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 $\delta$-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.

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

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

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.*

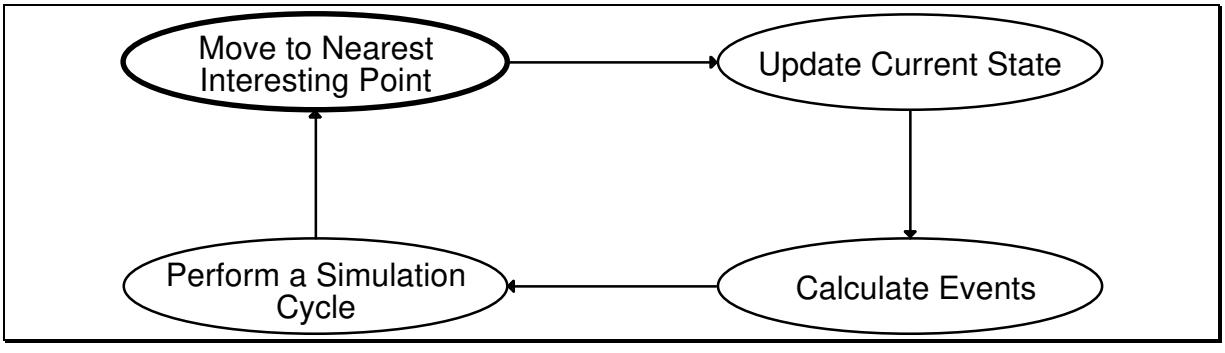


Figure 1.1: The Simulation Loop

**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

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:

# 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