAN_ZIP_EMPTY = PROVE(
    "!A B. (CLEAN_ZIP (\t.{}) A B) = (A ZIP B)",
    REPEAT GEN_TAC THEN REWRITE_TAC [CLEAN_ZIP;ZIP] THEN
    CONV_TAC (DEPTH_CONV let_CONV) THEN BETA_TAC THEN
    REWRITE_TAC [DIFF_EMPTY;EMPTY_DIFF;UNION_EMPTY;ETA_AX]);;
```

A.10 Theorem 4.3.3.6

```
let CLEAN_ZIP_SYM = PROVE(
    "!orig A B. (CLEAN_ZIP orig A B) = (CLEAN_ZIP orig B A)",
    REPEAT GEN_TAC THEN REWRITE_TAC [CLEAN_ZIP;ZIP] THEN
    CONV_TAC (DEPTH_CONV let_CONV) THEN
    CONV_TAC FUN_EQ_CONV THEN BETA_TAC THEN
    REWRITE_TAC [IN_IMAGE;DIFF_DEF;UNION_DEF;EXTENSION] THEN
    REPEAT GEN_TAC THEN CONV_TAC  (DEPTH_CONV SET_SPEC_CONV) THEN
    CONV_TAC (TOP_DEPTH_CONV NOT_EXISTS_CONV) THEN
    REWRITE_TAC [DE_MORGAN_THM] THEN
    CONV_TAC (ONCE_DEPTH_CONV NOT_EXISTS_CONV) THEN
    REWRITE_TAC [DE_MORGAN_THM] THEN EQ_TAC THEN
    CONV_TAC (TOP_DEPTH_CONV RIGHT_AND_FORALL_CONV) THEN
    CONV_TAC (TOP_DEPTH_CONV RIGHT_OR_FORALL_CONV) THEN
    CONV_TAC (TOP_DEPTH_CONV RIGHT_IMP_FORALL_CONV) THEN
    REPEAT GEN_TAC THEN
    CONV_TAC (ONCE_DEPTH_CONV LEFT_OR_FORALL_CONV) THEN
    CONV_TAC (ONCE_DEPTH_CONV RIGHT_OR_FORALL_CONV) THEN
    CONV_TAC (ONCE_DEPTH_CONV RIGHT_AND_FORALL_CONV) THEN
    CONV_TAC (ONCE_DEPTH_CONV RIGHT_OR_FORALL_CONV) THEN
    CONV_TAC RIGHT_IMP_FORALL_CONV THEN GEN_TAC THEN
    CONV_TAC (TOP_DEPTH_CONV LEFT_IMP_FORALL_CONV) THEN
    EXISTS_TAC "x':name" THEN EXISTS_TAC "y:value" THEN
    EXISTS_TAC "x'':(name#value)" THEN REPEAT STRIP_TAC THEN
    ASM_REWRITE_TAC[]);;
```

A.11 Theorem 4.3.5.3

```
let ADD_TO_THETA_SIMP = prove_thm(`ADD_TO_THETA_SIMP`,
    "!sigma' gamma' sigma'' gamma'' Tau now later.
      ADD_TO_THETA sigma' gamma'
                   (ADD_TO_THETA sigma'' gamma'' Tau now later)
                   now later =
      ADD_TO_THETA sigma' gamma' Tau now later",
     REPEAT GEN_TAC
       THEN REWRITE_TAC [definition `sim-rules` `ADD_TO_THETA`]
       THEN CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV) THEN GEN_TAC
       THEN BETA_TAC THEN COND_CASES_TAC
       THEN REWRITE_TAC [] THEN COND_CASES_TAC THEN REWRITE_TAC []);;
```

A.12 Theorem 4.4.1

```
let lemma1 = PROVE(
    "!mu Pr tau beh.
      (Simulate mu Pr tau beh /\
       QUIESCE(Sim_ENV_DLYS mu)(Sim_ENV_TIME mu)tau
              (Sim_ENV_GAMMA mu)) ==>
      (beh =
       (\t. (SND ((ADD_TO_THETA
                   (Sim_ENV_SIGMA mu)
                   (Sim_ENV_GAMMA mu)
                   (Sim_ENV_THETA mu)
                   (Sim_ENV_TIME mu)
                   (Sim_ENV_TIME mu)) t))))",
    REPEAT GEN_TAC THEN ONCE_REWRITE_TAC [GEN_Simulate] THEN
    DISCH_THEN (ASSUME_TAC o \th. let [x;y] = CONJUNCTS th in
                                  REWRITE_RULE [y] x) THEN
    ASM_REWRITE_TAC[]);;

let fwd th =
    let (th1,_) = EQ_IMP_RULE th in
    let th2 = CONV_RULE (ONCE_DEPTH_CONV LEFT_AND_EXISTS_CONV) th1 in
    let th3 = CONV_RULE (DEPTH_CONV RIGHT_AND_EXISTS_CONV) th2 in
    let th4 = CONV_RULE (TOP_DEPTH_CONV LEFT_IMP_EXISTS_CONV) th3 in
    let th5 = SPECL ["FST (x:(name#value))";
                     "SND (x:(name#value))"] th4
    in
        REWRITE_RULE [] th5;;
```

```
let SIMENV_CONV =
    let check1 = assert (\c. let tmp = fst (dest_const c) in
                                ((tmp=`Sim_ENV_DLYS`) or
                                 (tmp=`Sim_ENV_RHO`) or
                                 (tmp=`Sim_ENV_TIME`) or
                                 (tmp=`Sim_ENV_SIGMA`) or
                                 (tmp = `Sim_ENV_GAMMA`) or
                                 (tmp = `Sim_ENV_THETA`)))
    and check2 = assert (\l. not (is_var (hd l)))
    and SIMENV = map (definition `Femto-config`)
                     [`Sim_ENV_DLYS`;`Sim_ENV_RHO`;`Sim_ENV_TIME`;
                      `Sim_ENV_SIGMA`;`Sim_ENV_GAMMA`;`Sim_ENV_THETA`]
    in
    letrec build_conv lst conv =
       if null lst then conv
       else build_conv (tl lst) (conv ORELSEC (REWRITE_CONV (hd lst)))
    in
    let conv = (build_conv (tl SIMENV) (REWRITE_CONV (hd SIMENV)))
               THENC DEPTH_CONV (REWRITE_CONV FST
                                 ORELSEC REWRITE_CONV SND) in
    \tm. (let (_,_) = (check1 # check2) (strip_comb tm) in
          conv tm) ? failwith `SIMENV_CONV`;;

let (lemma2a,lemma2b,lemma2c) =
    let tac =
      REWRITE_TAC [EQ_TAU] THEN CONV_TAC (ONCE_DEPTH_CONV let_CONV) THEN
      CONV_TAC (ONCE_DEPTH_CONV LEFT_AND_FORALL_CONV) THEN
      CONV_TAC LEFT_IMP_FORALL_CONV THEN
      EXISTS_TAC "now:time" THEN REWRITE_TAC [EXTENSION;IN_IMAGE] THEN
      CONV_TAC (ONCE_DEPTH_CONV SET_SPEC_CONV) THEN
      CONV_TAC (ONCE_DEPTH_CONV LEFT_AND_FORALL_CONV) THEN
      CONV_TAC (LEFT_IMP_FORALL_CONV) THEN
      EXISTS_TAC "x:(name#value)" THEN
      CONV_TAC (DEPTH_CONV LEFT_AND_FORALL_CONV) THEN
      CONV_TAC (DEPTH_CONV RIGHT_AND_FORALL_CONV) THEN
      CONV_TAC LEFT_IMP_FORALL_CONV THEN
      EXISTS_TAC "x:(name#value)" THEN
      CONV_TAC (ONCE_DEPTH_CONV RIGHT_AND_FORALL_CONV) THEN
      CONV_TAC LEFT_IMP_FORALL_CONV THEN
      EXISTS_TAC "x:(name#value)" THEN
      STRIP_TAC THEN FIRST_ASSUM (ASSUME_TAC o fwd) THEN
      UNDISCH_TAC
        "(FST x) IN
         ((FST ports) UNION
          ((FST(SND ports)) UNION (SND(SND ports)))) /\
         x IN (tau' now) ==>
         (?sig val'.
           (x = sig,val') /\
           sig IN
           ((FST ports) UNION
            ((FST(SND ports)) UNION (SND(SND ports)))) /\
           (sig,val') IN ((tau'':transactions) now))" THEN
      ASM_REWRITE_TAC[IN_UNION] THEN STRIP_TAC THEN
      UNDISCH_TAC "x=(sig:name,val':value)" THEN
      DISCH_THEN (ASSUME_TAC o SYM) THEN
      UNDISCH_TAC "(sig,val') IN ((tau'':transactions) now)" THEN
      FILTER_ASM_REWRITE_TAC
           (\tm. (rhs tm = "x:(name#value)")?false)[] THEN
      FILTER_ASM_REWRITE_TAC
           (\tm. (rhs tm = "x IN sigma'':state")?false)[]
    in
```

```
    (GEN_ALL (PROVE(
        "EQ_TAU tau' tau''((FST ports) UNION
                           ((FST(SND ports)) UNION (SND(SND ports)))) /\
         (tau' now = sigma') /\
         (tau'' now = sigma'') /\
         (FST x) IN (FST ports) /\
         x IN sigma' ==>
         x IN sigma'''",tac)),
     GEN_ALL (PROVE(
        "EQ_TAU tau' tau''((FST ports) UNION
                           ((FST(SND ports)) UNION (SND(SND ports)))) /\
         (tau' now = sigma') /\
         (tau'' now = sigma'') /\
         (FST x) IN (FST (SND ports)) /\
         x IN sigma' ==>
         x IN sigma'''",tac)),
     GEN_ALL (PROVE(
        "EQ_TAU tau' tau''((FST ports) UNION
                           ((FST(SND ports)) UNION (SND(SND ports)))) /\
         (tau' now = sigma') /\
         (tau'' now = sigma'') /\
         (FST x) IN (SND (SND ports)) /\
         x IN sigma' ==>
         x IN sigma'''",tac)));;

let lemma2a' = ONCE_REWRITE_RULE [EQ_TAU_SYM] lemma2a
and lemma2b' = ONCE_REWRITE_RULE [EQ_TAU_SYM] lemma2b
and lemma2c' = ONCE_REWRITE_RULE [EQ_TAU_SYM] lemma2c;;

let lemma3a = GEN_ALL (PROVE(
   "((EQ_SIGMA (sigma' UNION sigma''') (sigma'' UNION sigma''')
               (get_PORTS X))/\
     ((FST x) IN (get_PORTS X)) /\
     (x IN sigma')) ==> ((x IN sigma'') \/ (x IN sigma'''))",
   REWRITE_TAC [EQ_SIGMA] THEN CONV_TAC (ONCE_DEPTH_CONV let_CONV) THEN
   REWRITE_TAC [EXTENSION;IN_UNION;IN_IMAGE] THEN
   CONV_TAC (ONCE_DEPTH_CONV LEFT_AND_FORALL_CONV) THEN
   CONV_TAC LEFT_IMP_FORALL_CONV THEN EXISTS_TAC "x:(name#value)" THEN
   CONV_TAC (ONCE_DEPTH_CONV SET_SPEC_CONV) THEN
   STRIP_TAC THEN FIRST_ASSUM (ASSUME_TAC o fwd) THEN
   UNDISCH_TAC
     "(x IN sigma' \/ x IN sigma''') /\ (FST x) IN (get_PORTS X) ==>
      (?x' y'.
        (x = x',y') /\
        ((x',y':value) IN sigma'' \/ (x',y') IN sigma''') /\
        x' IN (get_PORTS X))" THEN
   ASM_REWRITE_TAC [] THEN STRIP_TAC THEN
   FILTER_ASM_REWRITE_TAC (\tm. (lhs tm = "x:(name#value)")?false)[] THENL
    [FILTER_ASM_REWRITE_TAC
      (\tm. (tm = "(x',y') IN sigma'':state")?false)[]
    ;FILTER_ASM_REWRITE_TAC
      (\tm. (tm = "(x',y') IN sigma''':state")?false)[]]));;

let lemma3a' = ONCE_REWRITE_RULE [EQ_SIGMA_SYM] lemma3a;;

let tac1 =
let tac1a = REWRITE_TAC [DFLAT;get_PORTS] THEN STRIP_TAC
and lemma = GEN_ALL (CONV_RULE (ONCE_DEPTH_CONV SIMENV_CONV)
                    (SPEC "(dlys,now,sigma,gamma,theta):sim_env"
                          lemma1)) in
    tac1a THEN IMP_RES_TAC lemma;;
```

```
let lemma4a =
    let th1 = ASSUME "well_formed_DESIGN design" in
    let th2 = REWRITE_RULE [well_formed_DESIGN] th1 in
    let th3 = CONJUNCT1 th2 in
        GEN_ALL (DISCH_ALL th3)
and lemma4b =
    let th1 = ASSUME "well_formed_theta theta design" in
    let th2 = REWRITE_RULE [well_formed_theta] th1 in
    let th3 = CONV_RULE let_CONV th2 in
    let th4 = CONJUNCT2 (SPEC_ALL th3) in
    let th5 = DISCH_ALL (GEN_ALL th4) in
        GEN_ALL th5;;

let tac2 =
   let tac = CONV_TAC (ONCE_DEPTH_CONV LEFT_IMP_FORALL_CONV) THEN
             EXISTS_TAC "t:time" THEN ASM_REWRITE_TAC [] THEN DISCH_TAC
   in
   ONCE_ASM_REWRITE_TAC[] THEN REWRITE_TAC [ADD_TO_THETA;EQ_TAU] THEN
   CONV_TAC (DEPTH_CONV let_CONV) THEN BETA_TAC THEN GEN_TAC THEN
   REWRITE_TAC [EXTENSION] THEN
   CONV_TAC (ONCE_DEPTH_CONV SET_SPEC_CONV) THEN
   GEN_TAC THEN REWRITE_TAC [IN_UNION;IN_IMAGE] THEN
   COND_CASES_TAC THEN REWRITE_TAC [IN_UNION] THEN
   UNDISCH_TAC
     "behA = (\t. SND (ADD_TO_THETA sigmaA gammaA thetaA now now t))"
   THEN
   UNDISCH_TAC
     "behB = (\t. SND (ADD_TO_THETA sigmaB gammaB thetaB now now t))"
   THEN
   UNDISCH_TAC
     "behAC = (\t. SND (ADD_TO_THETA (sigmaA UNION sigmaC)
                                     (gammaA UNION gammaC)
                                     (\t.
                                       ((FST(thetaA t)) UNION
                                        (FST(thetaC t)),
                                        (SND(thetaA t)) UNION
                                        (SND(thetaC t))))
                                    now now t))" THEN
   UNDISCH_TAC
     "behBC = (\t. SND (ADD_TO_THETA (sigmaB UNION sigmaC)
                                     (gammaB UNION gammaC)
                                     (\t.
                                       ((FST(thetaB t)) UNION
                                        (FST(thetaC t)),
                                        (SND(thetaB t)) UNION
                                        (SND(thetaC t))))
                                    now now t))" THEN
   REWRITE_TAC [ADD_TO_THETA] THEN
   CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV) THEN
   BETA_TAC THEN REPEAT tac THEN
   REWRITE_TAC [IN_UNION] THEN
   EQ_TAC THEN STRIP_TAC THEN EXISTS_TAC "sig:name" THEN
   EXISTS_TAC "val:value" THEN ASM_REWRITE_TAC[];;

let lemma2_TAC lemma lst =
    let th = SPECL lst lemma in
     ASSUME_TAC th THEN UNDISCH_TAC (concl th) THEN
     ASM_REWRITE_TAC[] THEN STRIP_TAC THEN ASM_REWRITE_TAC[];;
```

```
let lemma3_TAC lemma lst =
    let th = SPECL lst lemma in
    ASSUME_TAC th THEN UNDISCH_TAC (concl th) THEN
    ASM_REWRITE_TAC[];;

let tac3a = lemma2_TAC lemma2a
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaA:state";
                        "sigmaB:state";"(sig:name,val:value)"]
and tac3b = lemma2_TAC lemma2b
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaA:state";
                        "sigmaB:state";"(sig:name,val:value)"]
and tac3c = lemma2_TAC lemma2c
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaA:state";
                        "sigmaB:state";"(sig:name,val:value)"]
and tac3d = lemma3_TAC lemma3a
                       ["sigmaA:state";"sigmaC:state";"sigmaB:state";
                        "C:DESIGN";"x:(name#value)"]
and tac3e = lemma2_TAC lemma2a'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaB:state";
                        "sigmaA:state";"(sig:name,val:value)"]
and tac3f = lemma2_TAC lemma2b'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaB:state";
                        "sigmaA:state";"(sig:name,val:value)"]
and tac3g = lemma2_TAC lemma2c'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "now:time";"sigmaB:state";
                        "sigmaA:state";"(sig:name,val:value)"]
and tac3h = lemma3_TAC lemma3a'
                       ["sigmaB:state";"sigmaC:state";"sigmaA:state";
                        "C:DESIGN";"x:(name#value)"];;

let tac4a = lemma2_TAC lemma2a
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaA:trace) t)";
                        "SND ((thetaB:trace))";"(sig:name,val:value)"]
and tac4b = lemma2_TAC lemma2b
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaA:trace) t)";
                        "SND ((thetaB:trace))";"(sig:name,val:value)"]
and tac4c = lemma2_TAC lemma2c
                       ["behA:transactions";"behB:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaA:trace) t)";
                        "SND((thetaB:trace))";"(sig:name,val:value)"];;
```

```
let tac4d =
    IMP_RES_TAC lemma4a THEN
    IMP_RES_TAC well_formed_DESIGN_sigs_lemma THEN
    IMP_RES_TAC lemma4b THEN
    ASM_REWRITE_TAC []
and tac4e = lemma2_TAC lemma2a'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaB:trace) t)";
                        "SND ((thetaA:trace))";"(sig:name,val:value)"]
and tac4f = lemma2_TAC lemma2b'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaB:trace) t)";
                        "SND ((thetaA:trace))";"(sig:name,val:value)"]
and tac4g = lemma2_TAC lemma2c'
                       ["behB:transactions";"behA:transactions";
                        "ports:((name)set#(name)set#(name)set)";
                        "t:time";"SND ((thetaB:trace) t)";
                        "SND(thetaA:trace))";"(sig:name,val:value)"];;

let QUIET_CONG_THM = save_thm (`QUIET_CONG_THM`,GEN_ALL (PROVE (
 "let A = ARCH ent archA ports declsA csA
  and B = ARCH ent archB ports declsB csB
  and dlysAC = ADD_DELAYS_CS dlysA dlysC
  and dlysBC = ADD_DELAYS_CS dlysB dlysC
  and sigmaAC = sigmaA UNION sigmaC
  and sigmaBC = sigmaB UNION sigmaC
  and gammaAC = gammaA UNION gammaC
  and gammaBC = gammaB UNION gammaC
  and thetaAC t = ((FST (thetaA t)) UNION (FST (thetaC t)),
                   (SND (thetaA t)) UNION (SND (thetaC t)))
  and thetaBC t = ((FST (thetaB t)) UNION (FST (thetaC t)),
                   (SND (thetaB t)) UNION (SND (thetaC t))) in
  let ports' = get_PORTS A
  and ports'' = get_PORTS C in
  let prts = ports' UNION ports'' in
      ((Simulate (dlysA,now,sigmaA,gammaA,thetaA)(DFLAT A) tauA behA /\
        QUIESCE dlysA now tauA gammaA /\
        Simulate (dlysB,now,sigmaB,gammaB,thetaB)(DFLAT B) tauB behB /\
        QUIESCE dlysB now tauB gammaB /\
        Simulate (dlysAC,now,sigmaAC,gammaAC,thetaAC)(DFLAT (DPAR A C))
                 tauAC behAC /\
        QUIESCE dlysAC now tauAC gammaAC /\
        Simulate (dlysBC,now,sigmaBC,gammaBC,thetaBC)(DFLAT (DPAR B C))
                 tauBC behBC /\
        QUIESCE dlysBC now tauBC gammaBC /\
        well_formed_DESIGN C /\
        well_formed_theta thetaC C /\
        EQ_SIGMA sigmaAC sigmaBC ports''/\
        EQ_TAU behA behB ports') ==>
       EQ_TAU behAC behBC prts)",
  CONV_TAC (TOP_DEPTH_CONV let_CONV) THEN tac1 THEN tac2 THENL
    [tac3a;tac3b;tac3c;tac3d;tac3e;tac3f;tac3g;tac3h;
     tac4a;tac4b;tac4c;tac4d;tac4e;tac4f;tac4g;tac4d])));;
```

APPENDIX B

A WORKED EXAMPLE

This appendix presents a HOL session comprising the deriviation of behaviours for a NAND gate and the proof of their equivalence that was described in Chapter 6.

B.1 Derivation for the Specification

```
============================================================================
              VHDL-HOL System (Version 1.0), built on 24/2/93
============================================================================

#let tm =
#    "let P = PROCESS {`A`,`B`}
#                     (INERT_SIG_ASSGN `C`(VNAND(Sig `A`)(Sig `B`))1)
#     and mu = (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},
#               {`A`},\t.({},{})) in
#         Simulate mu P (\t.{}) beh";;
tm =
"let P = PROCESS (`A`,`B`)
         BEGIN
            `C` <= (`A` NAND `B`) AFTER 1 NS;
         END PROCESS;
 in
 let mu =
     (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
 in
     mu |-- <P,(\t. {})> --Sim--> beh"
: term

#let expanded = ((TOP_DEPTH_CONV let_CONV) THENC
#                (ONCE_DEPTH_CONV get_delays_CS_CONV)) tm;;
expanded =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([1],now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{}))) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `C` <= (`A` NAND `B`) AFTER 1 NS;
    END PROCESS;,(\t. {})> --Sim--> beh
```

```
#let th1 = RIGHT_CONV_RULE ONCE_AROUND expanded;;
th1 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([1],(now + 1),{(`C`,~(a /\ b)),(`A`,a),(`B`,b)},
    {signl | ((signl = `C`) /\ ~(~(a /\ b) = c))},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `C` <= (`A` NAND `B`) AFTER 1 NS;
    END PROCESS;,INERT_POST `C` (~(a /\ b)) (\t. {}) now 1> --Sim--> beh

#let th2 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV finish_GAMMA) th1;;
th2 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (~(a /\ b) = c) =>
   ([1],(now + 1),{(`C`,~(a /\ b)),(`A`,a),(`B`,b)},{},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `C` <= (`A` NAND `B`) AFTER 1 NS;
    END PROCESS;,INERT_POST `C` (~(a /\ b)) (\t. {}) now 1> --Sim--> beh |
   ([1],(now + 1),{(`C`,~(a /\ b)),(`A`,a),(`B`,b)},{`C`},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `C` <= (`A` NAND `B`) AFTER 1 NS;
    END PROCESS;,INERT_POST `C` (~(a /\ b)) (\t. {}) now 1> --Sim--> beh
```

```
#let th3 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV ONCE_AROUND) th2;;
th3 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (~(a /\ b) = c) =>
   (beh = (\t. SND (ADD_TO_THETA {(`C`,~(a /\ b)),(`A`,a),(`B`,b)}
                        {} (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)}
                              {`A`} (\t'. ({},{})) now (now + 1))
                        (now + 1) (now + 1) t))) |
   ([1],(now + 1),{(`C`,~(a /\ b)),(`A`,a),(`B`,b)},{},
    ADD_TO_THETA {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} {`C`}
       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)} {`A`} (\t. ({},{}))
                     now (now + 1)) (now + 1) (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `C` <= (`A` NAND `B`) AFTER 1 NS;
    END PROCESS;,(\t. {})> --Sim--> beh

#let th4 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV ONCE_AROUND) th3 ;;
th4 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (~(a /\ b) = c) =>
   (beh = (\t. SND (ADD_TO_THETA {(`C`,~(a /\ b)),(`A`,a),(`B`,b)}
                        {} (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)}
                              {`A`} (\t'. ({},{})) now (now + 1))
                        (now + 1) (now + 1) t))) |
   (beh = (\t. SND (ADD_TO_THETA {(`C`,~(a /\ b)),(`A`,a),(`B`,b)}
                        {} (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c)}
                              {`A`} (\t'. ({},{})) now (now + 1))
                        (now + 1) (now + 1) t)))
```

```
#let th5 = REWRITE_RULE [ADD_TO_THETA;COND_ID] th4 ;;
th5 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. SND ((\t. (t >= now + 1) =>
                         ({},{(`C`,~(a /\ b)),(`A`,a),(`B`,b)}) |
                         (\t. (now >= now + 1) =>
                              ((t >= now) =>
                               ({`A`},{(`A`,a),(`B`,b),(`C`,c)}) |
                               ({},{})) |
                              ((now <= t /\ t >= now + 1) =>
                               ({`A`},{(`A`,a),(`B`,b),(`C`,c)}) |
                               ({},{}))) t) t)))

#let th6 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV if_arith_CONV) th5;;
th6 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. SND ((\t. (t >= now + 1) =>
                         ({},{(`C`,~(a /\ b)),(`A`,a),(`B`,b)}) |
                         (\t. (now <= t /\ t >= now + 1) =>
                              ({`A`},{(`A`,a),(`B`,b),(`C`,c)}) |
                              ({},{})) t) t)))

#let th7 = RIGHT_CONV_RULE (DEPTH_CONV simp_ATT_CONV) th6;;
th7 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. SND ((\t. (t >= now + 1) =>
                         ({},{(`C`,~(a /\ b)),(`A`,a),(`B`,b)}) |
                         (\t. (t >= now) =>
                              ({`A`},{(`A`,a),(`B`,b),(`C`,c)}) |
                              ({},{})) t) t)))
```

```
#let NAND_BEH = REWRITE_RULE [SND_SIMP] (BETA_RULE th7);;
NAND_BEH =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `C` <= (`A` NAND `B`) AFTER 1 NS;
           END PROCESS;
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c)},{`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. (t >= now + 1) =>
                {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} |
                ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})))
```

B.2 Derivation for the Implementation

```
#let tm =
#   "let P =
#    PAR (PROCESS {`A`,`B`}
#                 (INERT_SIG_ASSGN `TMP` (VAND (Sig `A`) (Sig `B`)) 1))
#        (PROCESS {`TMP`}(INERT_SIG_ASSGN `C` (VNOT (Sig `TMP`)) 0)) in
#    let mu = (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
#             {`A`},\t.({},{})) in
#         Simulate mu P (\t.{}) beh";;
tm =
"let P = PROCESS (`A`,`B`)
         BEGIN
            `TMP` <= (`A` AND `B`) AFTER 1 NS;
         END PROCESS;

         PROCESS (`TMP`)
         BEGIN
            `C` <= (NOT `TMP`) AFTER 0 NS;
         END PROCESS;
 in
 let mu =
     (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
      {`A`},(\t. ({},{})))
 in
     mu |-- <P,(\t. {})> --Sim--> beh"
: term
```

```
#let expanded = ((TOP_DEPTH_CONV let_CONV) THENC
#                (ONCE_DEPTH_CONV get_delays_CS_CONV)) tm;;
expanded =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([0;1],now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},{`A`},
    (\t. ({},{}))) |-- <PROCESS (`A`,`B`)
                        BEGIN
                           `TMP` <= (`A` AND `B`) AFTER 1 NS;
                        END PROCESS;

                        PROCESS (`TMP`)
                        BEGIN
                           `C` <= (NOT `TMP`) AFTER 0 NS;
                        END PROCESS;,(\t. {})> --Sim--> beh

#let th1 = RIGHT_CONV_RULE ONCE_AROUND expanded;;
th1 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([0;1],(now + 1),{(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)},
    {signl | ((signl = `TMP`) /\ ~(a /\ b = TMP))},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,INERT_POST `TMP` (a /\ b) (\t. {}) now 1> --Sim--> beh
```

```
#let th2 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV finish_GAMMA) th1;;
th2 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (a /\ b = TMP) =>
   ([0;1],(now + 1),{(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)},{},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
        `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
        `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,INERT_POST `TMP` (a /\ b) (\t. {}) now 1> --Sim--> beh |
   ([0;1],(now + 1),{(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)},{`TMP`},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,INERT_POST `TMP` (a /\ b) (\t. {}) now 1> --Sim--> beh

#let lemma =
#    PROVE("!x:bool. ~(x = ~x)",
#          GEN_TAC THEN BOOL_CASES_TAC "x:value" THEN REWRITE_TAC[]);;
lemma = |- !x. ~(x = ~x)
```

```
#let th3 = REWRITE_RULE [lemma] (SPEC "~(X/\y)" (GEN "TMP:value" th2));;
th3 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([0;1],(now + 1),{(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)},{`TMP`},
    ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} {`A`}
                 (\t. ({},{})) now (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,INERT_POST `TMP` (a /\ b) (\t. {}) now 1> --Sim--> beh
```

```
#let th4 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV ONCE_AROUND) th3;;
th4 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   ([0;1],(now + 1),{(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)},
    {signl | ((signl = `C`) /\ ~(~(a /\ b) = c))},
    ADD_TO_THETA {(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)} {`TMP`}
       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))}
           {`A`} (\t. ({},{})) now (now + 1)) (now + 1) (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,
    INERT_POST `C` (~(a /\ b)) (\t. {}) (now + 1) 0> --Sim--> beh
```

```
#let th5 = RIGHT_CONV_RULE (ONCE_DEPTH_CONV finish_GAMMA) th4;;
th5 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (~(a /\ b) = c) =>
   ([0;1],(now + 1),{(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)},{},
    ADD_TO_THETA {(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)} {`TMP`}
       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))}
           {`A`} (\t. ({},{})) now (now + 1)) (now + 1) (now + 1)) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,
    INERT_POST `C` (~(a /\ b)) (\t. {}) (now + 1) 0> --Sim--> beh |
   ([0;1],(now + 1),{(`C`,~(a/\b)),(`TMP`,(a/\b)),(`A`,a),(`B`,b)},{`C`},
    ADD_TO_THETA {(`TMP`,(a /\ b)),(`A`,a),(`B`,b),(`C`,c)} {`TMP`}
       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))}
           {`A`} (\t. ({},{})) now (now + 1)) (now + 1) (now + 1) |--
   <PROCESS (`A`,`B`)
    BEGIN
       `TMP` <= (`A` AND `B`) AFTER 1 NS;
    END PROCESS;

    PROCESS (`TMP`)
    BEGIN
       `C` <= (NOT `TMP`) AFTER 0 NS;
    END PROCESS;,
    INERT_POST `C` (~(a /\ b)) (\t. {}) (now + 1) 0> --Sim--> beh
```

```
#let th6 =
#   RIGHT_CONV_RULE (ONCE_DEPTH_CONV (ONCE_AROUND THENC ONCE_AROUND))
th5;;
th6 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (~(a /\ b) = c) =>
   (beh = (\t. SND (ADD_TO_THETA
                       {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),
                        (`B`,b)} {}
                       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),
                                      (`TMP`,~(a /\ b))} {`A`}
                           (\t'. ({},{})) now (now + 1)) (now + 1)
                       (now + 1) t))) |
   (beh = (\t. SND (ADD_TO_THETA
                       {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),
                        (`B`,b)} {}
                       (ADD_TO_THETA {(`A`,a),(`B`,b),(`C`,c),
                                      (`TMP`,~(a /\ b))} {`A`}
                           (\t'. ({},{})) now (now + 1)) (now + 1)
                       (now + 1) t)))
```

```
#let th7 = REWRITE_RULE [COND_ID;ADD_TO_THETA] th6;;
th7 =
|- let P = PROCESS (`A`,`B`)
           BEGIN
              `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
              `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. SND ((\t. (t >= now + 1) =>
                         ({},{(`C`,~(a /\ b)),(`TMP`,(a /\ b)),
                             (`A`,a),(`B`,b)}) |
                         (\t. (now >= now + 1) =>
                              ((t >= now) =>
                               ({`A`},{(`A`,a),(`B`,b),(`C`,c),
                                       (`TMP`,~(a /\ b))}) |
                               ({},{})) |
                              ((now <= t /\ t >= now + 1) =>
                               ({`A`},{(`A`,a),(`B`,b),(`C`,c),
                                       (`TMP`,~(a /\ b))}) |
                               ({},{}))) t) t)))
```

```
#let th8 = BETA_RULE (RIGHT_CONV_RULE ((ONCE_DEPTH_CONV if_arith_CONV)
#                                      THENC
#                                      (DEPTH_CONV simp_ATT_CONV)) th7);;
th8 =
|- let P PROCESS (`A`,`B`)
         BEGIN
             <= ( AND ) AFTER 1 NS;
         END PROCESS;

         PROCESS (`TMP`)
         BEGIN
             <= (NOT ) AFTER 0 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;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. SND ((t >= now + 1) =>
                    ({},{(`C`,~(a/\b)),(`TMP`,(a/\b)),(`A`,a),(`B`,b)}) |
                    ((t >= now) =>
                     ({`A`},{(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))}) |
                     ({},{})))))
```

```
#let NAND_IMP = REWRITE_RULE [SND_SIMP] th8;;
NAND_IMP =
|- let P = PROCESS (`A`,`B`)
           BEGIN
               `TMP` <= (`A` AND `B`) AFTER 1 NS;
           END PROCESS;

           PROCESS (`TMP`)
           BEGIN
               `C` <= (NOT `TMP`) AFTER 0 NS;
           END PROCESS;
   in
   let mu =
       (get_delays_CS P,now,{(`A`,a),(`B`,b),(`C`,c),(`TMP`,tmp)},
        {`A`},(\t. ({},{})))
   in
       mu |-- <P,(\t. {})> --Sim--> beh =
   (beh = (\t. (t >= now + 1) =>
               {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)} |
               ((t >= now) =>
                {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} |
                {})))
```

B.3 Equivalence Proof

```
#let theta' = rhs (rhs (concl (SPEC_ALL NAND_BEH)))
#and theta'' = rhs (rhs (concl (SPEC_ALL NAND_IMP)))
#and conj_to_cond = theorem `equiv-thms` `conj_to_cond`
#and EQ_TAU = definition `Femto-config` `EQ_TAU`;;
theta' =
"\t. (t >= now + 1) =>
     {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} |
     ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})"
: term
theta'' =
"\t. (t >= now + 1) =>
     {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)} |
     ((t >= now) => {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} | {})"
: term
conj_to_cond =
|- !signl A B P. (signl = A) /\ (signl = B) /\ P =
                    (A = B) => ((signl = A) /\ P) | F
EQ_TAU =
|- !tau' tau'' sigs.
      EQ_TAU tau' tau'' sigs =
         (!t. let sigma' = tau' t
              and sigma'' = tau'' t
              in ({sig,val | sig IN sigs /\ (sig,val) IN sigma'} =
                    {sig,val | sig IN sigs /\ (sig,val) IN sigma''}))

#g "EQ_TAU ^theta' ^theta'' {`A`,`B`,`C`}";;
"EQ_TAU (\t. (t >= now + 1) =>
             {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} |
             ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {}))
    (\t. (t >= now + 1) =>
         {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)} |
         ((t >= now) =>
          {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} |
          {})) {`A`,`B`,`C`}"

() : void
```

```
#e (REWRITE_TAC [EQ_TAU] THEN
#   GEN_TAC THEN BETA_TAC THEN COND_CASES_TAC THEN REWRITE_TAC [] THEN
#   CONV_TAC (ONCE_DEPTH_CONV let_CONV) THEN
#   REWRITE_TAC [IN_INSERT;NOT_IN_EMPTY;PAIR_EQ]);;
OK..
2 subgoals
"{sig,val |
 ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
 (sig,val) IN ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})} =
    {sig,val |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    (sig,val) IN ((t >= now) =>
                  {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} |
                  {})}"
    [ "~(t >= now + 1)" ]

"{sig,val | ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
            ((sig = `C`) /\ (val = ~(a /\ b)) \/
             (sig = `A`) /\ (val = a) \/
             (sig = `B`) /\ (val = b))} =
    {sig,val | ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
               ((sig = `C`) /\ (val = ~(a /\ b)) \/
                (sig = `TMP`) /\ (val = a /\ b) \/
                (sig = `A`) /\ (val = a) \/
                (sig = `B`) /\ (val = b))}"
    [ "t >= now + 1" ]

() : void

#e (REWRITE_TAC [RIGHT_AND_OVER_OR] THEN
#   REWRITE_TAC [LEFT_AND_OVER_OR;conj_to_cond] THEN
#   CONV_TAC (ONCE_DEPTH_CONV string_EQ_CONV) THEN
#   CONV_TAC (ONCE_DEPTH_CONV COND_CONV) THEN
#   REWRITE_TAC[]);;
OK..
goal proved
|- {(sig,val) |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    ((sig = `C`) /\ (val = ~(a /\ b)) \/
     (sig = `A`) /\ (val = a) \/
     (sig = `B`) /\ (val = b))} =
   {(sig,val) |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    ((sig = `C`) /\ (val = ~(a /\ b)) \/
     (sig = `TMP`) /\ (val = a /\ b) \/
     (sig = `A`) /\ (val = a) \/
     (sig = `B`) /\ (val = b))}

Previous subproof:
"{sig,val |
 ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
 (sig,val) IN ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})} =
    {sig,val |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    (sig,val) IN ((t >= now) =>
                  {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} |
                  {})}"
    [ "~(t >= now + 1)" ]

() : void
```

```
#e (COND_CASES_TAC THEN REWRITE_TAC [IN_INSERT;NOT_IN_EMPTy;PAIR_EQ] THEN
#   REWRITE_TAC [RIGHT_AND_OVER_OR] THEN
#   REWRITE_TAC [LEFT_AND_OVER_OR;conj_to_cond] THEN
#   CONV_TAC (ONCE_DEPTH_CONV string_EQ_CONV) THEN
#   CONV_TAC (ONCE_DEPTH_CONV COND_CONV) THEN
#   REWRITE_TAC[]);;
OK..
goal proved
|- {(sig,val) |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    (sig,val) IN ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})} =
   {(sig,val) |
    ((sig = `A`) \/ (sig = `B`) \/ (sig = `C`)) /\
    (sig,val) IN
    ((t >= now) => {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} | {})}
|- EQ_TAU
   (\t.
     ((t >= now + 1) =>
      {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} |
      ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {})))
   (\t.
     ((t >= now + 1) =>
      {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)} |
      ((t >= now) => {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} | {})))
   {`A`,`B`,`C`}

Previous subproof:
goal proved
() : void
```

```
#let NAND_THM = GEN_ALL (top_thm());;
NAND_THM =
|- !now a b c.
     EQ_TAU (\t. (t >= now + 1) =>
                 {(`C`,~(a /\ b)),(`A`,a),(`B`,b)} |
                 ((t >= now) => {(`A`,a),(`B`,b),(`C`,c)} | {}))
         (\t. (t >= now + 1) =>
              {(`C`,~(a /\ b)),(`TMP`,(a /\ b)),(`A`,a),(`B`,b)} |
              ((t >= now) =>
               {(`A`,a),(`B`,b),(`C`,c),(`TMP`,~(a /\ b))} |
               {})) {`A`,`B`,`C`}
```

```
#quit();;
```