# Table Of Contents

# An Application of a Synchronous/ Reactive Semantics to the VHDL Language


**Wendell Baker**
**U.C. Berkeley**



**December 18, 1992**

# Abstract

The goal of this project was to define and demonstrate a subset of the VHDL language [IEEE87] which is *consistent with the simulation semantics* and yet has an *interpretation as a specification*. The subset shown in this work incorporates as much of the language as is possible without compromising its interpretation as a specification of system behavior. There is no other subset of the VHDL language which can incorporate more of the semantics of the language, yet retain an interpretation as a specification.

To show why this is the larges subset of the VHDL language which has this property, a review of some of the many uses of VHDL is given along with an explanation of why the interpretation of the full VHDL language as a specification within this context is problematic at best. A definition of what it means to interpret an executable description as a specification is given in the form of an example of an existing class of languages, *the reactive languages,* which have this property: being executable yet having an interpretation as a specification. The question then is whether there is a useful subset of the VHDL language that can be shown to be reactive. In fact, within the wide range of uses of VHDL, there is a class of uses that can be interpreted as a reactive specification. The identification of this class of uses, and the restrictions on the language structure that accompany it form the basis of the subset definition.

Reactive systems, or equivalently synchronous systems, have the property that they respond to events from the environment in which they reside; nothing of interest outside these events and their responses occurs in these systems. The formulation of the definition of reactive systems in terms of events allows for a succinct description of their behaviors as *regular expressions.* Hence the implementation of a reactive system is naturally in terms of one of the many finite automata which recognizes the regular language that defines the system's behavior. The goal of the subset definition then is to identify the reactive portion of the VHDL language, independent of its language structures, so that the behavior of programs written in the subset may be interpreted as regular expressions, and implemented in terms of (communicating) finite automata.

The Synchronous VHDL subset presented here is derived through a restriction on the abstract simulator that defines the meaning of a VHDL program. The focus here is on restricting the simulator behavior and using that restriction to drive the definition of the VHDL subset, instead of the other way around as has been traditional in the definition of other VHDL language subsets. The result of this work is a description of the subset of the VHDL language that uses only finite-automata-like behavior; it thus has well-defined implementation as a network of communicating finite state machines.

In addition to the definition of Synchronous VHDL, a goal of this project was to demonstrate the subset in use. Thus, a major portion of this project was the implementation of a compiler front-end which is used to analyze VHDL source text, converting from the syntax of the language into a form suitable for compilation into a reactive language.

# Acknowledgments

I am indebted to my research advisor, Professor Richard Newton, for his encouragement and support throughout the course of this work. I am also grateful to Professors Susan Graham and Bob Brayton and Kathy Yelick for the interest they have shown in this project.

This work is supported by the Semiconductor Research Corporation under contract 91-DC-008. Their support is gratefully acknowledged. Additional support was provided by Digital Equipment Corporation through their support of the CAD Group's DECstation-based computing environment.

I thank Cygnus Support Inc. for their work over the past two years maintaining the GNU C++ 2.0 compiler and debugger and porting that system to the DECstation. I also thank the Microelectronics and Computer Technology Corporation (MCC) for the use of their VHDL simulator during the course of this work. I also thank Gérard Berry of INRIA and Hervé Touati of DEC PRL for the use of the Esterel V.3 compiler system.

All of the members of the Berkeley CAD Group, past and present, have helped me in various ways during my tenure here; in particular Gregg Whitcomb, Paul Gutwin, and Dr. Rick McGeer. I thank Jim Rowson and David Ku of Redwood Design Automation for the many discussions that we have had.

Above all I would like to thank my wife, Mary Baker for her patience and support during this project.

# Chapter 1

# Introduction

The 1980s saw the introduction of VHDL as a standard hardware description language for producing simulation models of existing hardware components. The original purpose of VHDL was to provide a standard language in which the behavior of electronic devices procured under U.S. Department of Defense programs could be encoded. The language designer's goal was allow the seller to transmit a description of the part to the purchaser along with the part so that a definite idea of its behavior would be imparted. In addition to describing behavior, the description was to act as a specification for the part so that another one could be built in the future; to build a second copy, one must know what the first copy did.

Coupled with other developments such as an increased use of logic-level synthesis, larger designs being attempted, and more complicated designs which require better testing, the use of VHDL changed from being purely a *descriptive* language to one which is now being applied in far more varied ways. These areas range across testing, specification, netlist representation and even into device-level simulation and analog simulation! For this project, the interest in VHDL is solely in the area of *specification*; system-level specification for use in high-level and sequential-level synthesis.

It would be tempting to ignore these other areas to concentrate on VHDL as specification but unfortunately these other application areas of VHDL impinge upon its use as a specification language. Both the simulator origins of the language and the subsequent reinterpretation of it to apply it to new uses impose difficulties for the sound definition of VHDL as a specification for computing hardware. A sound definition of specification is a precondition for defining a synthesis path from VHDL to sequential-level or logic-level synthesis.[1]

**The Theses of This Work**

The thesis being investigated in this project is whether or not it is possible to define a subset of VHDL which is both consistent with the simulation semantics described in its 1987 definition [IEEE87] and which also has an interpretation as a specification of hardware. The interpretation as a specification is given by the restriction of the VHDL simulator behaviors to those behaviors allowable under the synchronous system hypothesis. The consistency of the subset with the full simulation semantics is guaranteed by defining the language subset based on a restriction of the simulator model and deriving the effects on the language.

The validity of this thesis implies that it is possible to define the meaning of a VHDL program in this subset in a rigorous way; the meaning is as a specification for a network of communicating finite state machines. The computations of this network, its states and its state transitions, will be exactly the same as those of any correct VHDL simulator. Thus this subset of VHDL can be used as an input for both sequential optimization and synthesis and also it can be checked for correctness by automata-theoretic or temporal-logic based verification tools such as such as COSPAN [HK90] and [CJLM91].

A schematic of the desired VHDL subset is shown in Figure 1. Depicted there, as a subset of all possible VHDL programs, is the synchronous subset. This subset is defined, not by the set of syntactic constructs allowable in the subset, but by the behavior of the simulator on those programs. The subset is defined by the behavior of the simulator, which implies restrictions on the syntax, not the other way around as has been traditionally done. Figure 2 and Figure 3 illustrate this distinction. A major point underpinning the synchronous subset is that this subset, while defined through the dynamic pro-

1. Exactly the same can be said for Cadence Design System's Verilog language [Ver91], save for its history and origins. As the capabilities of VHDL are a strict superset of those in Verilog, the work here applies equally to well to each language; Verilog will not be mentioned further.

cess of simulation, can actually be identified in a static analysis of program text; it is possible to determine whether a VHDL program obeys synchronous semantics at compile time.

**Figure 1** A Schematic of the Synchronous VHDL Subset

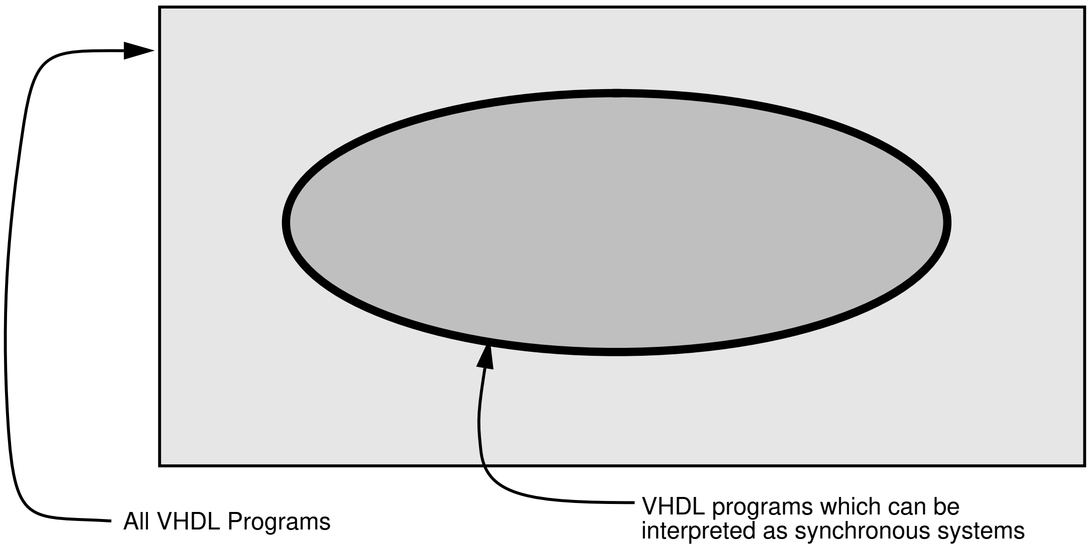
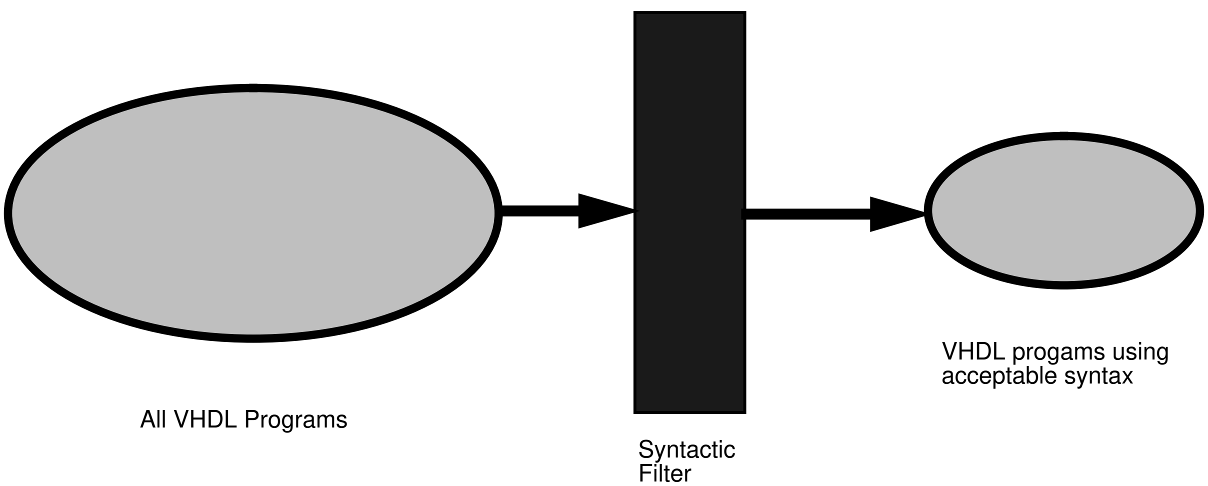


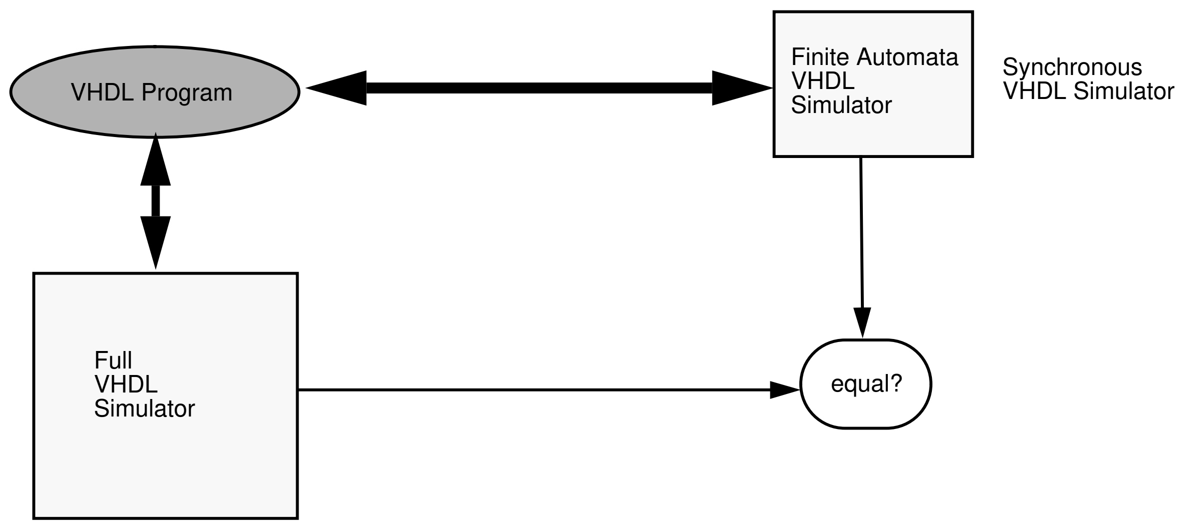

**Figure 2** Traditional VHDL Subset Definition




**Long- and Short-Range Goals**

Ultimately the goal of this work is to define the synthesis semantics of VHDL in a rigorous way. That being done, the effect of such a rigorous definition will show up elsewhere in the chain of tools that consume design descriptions. In addition to synthesis applications there are also verification aspects to design which are current areas of research. Providing a standard front-end language to those tools is also a goal of this work. Further, though, a consistent semantics of VHDL as finite automata will have affects in the design of simulators for VHDL; surely a simulator which used a network of communicating finite state machines would be simpler to partition and parallelize than a simulator which was fashioned as prescribed in the definition of the VHDL language [IEEE87][2] (*c.f.* [Vel90]).

**Figure 3** Synchronous VHDL Subset Definition




Despite these far-reaching goals, the focus here is on a far more tractable problem: using the proposed synchronous semantics of VHDL to define a translation path from VHDL to an executable format (*e.g.* to C code). This translation path defines a simple simulator, thus forming a proof of concept both for the soundness of the synchronous subset of VHDL and for the idea of using reactive compilation to define the specification aspect of this subset of VHDL.

Attempting to produce a system that performs both the identification of the synchronous subset of VHDL and the translation to the final executable form required too great an effort for a simple proof of concept study such as this one. Instead, the approach taken for this work was to break that job up into two parts: the construction of a compiler front-end for the VHDL language, and the use of that front-end to convert VHDL source text into a form which can be conveniently compiled by a reactive language compiler. The reactive language used is Esterel [CIS88], and thus this report describes the design of a translation path from VHDL into Esterel and thus to a final executable form in C[3]. The main body of the report describes the design and definition of Synchronous VHDL, leaving to the appendices the documentation of the salient facts about the C++ compiler front-end for VHDL that was developed to support this work.

**Overview**

The goal of this work is to demonstrate the feasibility of constructing the translation path. The description here is simply that of a proof of concept study. The simulator described here is in no sense yet a product, though the hope is that the technique described here for using reactive compilation to implement certain classes of simulation models is powerful enough that one day it may lead to a product-quality approach.

The ideas described in this report have been given form in the development of a toolkit of compiler algorithms, which have been written in C++. Because the construction of a complete VHDL simulator is a rather gargantuan task, it has not been possible to produce a full VHDL simulator implementation within the scope of this project. Instead, what this work represents is a study of the issues involved in developing such a simulator. Thus only those aspects of the simulator which are directly related to the use of the reactive semantics as an implementation method for simulator models has been imple-

2. The description of the required simulator event processing loop, as found in Chapter 12 of [IEEE87], is reproduced in Appendix B.

3. Again, it must be stressed that the use of the Esterel compiler in this work is merely a convenience; a stand-alone reactive compilation algorithm would allow for its replacement.

mented. A good portion of the implementation which is not germane to the use of reactive compilation has been forgone. The C++ compiler toolkit which was implemented for this study is described in the presentations of Chapter 4 and Chapter 5. The hope is that with this work as the basis, a more extensive investigation of the use of reactive compilation techniques can be attempted. Based on the investigation done here, those techniques hold the promise of enabling improvements in a number of application areas ranging from embedded software applications to discrete-event simulators as is the case here.

There are a number of problems involved in defining the translation path proposed here over and above the simple development of a relationship between a VHDL subset and Esterel. The first of these is to motivate why a sound definition of a translation path from VHDL to *any* other representation of computation is problematic; this is the subject of Chapter 2.[4] The properties of reactive/synchronous languages with emphasis on the Esterel language is described in Chapter 3. That chapter provides some background on Esterel's semantics - both the interpretation semantics and the compilation semantics. The synchronous subset of VHDL which is motivated by the translation of VHDL processes into Esterel is presented in Chapter 4. A simple VHDL simulator for the Synchronous VHDL subset which uses the reactive language compilation capability of the Esterel compiler as a code generator is presented in Chapter 5. Finally, a review of the conclusions which can be drawn from the project described in this report is given in Chapter 6.

4. For the purposes of this report it is assumed that the reader is familiar with VHDL. The overview presented in the next chapter is not a general presentation of the language, rather it focuses on the issues involved in interpreting VHDL models as synchronous specifications. Readers unfamiliar with VHDL may wish to see [LSU89], [IEEE87], [Coe89] or [ALG+91].

# Chapter 2

# The VHDL Problem

Expanding the interpretation of VHDL descriptions away from simulation is problematic because simulation models are *descriptive* and the reduction relationship from reality to the modeling domain is held largely in the mind of the observer. The differences between simulation models and specifications is shown schematically in Figure 4 and Figure 5. The purpose of this chapter is to motivate why attempting to find a sound definition of a VHDL subset that can be interpreted as a specification is hard.

**Figure 4** A Simulation Model is a Reduction of Reality

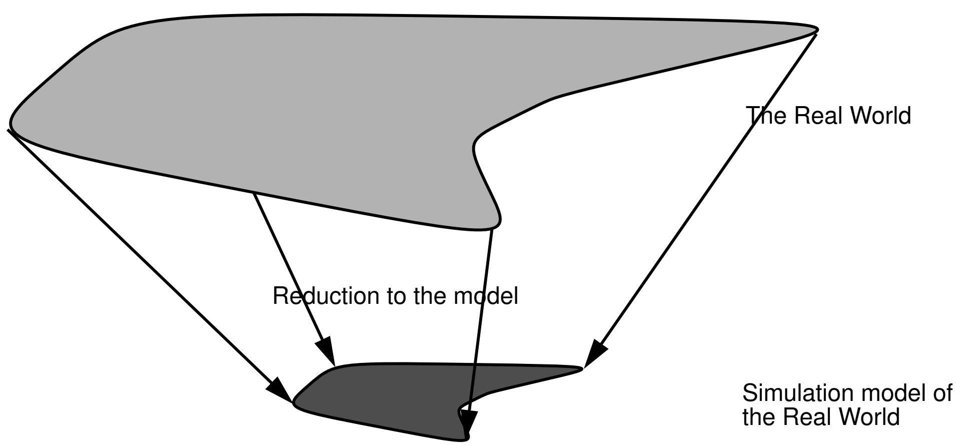
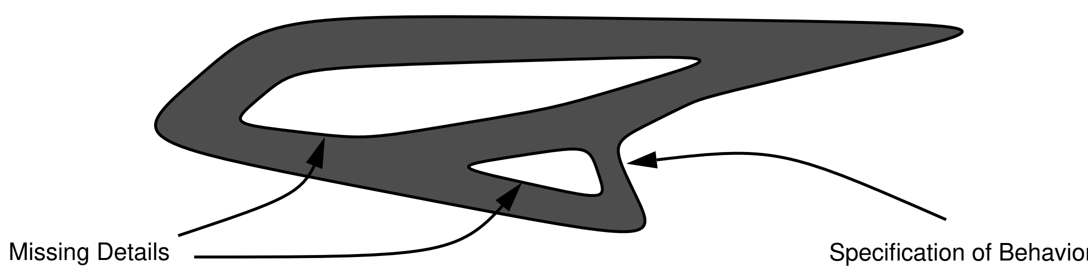



**Figure 5** A Specification Lacks Details




The difficulties in subset definition fall into roughly four categories which can be summarized in the following way:

- A *syntax-directed* policy of use for the language is insufficient; this is addressed in Section 2.1.
- The *inexact focus* of the level of VHDL description makes interpretation at any *specific level* for specification problematic. This issue is addressed in Section 2.2.

- The *asynchronous nature of parallelism* in VHDL models is difficult to work with in simulation models and extremely difficult to interpret as a specification. This and the simplification to *synchronous parallelism* being used here are described further in Section 2.3.
- The VHDL simulator loop and the sequential code execution rules introduce an *interpreter bias* into the use of VHDL for specification. Interpreter bias, the problems that it causes and the use of the synchronous system paradigm as an attempt to solve them are described in Section 2.4.

Qualifying the difficulty of the problem is important so that the reader can better understand that all of the mechanism and formality introduced in the later sections are necessary to achieve these ends. This proposal is for a semantics-driven policy which defines a VHDL subset that achieves the end of an interpretation for specification that covers as much of the VHDL language as possible while still preserving the observable simulation behavior of the VHDL models.

This interpretation is not necessarily the easiest to implement nor is it the simplest subset policy ever devised[5]. The claim is however, that this policy is the most *complete* in the sense that it allows for as much of the VHDL language as possible to be interpreted as a specification of behavior. This is a very difficult claim to prove in the traditional formal sense of lemmas and theorems, so instead a convincing argument of this completeness is presented. The purpose of this section is to review the problems which must be solved to define the notion of VHDL as a specification. Thus, this section provides the basis for that convincing argument - that the synchronous interpretation of VHDL as described in Chapter 4 is both necessary and sufficient for use as a specification of behavior.

## 2.1 Policies of Use

One of the first objections which arises in a semantics-modification proposal which is as invasive as is this one is whether or not it could have been achieved with fewer restrictions or whether a result which was *good enough* could have been achieved with a simple but mathematically inelegant set of restrictions on the language. In an attempt to head off those arguments, this section is devoted to a description of the options for a VHDL policy in the traditional style (*c.f.* Figure 2).

The argument here is that each of these possibilities save for one, the synchronous subset which is based on the behavior of the simulator directly, is insufficient or incomplete in some important way. It is important to remember throughout, that the purpose here is to investigate the possible ways that a most-general subset for specification might be derived. The insufficiency of any one of the following attempts on the problem does not indicate that it *cannot* be used; certainly there are a number of synthesis systems which use each one. The point is that there is a more general and more elegant solution available through the use of the Synchronous VHDL subset.

**Using Identifiable Sublanguage**

One of the more common restrictions to impose is that the VHDL description be restricted to use only an identifiable sublanguage. A common one is the dataflow portion of VHDL; another is a restriction to structural descriptions from a library of known parts.

Examples of dataflow and structural subsets of VHDL are shown in Figure 6 and Figure 7 respectively. Both of these examples are fairly straightforward as they are combinational examples; they contain no feedback and no internal state.[6]

5. A quick look through the relevant literature on VHDL-based high-level synthesis systems confirms this [CBH+91] [Che91] [HCD90] [LiGa89] [RoVe89] [UdVe89].

6. The processes in VHDL do have internal state in the signal drivers; that state is being ignored for these examples.

When the examples use data types which are more complex than **Bit**[7] or when the flow dependencies in the models are not acyclic, then the interpretation of the meanings of the architectures as specifications becomes unclear.

**Figure 6** An Example of the Dataflow Sublanguage of VHDL

```
entity Some_Function is
   port(A, B, C, D: in Bit; O1, O2: out Bit);
end Some_Function;

architecture Dataflow of Some_Function is
begin
   O1 <= (A and B) or (C and D);
   O2 <= (A nor B) nand (C nor D);
end Dataflow;
```

**Figure 7** An Example of the Structural Sublanguage of VHDL

```
entity Some_Other_Function is
   port(A, B, C, D: in Bit; O1, O2: out Bit);
end Some_Other_Function;

use Lib.Specification_Parts.all;
architecture Structural of Some_Function is
   signal Tmp1, Tmp2, Tmp3, Tmp4: Bit;
begin
   U1: and2
      port map(I1 => A, I2 => B, O => Tmp1);
   U2: and2
      port map(I1 => C, I2 => D, O => Tmp2);
   U3: or2
      port map(I1 => Tmp1, I2 => Tmp2, O => O1);
   U4: nor2
      port map(I1 => A, I2 => B, O => Tmp3);
   U5: nand2
      port map(I1 => C, I2 => D, O => Tmp4);
   U6: nand2
      port map(I1 => Tmp1, I2 => Tmp2, O => O2);
end Structural;
```

Somehow it would be more satisfying if the interpretations of Figure 6 and Figure 7 as specifications were derived from the behavior of the simulator rather than an intuitive understanding of what is written in English. Such a specification would be more sound in general and could also trivially take into account the redefinition of the **and**, **or**, **nand** and **or**

7. The **Bit** data type is not predefined *in* VHDL; rather, it is a data type which is required to *be available* in the standard library by the declaration: **type Bit is ('0', '1');**

operators. It is rare for a user to redefine these operators on the **Bit** type, but quite common to do it for other data types. The mechanics of the treatment of a VHDL specification subset should be robust under such redefinitions.

**Using Structural Patterns**

Another treatment of VHDL for specification involves using a known structure of the syntax to imply a specification. For example, in Figure 8 a pattern of usage is indicated which will ultimately be interpreted as a clocked latch by the tool that interprets the specification - the synthesis system.[8]

**Figure 8** An Example of a Pattern of Usage

```
entity A_Latched_Function is
   port(A, B, C, D: in Bit;
         CLK, RST: in Boolean;
         O: out Bit);
end A_Latched_Function;

architecture Pattern_Match of Some_Function is
   signal Tmp_O: Bit bus;
begin
   Reset:
   block(RST = FALSE)
   begin
      Tmp_O <= guarded '0';
   end block Reset;

   Func:
   block(not Clock'stable and Clock = True and RST)
   begin
      Tmp_O <= guarded (A and B) or (C and D);
   end block Func;

   O <= Tmp_O;
end Pattern_Match;
```

It may take more than a moment of reflection to determine the behavior of the two guarded blocks operating in tandem driving a local signal of kind **bus** which is ultimately fed to the output. In fact, the simulator does quite a bit of work to simulate these three constructs, so it is not surprising that the pattern is hard to recognize as a clocked latch with a reset without some *gedanken* experiments on the abstract VHDL simulator.

This style of description is convenient, however complicated it may seem, for the tool writers for the “synthesis system” need only recognize the three patterns: two guarded blocks driving the same bus signal and a third concurrent assignment feeding the result out. This pattern can be used as a specification for a query into “the library” in which there will be just such a clocked latch. The right-hand side of the **Func** block is taken as the specification of the logic function to be computed and stored in the clocked latch.

The important point of this very structured use of VHDL is that it represents an extension to the language. The patterns which are recognized by the synthesis tools, though defined completely within the syntax of the language standard, repre-

8. This style of specification in VHDL is in daily use at a major electronics company.

sent idioms which are imbued with meanings defined outside of the language standard. Thus the use of a style or policy of use VHDL ultimately represents a change in the language to suit a specific tool implementation or design library. As this style of specification is so implementation-specific and depends so much on the modeler's interpretation of the meaning of the cooperation between the two blocks (**Reset** and **Func** in the figure) it will not be considered further here. It is introduced here because this style of specification with VHDL is in fact used and is considered acceptable *as a specification* to those who use it - it allows them to get designs done.

**"... Well, Then We Won't Support That"**

Finally, there is the common method building a VHDL policy for synthesis specification which is completely *ad hoc*. Essentially the procedure used is to start with a small subset of VHDL, such as the dataflow subset described above, and grow by iteratively adding one more construct to the subset until it becomes difficult or impossible to support the new addition. On the positive side, this method does tend to keep constructs out of the resulting subset which are obviously incongruous: pointers and files and the like.

The problem with this method, as with the other syntax-directed policies is that there is no unifying principle with which to accept or reject constructs from the subset language. The process of selection tends to devolve into attempting to map a VHDL statement's perceived meaning onto the target tool and if that cannot be done, then *"... well, we won't support that.*"

**Restricting the Simulator**

Developing a policy for VHDL that can be interpreted as a specification which is based on restricting the syntax that will be supported sounds easy and attractive. Its appeal lies in the ability to use the grammatical structure of the language, as defined in Appendix A of [IEEE87], as an aid to the categorization and elimination of statements or constructs which are considered too difficult to use. This unfortunately has serious effects on both the breadth of the language which can be supported and the interpretation of that language after the allowed subset is identified.

After the allowed subset is identified from a review of the language definition there is no guarantee about *what* the review process generated in terms of a subset (how much of the language is supported). Further, there is no set of rules for determining the meaning of the constructs in the subset after selection and ensuring that the specification meaning is the same as the simulation meaning. This problem arises from the procedure used to accept or deny support for the various statements; it is based on what is known to be supported in some target tool.

What is being searched for in an interpretation of VHDL as a specification is exactly backwards from the syntax-directed case: there is no tool yet, so the goal is to find a subset of VHDL that specifies computing structures *such that* a tool can be built which will derive an implementation for that specification. The presentation of that subset and the identification of a tool, a VHDL-to-Esterel translator and its use as a simulator generator is the subject of Chapter 4 and Chapter 5.

## 2.2 The Breadth of VHDL Description

VHDL is a large language as many have noted. It is surprising just how large it is however when one considers not only the languages structures involved, but also the various interpretations or policies with which its users mold the language to suit various needs. These needs are outlined in the following sections; they range from describing test benches to black box models to pure netlists and on into more exotic areas where VHDL was never intended to be applied.

In this section is exhibited the breadth of use to which VHDL can be put in order to frame the uses to which the specification interpretation of VHDL will be made. These areas are not the areas to which VHDL is *best* put to use necessarily, but they are areas where it can be demonstrated that VHDL *is currently being* put to use. It is important to understand this breadth in order to provide the proper context for the presentation of the very constrained usages required of the synchronous subset of VHDL. The desire is to motivate here is that the range of uses of VHDL are driven by needs for design description and management which are outside the scope of this project. They are driven by a need for a language-driven design framework which uses a common language across all levels of design ranging from high-level specification to testing to low-level modeling of semiconductor behaviors. This goal may or may not be a desirable one to achieve. It is however being actively pursued and as will be shown in the following sections, it has a large effect on how well and how much of VHDL can be interpreted as a specification.

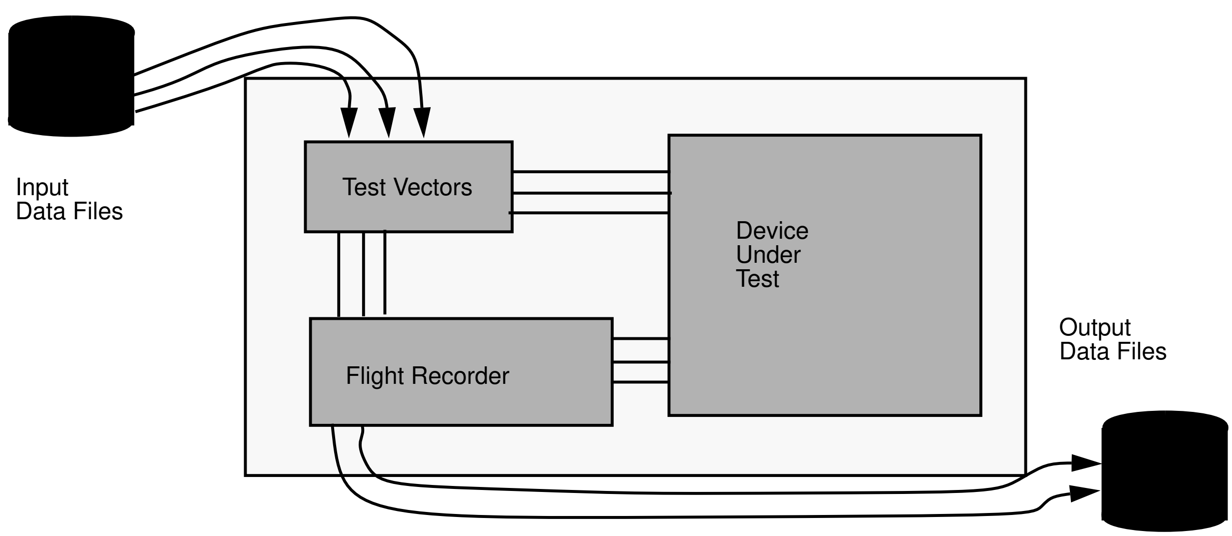
The important point of this section is that from a specification viewpoint, there is no way to tell these various uses apart: a test bench in VHDL looks remarkably like a switch-level model of a CMOS circuit - from the perspective of an automated tool which consumes VHDL specifications. Typically it is only the identifier names used in the models which distinguish them (*i.e.* names like **test_bench** or **and3** or **nfet**). An automated tool cannot be expected to extract and comprehend this intended use. It is thus critical that the user be able to identify these uses and disallow their submission as specifications in the first place; this is a "well *we just won't let users do that"* situation. A taxonomy of VHDL is provided in this sectoin so that when users are told that they can't use a certain class of VHDL as a specification, then they can have a name for that class and a reason for the claimed inappropriateness as a specification.

### Test Bench Construction

The test bench is where the full power of the VHDL language is most useful. In most design situations, it is up to the designer of the model to provide a rig to test out the device model. Instead of letting that task be provided in an implementation-dependent manner, the VHDL language designers have allowed VHDL itself to serve that function. In fact, there are standards for test benches written in VHDL such as the WAVES standard [WAV90] and the BSDL on-board test structures [PO90]. These test bench standards describe test structures ranging among the vector formats in files, the procedures to apply test patterns, files that they will read and write and the in-device test structures required for the methodology.

The test bench is simply a VHDL entity and architecture which instantiates the *device under test* (DUT) and applies test vectors and records the results. A schematic of this is shown in Figure 9. Some simulators provide this facility directly for simple models; however for complex models like processors or controllers, it is typically advisable to build a flight recorder and test vector application unit. This way the full power of VHDL can be used to filter the events being input to and emanating from the DUT and record only the interesting ones.

**Figure 9** A Schematic of a Test Bench



A test bench for the traffic light controller is shown in Figure 10.[9] Although this test bench does not contain all of the parts which are shown in Figure 9, it does contain the essential aspects which are: the test bench is autonomous, it does not have any ports connecting *it* to anything, the test bench instantiates some component, and it applies test vectors to that component and records the results of those tests. In Chapter 3 these conditions are shown to imply that a test bench is not a reactive component and cannot not have an interpretation as a specification.

**Figure 10** A Test Bench for the Traffic Light Controller

```
entity TLC_Test is
end TLC_Test;

use Work.Traffic_Package.all;
architecture Test of TLC_Test is
   signal Car_On_Farm_Road: Boolean := FALSE;
   signal Highway: Color := Green;
   signal Farmroad: Color := Red;
   component TLC
      generic(Long_Time: Time; Short_Time: Time);
      port(Car_On_Farmroad: Boolean in Boolean;
          Highway: Light: out Color;
          Farmroad: Light: out Color);
   end component;
begin
   Controller: TLC
      generic map (5 ns, 2 ns)
      port map(Car_On_Farm_Road, Highway, Farmroad);

   Car_on_Farm_Road <= FALSE,
                         TRUE after 1 ns,
                         FALSE after 3 ns,
                         TRUE after 10 ns,
                         FALSE after 20 ns;
end Test;

use Work.all;
configuration spec of TLC_TEST is
   for Test
      for Controller: TLC
         use entity Work.Traffic_Light_Controller(Specification);
      end for;
   end for;
end spec;
```

### Black-Box Modeling

The essential idea behind a black box model is that it doesn't matter how the part is described so long as it simulates the correct responses. In fact, there is a retail market for models of processor blocks, controllers and other large blocks. These

9. This example is taken from [LSU89], pages 186-188.

models are used *in* other models such as models of boards or backplanes to ensure that the complete design works correctly with the part.

What is considered to be behavioral or black-box modeling is shown in Figure 11.[10] The idea is that all which is important about an entity is its simulation behavior *as perceived by the outside world*. So, in this respect it is irrelevant whether the description in the architecture is given by a complete cover for the boolean function as is the case in Figure 11 or by a chain of interrelated processes.

**Figure 11** A Black Box Description

```
entity Decoder is
   port(Enable: in Bit;
         Sel: Bit_Vector(2 downto 0);
         Dout: out Bit_Vector(7 downto 0));
   constant Delay: Time := 5 ns;
end Decoder;

architecture Selected of Decoder is
   type vec8x8 is array(integer range 0 to 7) of bit_vector(0 to 7);
   constant one_hot: vec8x8 := (
      "00000001", "00000010", "00000100", "00001000",
      "00010000", "00100000", "01000000", "10000000");
   function cvt(bv: in bit_vector(2 downto 0)) return integer is
      variable sum: integer := 0;
   begin
      if bv(0) = '1' then
         sum := sum + 1;
      end if;
      if bv(1) = '1' then
         sum := sum + 2;
      end if;
      if bv(2) = '1' then
         sum := sum + 4;
      end if;
      return sum;
   end cvt;
begin

   with Sel select
      Dout <= one_hot(cvt(Sel));

end Selected;
```

Some synthesis systems accept this type of description as a specification because it is interpreted as specifying combinational logic only. That interpretation is due to the use of the dataflow **with** construct in the main body of the architecture.

10. This example is taken from [LSU89], page 100.

The **with** construct need not indicate combinational logic, especially in the presence of feedback or if the **after** clauses are not all the same.

Typically though black box models are written with simulation efficiency in mind as they are to be used *inside* of other models as leaf nodes. As such, black box models are typically not considered as specifications. They could be considered as such though if the consuming tool were powerful enough to disambiguate the specification of behavior from the typically hyper-efficient coding of function at the behavioral level of VHDL.

**Behavioral Specification**

This is the level of description that is typically considered the most interesting use of VHDL for synthesis or verification. The designs are written to be clear and expository, as opposed to being efficiently executable in a compiled-code simulator. For example, contrast the two styles of description for the same decoder in Figure 11 and Figure 12; one is clearly going to be faster to simulate in a compiled-code environment than the other.

**Figure 12** A Specification of the Decoder

```
entity Decoder is
   port(Enable: in Bit;
      Sel: Bit_Vector(2 downto 0);
      Dout: out Bit_Vector(7 downto 0));
end Decoder;

architecture Selected of Decoder is
begin
   with Sel select
      Dout <= "00000001" when "000",
            "00000010" when "001",
            "00000100" when "010",
            "00001000" when "011",
            "00010000" when "100",
            "00100000" when "101",
            "01000000" when "110",
            "10000000" when "111";
end Selected
```

While both could be treated as specifications, it is more natural to consider the description in Figure 12 as a specification and Figure 11 as a description. There is more to this than a simple intuitive notion of what is declarative and what is efficiently executable. The notion of interpreter bias, which is described in more detail in Section 2.4 clarifies this by codifying the effect of the operational model with which a description is given meaning on the interpretation of the description as a specification. Interpreter bias is the effect of a specification being forced to look like an interpreter for the programming language due to undesired corner conditions in the operational model.

In this example, the interpreter bias would predispose the specification of Figure 11 to *look like* a simulator with a stack for the function call and some memory for the constants and variable. It would take a very fancy set of analyses to determine that the design of Figure 12 was the true intent of the specification. Maybe these sorts of analyses for VHDL-style languages will exist some day; for now however, they don't and so it is useful to draw a distinction between efficiently-executable black-box models and expository specification models in VHDL.

**Netlist Declaration**

This sort of VHDL description is simply the assembly of pre-existing parts. It is not terribly important or interesting but it is brought up here because it is a *specification* in the pure sense; a netlist is a partial specification for the placement and routing portion of the design process (the remaining items being composed of information describing constraints and estimates of resistances, capacitances, areas, delays and so forth). An example of netlist-based description is shown in Figure 13.[11] These descriptions are pure structural description and amount to only a *very* fancy netlist format.

**Figure 13** An Example of Netlist-Based Description

```
entity Full_adder is
   port(A: in Bit; B: in Bit;
         Carry_in: in Bit;
         AB: out Bit;
         Carry_out: out Bit);
end Full_adder;

architecture Structure of Full_adder is
   signal Temp_sum: bit;
   signal Temp_carry_1: bit;
   signal Temp_carry_2: bit;
   component Half_adder
      port(X: in Bit; Y: in Bit;
            Sum: out Bit;
            Carry: out Bit);
   end component;
   component Or_gate
      port(In1: in Bit: In2: in Bit;
            Out1: out Bit);
   end component;
begin
   U0: Half_adder
      port map(X => A, Y => B,
               Sum => Temp_sum, Carry => Temp_carry_1);
   U1: Half_adder
      port map(X => Temp_sum, Y => C,
               Sum => AB, Carry => Temp_carry_2);
   U2: Or_gate
      port map(In1 => Temp_carry_1, In2 => Temp_carry_2,
               Out1 => Carry_out);
end Structure;
```

**Switch-Level Modeling**

VHDL is a modeling language and one common problem in electronic design automation is the need to model the behavior of a digital circuit at the transistor level, viewing the circuit as a network of switches. In this case, an attempt is made to approximate the continuous phenomenon of transistor operating-point behavior with a discrete approximation: that of a bidirectional switch. When the switch-level circuit modeling problem is cast within the discrete-event framework a further

11. This example is taken from [LSU89], pages 20-21.

approximation must be made as the discrete-event paradigm is a unidirectional one, events propagate unidirectionally down wires, whereas the switch-level model is bidirectional as switches can carry information both forwards and backwards.

The key point here is that descriptions such as switch-level models are attempting to model *physics*. In this case, an attempt is being made to approximate a continuous phenomenon within the discrete-event framework. This is very bad for specification for it depends heavily on the level of the description: there is a huge reduction in detail from the continuous domain to the abstract discrete-event domain and much of the meaning in the description is held in the mind of the beholder. Such descriptions are very ambiguous due to this implicit reduction and can hardly be expected to be intuited by automated means. Expecting a synthesis system to be able to intuit the continuous behavior from the abstract discrete-event behavior is effectively asking for the inverse of the reduction shown in Figure 4. Although this sort of low-level representation is not useful for specifying designs, many current simulator implementations are oriented towards this level of simulation [MCC91] and provide special-cased value systems which are known to be more efficiently treated by the simulator.

Ause of VHDL to model a bidirectional transmission gate is shown in Figure 14.[12] This gate is part of a larger switch-level modeling package and value system which is developed in [Coe89]. As one might expect, attempting to model bidirectional switches with a unidirectional switch-level simulator is inefficient, but not impossible. The essential trick involved in switch-level modeling using VHDL is the abstraction of a transistor (as a switch) into a small finite-state machine. The various strengths emitted by the switch are modeled as values in a lattice-like value system encoded in VHDL’s enumerated data types.

**Figure 14** Modeling a Transmission Gate

```
USE std.std_logic.ALL;
USE work.ALL;
ENTITY nfet IS
   GENERIC(gdelay: time := 3 ps;
            maxstrength: t_strength := ‘R’);
   PORT(g: IN t_wlogic;
         src, drn: INOUT t_wlogic);
END nfet;

ARCHITECTURE nfet_behavior OF nfet IS
   COMPONENT bxfr_type
      GENERIC(gdelay: time := 3 ps;
               maxstrength: t_strength := ‘R’);
      PORT(g: IN t_wlogic;
            src, drn: INOUT t_wlogic);
   END COMPONENT;
BEGIN
   i1: bxfr_type
      GENERIC MAP(gdelay, maxstrength)
      PORT MAP(g, src, drn);
END nfet_behavior;
```

12. This example is taken from [Coe89], pages 104-108. The **bxfr_type** model implements a bidirectional transmission gate as a finite state machine; that description covers two full pages and is omitted here.

This example of the use of VHDL to model digital circuits at the switch-level seem to be so egregious as to be rejected outright as a misuse of the language. It is important however for it is an example of an interpretation of VHDL text which is given meaning beyond the context of the VHDL language and as such it illuminates a far subtler issue of specification: the use of different **Bit** data types:

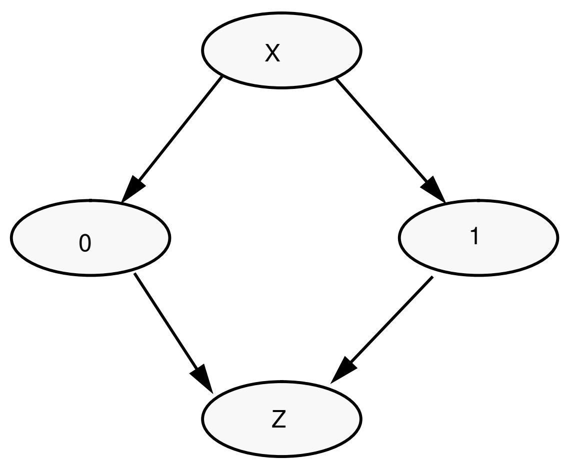
```
type Bit is (‘0’, ‘1’);
```

but for historical and simulation accuracy reasons they were written with the commonly-declared type **MVBit**,

```
type MVBit is (‘0’, ‘1’, ‘X’, ‘Z’);
```

which has an intuitive interpretation as a “bit” which is embedded in the lattice shown in Figure 15. This lattice is extremely useful for simulation purposes as it allows the propagation of unknown or undefined values. There are even more complicated ones which have been proposed[Coe89] and some have even been standardized in their interpretation by the EIA and the IEEE[13] such the one shown in Figure 16.

**Figure 15** A Four-Valued Bit Lattice




The relevance to specification here is that the specification interpretation of these “bits” is not the ones with which they have been imbued by their designers. As far as a specification interpretation is concerned, these are all multi-valued variables and it will take the relevant number of bits to encode them. For example, the specification interpretation of the data type shown in Figure 16 which is taken from the IEEE **LOGIC_SYSTEM** package [BIL90] is that of a multi-valued variable with 9 possible values. Thus, with respect to specifying a digital circuit, at least 4 bits to encode each of the possible values of the data type; this was certainly not the intent of the IEEE models-standardization committee.

The interpretation as a specification of the various gate-level value systems is certainly not what the designers of these systems intended. They had an idea which was more like that of Figure 4 wherein these value systems approximate reality within the framework of VHDL. To a certain extent they accomplish that aim. The importance for the consumer of specifications to be made aware of the limitations of the specification interpretation of VHDL; that there is an implicit level of

13. The example is taken from [Bil90]

**Figure 16** A Multi-Valued Bit Type Declaration

```
TYPE std_ulogic is (
      'U', -- Uninitialized
      'X', -- Forcing 0 or 1
      '0', -- Forcing 0
      '1', -- Forcing 1
      'Z', -- High Impedance
      'W', -- Weak 0 or 1
      'L', -- Weak 0 (for ECL open emitter)
      'H', -- Weak 1 (for open Drain or Collector)
      'D' -- don't care
);
```

specification and that the reductions in detail described in Figure 4 do not apply to specifications where everything is defined explicitly.

**Analog Modeling and Other Exotic Uses**

There are a number of other uses to which VHDL has been put. One of the most inventive uses is modeling analog signals for mixed analog-digital simulation *completely within VHDL*[14]. This is accomplished by defining behavioral VHDL models which update the values on signals as per the usual methods of analog behavioral modeling. The values on signals are records which contain the coefficients of polynomials that, if evaluated, would give the analog voltage on the wire. This may be inelegant and inefficient, but it is a "recommended" practice for certain government contracts.

**Summary**

The point of this tour through the range of possible descriptions in VHDL was to provide some idea of where VHDL specifications might fit, were it to be defined. Of all of the descriptions that have been presented, none of them indicate in *any* way, save for the English words used in the identifiers, which sort of description they are. The policy is in the mind of the beholder.

Upon being given a random chunk of VHDL text, a simulator can analyze it, compile it and simulate it, independent of the intended user's interpretation of the description. The operational rules for simulation are fixed; it is the user's interpretation of the results which change across the various styles. On the other hand, it is a much different situation for a synthesis system to interpret VHDL as a specification, for there is some *presumed level* of description at which the VHDL text is written. This cannot change over time if the specification is to be considered sound and rigorous.

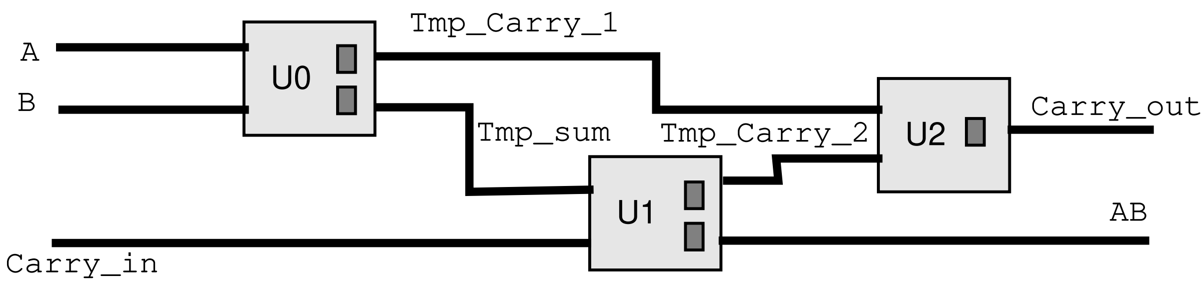
A synthesis system cannot be expected to analyze any arbitrary VHDL description and intuit the writer's original idea. As a specific example of this, consider the 1-bit combinational full-adder shown in Figure 13. While it is described as a full adder and a simulator will produce the outputs of a full adder on **AB** and **Carry_out**, that is not what is being *specified*. What is being indicated in that VHDL text, when interpreted as a specification, is three communicating finite state machines **U0**, **U1** and **U2** each with its own state: the drivers of the output signals of each instance: **Temp_carry_1** and **Temp_sum** for **U0**, **Temp_carry_2** and **AB** for **U1** and **Carry_out** for **U2**.

In spite of the fact that the system of Figure 13 describes a network of three finite state machines, it can be interpreted by the user as a combinational network. This is because the three machines **U0**, **U1** and **U2** are connected in a unidirectional

14. This was first described to me by Alfred Gilman of Intermetrics as part of a modeling standard for DoD projects.

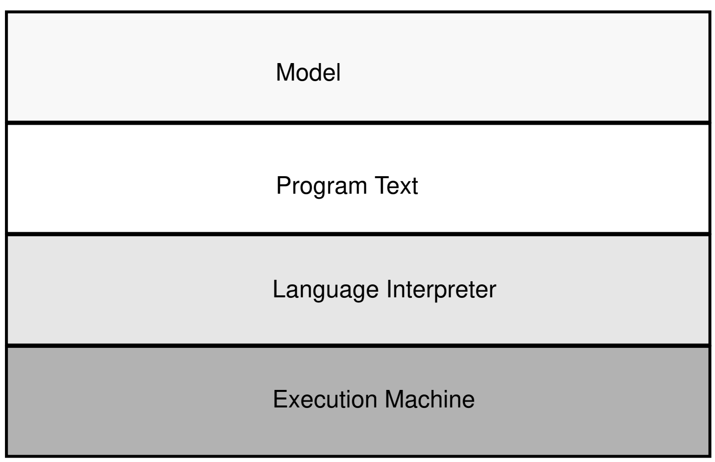
manner thereby forming an acyclic pipeline as shown in Figure 18. Viewing the behavior of whole entity, one can interpret its behavior as that of a full adder. With respect to the inputs **A**, **B** and **Carry_in** then, the two externally-visible outputs **AB** and **Carry_out** are recognizable as the cover of the 1-bit combinational adder. Such is the distinction between an interpretation of VHDL as a specification *declaration* and as an executable *simulation* model.

**Figure 17** Combinational Logic as A Simple Pipeline




This presumed level of specification relates directly to the policy level shown in Figure 18. With respect to the use of VHDL for simulation this policy level can change radically; it can change from being a test bench to being a register-transfer description to being a switch-level transistor netlist or even something else. With respect to an interpretation of VHDL as a specification for synthesis or verification, the definition of this policy level cannot change; it must always remain fixed.

**Figure 18** The Traditional Levels of A Simulation Model




Concretely this means that it is not possible to have mixed interpretations of the VHDL descriptions which can still be interpreted as specifications. The interpretation that is proposed here is the synchronous one and that means that the interpretation of the VHDL, the presumed level of description, is one of communicating finite state machines: there is no structure and there is no “physics” allowed. This means for example, that there is no provision for an “extra” piece of combinational logic or a “declaration” of a clocking discipline, or an asynchronous handshake or anything of that nature.

The *whole* VHDL model is being interpreted as a specification for a finite automata, independent of what the syntax of the model or the identifiers used in the model mean to the reader.

To drive home how this affects a user's view of the language, consider the example of Figure 19. This example shows what one might think of as a three-input "and" gate operating on a "bit" signal on a domain of *true*, *false* and *unknown*. In fact, however the synchronous semantics of VHDL would interpret this as three communicating finite state machines, each of which has one state variable which ranges over three possible values. The important point is that each VHDL process is interpreted as an individual finite state machine *for the purposes of specification*.

**igure 19** Really a Finite State Machine

```
package ThreeValued is
   type Bit3 is ('0', '1', 'X');
   function "and"(A, B: Bit3) return Bit3;
end ThreeValued;

package body of ThreeValued is
   function "and"(A, B: Bit3) return Bit3 is
   begin
      if A = 'X' or B = 'X' then
         return 'X';
      elsif A = '1' and B = '1' then
         return '1';
      else
         return '0';
      end if;
   end "and";
end ThreeValued;

use Work.ThreeValued.all;
entity and3 is
   port(In1: in Bit3; In2: in Bit3; In3: in Bit3;
         Out1 out Bit3);
end and3;

architecture really_is_an_fsm of and3 is
   signal tmp1, tmp2: Bit3;
begin
   tmp1 <= in1 and in2;
   tmp2 <= in2 and in3;
   Out1 <= tmp1 and tmp2;
end;
```

### 2.3 The Nature of Parallelism

In addition to the presumed level of interpretation of VHDL, there is the aspect of parallelism involved in its specification of computing systems. Parallelism is a complex phenomenon for it deals with both *occurrence* and *time*. The attributes of parallelism in a language semantics drastically affects the form that any realizations of programs written in that language can take. The most problematic aspect of parallelism is that of time (not of occurrence); what is assumed about time in the semantics affects all else. It is useful to consider that time has the following aspects:

**Partial Ordering of Events**
Events can be considered to occur only in relation to one another via the relations *before* or *after*. Some events may be incomparable under this relation; such events are said to occur in parallel.

**Equivalence of Events**
In addition to the ordering of before and after, there is the notion of *at the same time*. Under this model only events occurring at the same time are said to occur in parallel.

**Delay Between Events**
Finally, there is delay where events are not points in time, but rather intervals. This reflects the aspect of reality wherein devices react in finite time. Two events are said to occur in parallel under this model if each occurs *during* the other.

The following section describes the idea of parallelism as found in VHDL and then synchronous parallelism is presented in contrast.

### 2.3.1 Asynchronous Parallelism

The fundamental goal of the VHDL simulator is to provide a means for simulating the parallel execution of the models. Most VHDL simulators are purely sequential programs[15] and use the standard techniques to simulate parallelism within their sequential framework. These techniques involve maintaining queues of pending values and lists of processes to be executed based on their sensitivities to signals on which the values are propagated.

What ultimately results from this operational model is that the type of parallelism which is exposed at the programmer level is *asynchronous parallelism*. In this sort of parallelism, two events can only occur in relation to each other; either one is first and the other is second, or vice versa. Two events on two separate signals never occur *at the same time.*

### 2.3.2 Synchronous Parallelism

The standard notions of VHDL parallelism is asynchronous parallelism because it is not possible to relate state changes in any stronger way than by a partial ordering of events. Nothing can happen at the same time and so there is only before and after.

The synchronous model of time abstracts time into discrete instants between which nothing of interest occurs. Synchronous parallelism then requires that events in parallel occur at the same time. There is still a notion of before and after, but the partial ordering of events is not used to define parallelism.

Synchronous parallelism does not offer or allow nondeterminism. Because of this restriction, and due to the fact that there are only a finite number of possible events, an interpretation of synchronous parallelism as a finite automaton can be generated; Section 3.3.2 outlines how this is done for an existing synchronous language - Esterel.

**The Synchrony Hypothesis**
The interpretation of synchronous systems as finite automata is driven by two hypotheses: the *synchrony hypothesis* and the *strong synchrony hypothesis*. These are termed hypotheses because synchronous systems are only guaranteed to func-

15. The VHDL simulator described in [Wil90] is one that is not; it distributes the simulation across a network of workstations.

tion correctly if these preconditions are true. The verification that the preconditions are met in any given implementation of a synchronous system is a proof obligation imposed on the designer.

The synchrony hypothesis states that only explicit delay exists. All else (all other computations) are instantaneous. This implies that the program only *reacts* in response to its environment. The strong synchrony hypothesis requires the synchrony hypothesis and requires in addition that control steps take no time.

Both of these hypotheses are clearly not true in any physically realizable system. However, because time is measured only at specific *instants* between which nothing of interest to the system occurs, it is possible to preserve this rather elegant fiction through a timing analysis step. So long as the implementor can assure that for the given instance of the system, all computations triggered by an event finish before the arrival of the next event then the conditions of the two synchrony hypotheses can be considered to have been met. This is the timing verification step, and it corresponds exactly to timing verification in hardware design.

## 2.4 Language Interpreters versus Program Specifications

In addition to the issues of the presumed level of interpretation and the definition of parallelism, there is also the issue of how the language and the language's semantics affects the specification. It has been said that language affects what can be said to such a degree that certain concepts which are inelegantly dealt with in language simply are not treated. Thus a qualification of the effect of the phrasing of the description on its interpretation as a specification needs to be made.

The comments of this section are phrased in terms of *programming languages* and *interpreters* and generally have a software orientation. This is done on purpose to accentuate the operational aspects of the VHDL simulator as an *interpreter* for the VHDL language which is a *parallel programming language*. It is within this framework of a formal language and its interpreter that a notion of specification is being proposed. This extraction process is more difficult than it might seem due to the of the intrusive effects of the language's interpreter as explained below.

For the purposes here, the ultimate goal of hardware description languages is in the expression of computing, not in the generation of the specific syntax by which that computation is expressed; *i.e.* the language is a means to an end, not an end unto itself. While this seems obvious, there are those who take the opposite tack - namely that the purpose of the programming language is to record all relevant design decisions and further, that any support for comments in a programming language are an admission that providing such all-encompassing design flow support has not been possible. Such an extreme position is not necessary here as only the aspects of the specification represented in the language is relevant.

In Figure 18 the traditional levels of a hardware description language program are shown. Description is accomplished through various levels of abstraction until finally a machine can be instructed to perform the computation; this defines the meaning of that hardware description. The trick of hardware description is to define the intervening abstractions so that they can ultimately be implemented efficiently.

The following sections argue that the two intervening levels of Figure 18, the *program* and *interpreter* levels, are incidental to the task at hand and introduce *interpreter bias* into implementations. Interpreter bias is the term that is being used here for tendency of final implementations to look more and more like a general facility which indirectly computes by manipulating a data structure representing the computation. Optimizations tend to reduce this effect, but do not completely erase it because of their partial nature. A language's interpreter bias prevents the optimization of these data structure manipulations into native machine-level operations.

One can treat the top level of the hierarchy shown in Figure 18 as a *specification* of the computation to be performed on the machine. The job of the synthesis system, in the case of hardware, or the compiler in the case of software, is then to produce *something* executable at the lowest level (the *machine*) which will satisfy the specification. An option for defining the meaning of the top-level specification is to use quantified logic formulae or some sort of exhaustive listing of input/output pairings. Clearly though, for this notion of specification to be practically useful it must to be more concrete and succinct than large masses of formulae or an unstructured mapping. Elaborating the representation of a true and meaningful specification for computation is the subject of the next sections.

**Program Specifications**

The goal of high-level design is to take a specification (an idea) and produce something (an executable) that will compute the required function. A specification is something which defines the relationships that shall hold on any implementation. Many specifications are considered to be but an exhaustive list of the possible input/output relationships that hold over all possible implementations. Ideally this exhaustive list is succinct. For a small class of simple functions, this can be done. The boolean functions are such an example, for succinct specifications of logic functions can be given in either sum-of-products or as a binary decision diagram. For more complex functions, especially those whose output depends on past inputs or outputs (*i.e.* they maintain internal state) the task is far more complicated.

The key point to notice about a specification is that it indicates what observations the environment wishes to be able to make about any implementation. At the extreme case, if nothing can be observed, as is the case with a so-called “black box,” then the exhaustive list of input/output pairs is required. At the other extreme, if everything can be observed, then the result is a “glass box” that is a pure structural description of the computation. Somewhere in the middle is the notion of specification that is desired here.

The idea is to define specifications that allow for a trade-off along the axis of internal visibility; ranging from black box to glass box. Each of these alternatives will have varying degrees of succinctness and, conversely, flexibility of implementation.

**Regular Expressions as FSM Specifications**

What can be observed from the study of regular expressions [HU79] is that they provide just such a specification for the class of finite automata. As has already been argued, the use of the finite state machine model[16] is a good one in the case of real-time software and simulator kernels, so it is very convenient that regular expressions are a succinct specification for the domain of finite state machines.

For each regular expression, there is some set of finite automata that recognize that regular expression. In fact, it is convenient, for this example, to examine the software tool which produces one of these automata for the case of software: **lex** [LS75]. The input to the program is a set of regular expression which is taken as the specification of the automaton to produce. Depending on the required state-space size and code-size trade-offs requested, the resulting implementation is either larger and faster or smaller and slower.

On the other hand, examine this same example in the light of interpreter bias. by considering the case of a hand-coded scanner versus the **lex**-generated one. It would take an extremely powerful optimizer to determine the function of the hand-generated scanner to the level of detail required for use as a specification for the automaton. On the other hand, the regular expression specification is entirely adequate for it only identifies the minimal set of points of user-observability in the final automaton.

The use of regular expressions as a specification for a finite automaton was not presented only to illustrate the notion of a language interpreter versus a program specification. This exact idea is used to define the specification nature and thus the compilation procedure for the seemingly imperative language, Esterel and that procedure is outlined in Section 3.3.2.

16. Finite state machines, finite automata, (finite) state transition graphs and regular expressions can all be shown to be equivalent [HU79]. Here these terms are used interchangeably with *regular expression* emphasizing specification and *finite automata* or *finite state machine* emphasizing implementation.

# Chapter 3

# Synchronous/Reactive Languages

One of the most well-developed and extensively published synchronous languages is the Esterel language [CIS88] [Ber91] [BDS91] [BCG86] [BC84].[17] What is presented here by no means represents the whole language, rather only the most relevant highlights of semantics are presented with a few examples of syntax. This chapter is focused mostly on Esterel but the final section, Section 3.4, will describe the implications of the synchronous assumptions for VHDL.

Esterel, like other synchronous languages, requires that the strong synchrony hypothesis hold, as described in Section 2.3.2. The Esterel program is thus a specification for the computing which is to be performed rather than a representation of a data structure on which computation shall occur. Also, as a synchronous language, an Esterel program is reactive, that is it computes only in response to changes in its environment. All changes in the environment, even the passage of "real" time are measured in terms of events impinging on the program. This indistinguishability between metric time (in seconds) and symbolic time (in "ticks") will become important in the next section as the examples show the interchangeability between the signals **SECOND** and **BUTTON** with no loss of description.

The strong synchrony hypothesis allows the following claims can be made about Esterel semantics: non-delay events take zero time and delay elements take exactly the amount of time specified. The effect of the strong synchrony hypothesis is the separation of the correctness of an Esterel program into two independent parts: a functional correctness part and a temporal correctness part. This independence of function and time is much the same as that which is found in clocked digital circuits where so long as the circuit is able compute its next-state and output functions faster than the clock cycle time, the circuit is considered to be correct. Different methods are used to verify the function and performance of the circuit, within the clocked-digital paradigm they are seen to be independent. In the case of Esterel programs, the strong synchrony hypothesis allows for different means of verification to be applied to each aspect, *e.g.* automata-theoretic methods to verify functional correctness and linear instruction scheduling methods to verify the temporal correctness.

### 3.1 Overview of the Language

The Esterel language is divided into two levels in order to define a simpler sound semantics for the language. There is a *core language* over which the semantics is defined. In addition, there is an extended language which is defined in terms of the core. I will only review the key concepts of the core language as the extended language provides only syntactic convenience and it adds no new semantic power.

The important semantic feature of the language is that it has both an interpretive aspect and a specification aspect and these are both exposed in the construction of the Esterel interpreter and the Esterel compiler respectively. These aspects are described in Section 3.3.1 and Section 3.3.2. The interpretive aspect indicates how the execution of a statement changes the state of the interpreter and what successor statement will be executed. The specification aspect is consistent with the interpretive aspect but indicates only what computations and states must be observed in any realization of the specification so that the compiler generates software which respects this minimal set of observability conditions.

The execution semantics of Esterel follows the synchronous paradigm. Time is separated into discrete *instants* between which nothing of interest occurs. State transition computations are considered to be instantaneous and the only delays are

17. The descriptions presented here are from [BDS91] and so should be consistent with the latest available compiler [CIS88].

those which are explicitly declared. There is an issue of the causal correctness of an Esterel program; but [BC84] demonstrates how checking for this condition can be performed statically during compilation.

Esterel programs consist of networks of *signals* and *processes* with processes maintaining *statements* and *variables*. Signals are the only means by which processes are allowed to communicate (shared variables are disallowed). A process has an interpretative aspect as the statement which will be executed in the current instant and a specification aspect as the current state (of a finite state machine). Variables hold data values which are local to a process. The variable values and the signal values available on the input signals determine the next statement to be executed, or the next state to be entered, depending on which aspect of a process one is considering.

### 3.2 Core Language Constructs

Seven core language constructs are presented here which show the essential characteristics of Esterel. There are of course a host of other constructs which are not presented here; they can be found in [CIS88].

The basic construct of Esterel is the *statement*. The purpose of a statement is to perform some work and possibly pass control on to a successor statement in the same instant. If statement completes its work and passes control, then the statement is said to terminate in the first instant. If the statement does not release control in the first instant then it must be a delay of some sort.

Each of the statements below are part of the core language. Each statement is presented with its syntax, informal semantics and an indication as to whether the statement terminates in the first instant.

**nothing**

This statement does nothing and terminates instantaneously. It is mainly used in conjunction with other constructs to define the meaning of statements in the extended language.

**halt**

This statement does nothing and *does not terminate* - ever. It too is used mainly to define statements in the extended language.

**emit signal(exp)**

The value of **exp** is made available on **signal** in the current instant. The statement terminates in the current instant.

```
loop
  instruction
end
```

The **instruction** is executed repeatedly. It is a static error for **instruction** to take zero time for that would represent an infinite amount of work being completed in zero time.

```
[
  instruction1
||
  instruction2
||
  ...
||
  instructionN
]
```

Each **instruction** is executed in parallel, synchronously with the others. The parallel statement completes when every **instruction** has completed. This is a statement where the distinction between **nothing** and **halt** is apparent and is often used to advantage.

**tag T in**
  **instruction**
**end**

The **instruction** is executed in the current instant. If an **exit T** is executed then the whole tag-body terminates. Otherwise the tag-body terminates when **instruction** does. This is another statement where the distinction between **nothing** and **halt** is apparent and where that distinction is useful in conjunction with looping and parallelism in the contained **instruction**.

**do**
  **instruction1**
**watching signal(variable)**
**timeout instruction2**

The **watching** statement has a rather large syntax, however, it is the one construct in the core language which accomplishes "real work" for only it introduces delay. The **instruction1** is executed in the current instant; if it terminates then so does the watching statement. If **signal** occurs in the current instant then **instruction1** is killed and control is passed with **variable** bound to the value on **signal** at the time of the event. The time-out **instruction2** is executed if an event on **signal** occurs before the main **instruction1** terminates.

There are other constructs in the core language which provide for sequential composition of statements, parallel composition, conditional execution, instantiation of local signals and variables and operations on variables. There is also a weak **module** construct which provides for macro-like modules, an example of which is presented in Section 3.2.1.

**A Small Example**

The following simple examples demonstrate the use of the core language. Of note in these examples is the duality between so-called relative time demarcated by events with names like **BUTTON** and so-called metric time which is measured in seconds. Here both are measured in terms of events on the signal **BUTTON** and **SECOND** respectively[18].

```
do
   await BUTTON;
   emit ACTION
watching SECOND
timeout
   emit ALARM
```

The statement above implements the specification "wait for a button press and then do the action or else time-out in one second and ring the alarm."

```
trap END in
   [
      await SECOND;
      emit ALARM;
      exit END
   ||
      await BUTTON;
      emit ACTION;
      exit END
```

18. This implies that if one is interested in an Esterel program with a real time behavior then that program will have to exist in an environment in which it is "poked" to tell it to keep count of the passing (nano)seconds. Ensuring that the software can compute at speed is the temporal verification obligation alluded to in Section 2.3.2.

```
   ]
end
```

This second example uses the parallelism operator and a tag-body to get the same effect. Its informal specification would be "wait for one button press and perform the action while waiting for one second at which time, ring the alarm. Quit after which ever occurs first."

### 3.2.1 An Example - A Mouse Handler

This example shows how Esterel is used to create a simple application. This example produces indications of the number of mouse clicks on the mouse in a given interval.[19] The example is broken up into three separate modules, **Counter**, **Emission** and **Mouse**.

The **Counter** module counts occurrences of **CLICK**. When a **RST** is present, **Counter** emits a valued signal **VAL** whose value is the number of occurrences of **CLICK** since the last reset.

```
module Counter:
input RST, CLICK;
output VAL(integer);

var v: integer in
   do
      v := 0;
      every immediate CLICK do
         v := v + 1;
      end;
   watching RST;
   emit VAL(v);
end
```

The input signals **RST** and **CLICK** in the above code are "pure" in the sense that they do not carry any value and what is important about them is merely their presence or absence. The output signal **VAL** on the other hand carries a value, although all that is important with respect to the execution of the example is whether or not a value, any value, was emitted onto VAL in an instant.

The module **Emission** processes the value of **VAL** which is assumed to be consistent with the definition found in module **Counter**. The **Emission** module outputs one of **NONE**, **SINGLE** or **MANY** to indicate which decision was made; the count is communicated to **Emission** via the signal **VAL**.

```
module Emission:
input VAL(integer);
output NONE, SINGLE, MANY;

await VAL;
if ?VAL = 0 then
   emit NONE
```

19. This example of a mouse handler written in Esterel is taken directly from [BDS91], page 1296, as is the description of the function of the mouse handing code.

```
else
   if ?VAL20 = 1 then
      emit SINGLE
   else
      emit MANY
   end
end
```

Finally, the main module, **Mouse** in a global loop puts in parallel a copy of the **Counter** module and the **Emission** module and a process which resets the **Counter** every five **TOP**.

```
module Mouse;
input CLICK, TOP;
output NONE, SINGLE, MANY;

signal RST, VAL(integer) in
   loop
      copymodule Counter
   ||
      await 5 TOP;
      emit RST
   ||
      copymodule Emission
   end
end
```

The important point to notice in the example is the instantaneous communication which is *declared*. There is instantaneous communication inside the parallel statement since the second branch emits the signal **RST** that is received by the **Counter** in the first branch. Then at the same instant, **Counter** emits **VAL** that is received and processed by **Emission**.

A second point to notice in the example is the weakness of the **module** construct. It is merely a macro construct and provides only a convenient textual substitution mechanism. What is desired in large designs is *true hierarchy* which allows for modules to be abstracted in terms of simpler automata that specify the same behaviors.

## 3.3 Interpretation and Specification

As mentioned previously Esterel can be treated both as a language to be interpreted and as a specification for which there exists a satisfying implementation. In this first sense it is no different than any other formal language; it is in the second sense that it is different. Both of these treatments are reviewed in the following subsections..

### 3.3.1 The Interpretation of Esterel

Because Esterel is a formal language which describes computation, it has the property that it can be interpreted. The definition of this interpreter is hinted at in both [BCG86] and [BC84] but as that implementation was far and away slower than the result of the specification treatment,[21] not much attention is paid to the interpreter. The purpose of reviewing its existence here is simply to show that it exists but that its implementation is quite awkward.

20. The notation **?VAL** indicates that the *signal* namespace should be used for acquiring the rval in this expression. There are five namespaces in Esterel, as described in [CIS88]. In this example one can see two namespaces straight away: the *signal* namespace and the *variable* namespace.

The interpreter consists of a set of *potential functions* which are maintained. These potential functions are data structures, one for each signal, which describe the possibility of an event occurring on that signal in the current instant. In addition to existing and having to be maintained by the interpreter, the implementation of the functions themselves is problematic. The functions are charged with the duty of computing the successor statement to execute after a given event occurs. This is problematic because the execution semantics of Esterel is *not* syntax directed. That is the successor statement is not always trivially obvious from the syntax of the statement being executed. Consider the following example:[22]

```
input single signal s1(int),
output single signal s2(int) in
   var X, Y: int in
      await s1(X);
      every next s1(Y) do
         emit s2(X);
         X := Y
      end
   end
end.
```

The successor *statement* upon receiving an input event on **s1** is as follows:

```
input single signal s1(int),
output single signal s2(int) in
   var X := 3, Y: int in
      every s1(Y) do
         emit s2(X);
         X := Y
      end
   end
end
```

This second statement is essentially a “currying” of the data into the new process; this new process must await further events.

The definition of the interpreter for Esterel is obtained through the use of derivatives of regular expressions [Brz62]. The interpreter computes these derivatives in the form of successor *statements* on the fly. This was the original definition of the meaning of an Esterel program. Subsequently it was observed that only a finite number of such derivatives exist and further that these implied a finite number of successor statements and that all derivatives and successors could be precomputed, numbered, and stored as the compiled form of the program. Thus the compiled version of Esterel was developed and the interpreter faded away.

### 3.3.2 The Compilation of Esterel

Compilation in Esterel is derived from the definition of execution of the language. Execution consists of processing *events*. An event is the conjunction of the conditions on each of the signals in the whole Esterel program.

21. The implementors describe the interpreter acting on the order of minutes while the compiled code (treating the program as a specification) resulted in millisecond response times - quite adequate for real-time software applications.

22. This is taken directly from [BC84]. There have been several changes to the Esterel language between the publication of [BC84] and the release of [CIS88] which are being ignored here.

$$E = (s_1 \equiv V_1) \wedge (s_2 \equiv V_2) \wedge \ldots \wedge (s_n \equiv V_n)$$

Each event is a conjunctive predicate of the values on each of the signals. The boolean values in the conjunctive form are interpreted as a bit vector, the complete set of which consists of $2^n$ codes. This set of codes is interpreted as the set of encodings of an *alphabet* for a finite automaton [HU79].

The conditions of synchronous parallelism ensure that there is only one event per instant and thus that there is only one state transition per event. This is exactly the condition required for a finite automaton. The alphabet and the transitions indicated by the program text are combined to form a set of states and state transition functions for the finite automaton. The standard algorithms [HU79][ASU86] are used to generate automata from this point.[23]

What is interesting to note here in this process is that it exploits exactly the same algorithms used in the Unix scanner generator **lex** [LS75]. The algorithms for determining the regular expression implied by the program text are unique to Esterel but the essential algorithms of the Esterel compiler have been published and used elsewhere extensively. Those parts are the conversion of the regular expression through its derivatives [BRZ62][BCG86] and into automaton code.

The novel aspect of the Esterel compiler over and above **lex**, for example, is the dynamic assembly of the alphabet for the finite automaton based on the boolean algebra induced by predicates on the signal conditions[24] and the imperative-style that the syntax of the language gives to what is fundamentally a regular expression specification for a finite automata.

This review of Esterel has described enough of that language that relationships between it and VHDL can easily observed; those relationships are the subject of the next section.

### 3.4 Implications for VHDL

Esterel and VHDL are, at first glance, very close in the *style* of description, and in the major artifacts of the languages. Both languages contain *processes*, *signals* and have an execution semantics described in terms of *events* and *waiting on events*. There are differences of course, and the purpose of this paper is to define what those differences are so that only the subset of VHDL which has the same semantics as Esterel will be used as the synchronous semantics.

#### The Treatment of Processes

A process in Esterel is a reactive entity which consists of a nested set of statements. The interpretive semantics of the process is that of the next statement to be executed in the first instant, upon reacting to some event. The compilation semantics of the Esterel process is that of a finite state machine in a specific state which will execute a state transition function on the next event.

A restricted class of VHDL processes posses these same qualities. A process is considered to be *waiting on its sensitivity list* for events to occur on that sensitivity list. When an event occurs, then the process is activated and it executes. For example, the following would be an implementation of the **Counter** described in Section 3.2.1:

```
architecture somehow of something is
   signal VAL: integer := 0;
begin
```

23. In [BDS91] the claim is made that the automata generated by the Esterel compiler [CIS88] are minimal.

24. This boolean algebra formulation is the same as that which is required by **COSPAN** [HK90] and other automata-theoretic verification tools.

```
    Counter:
    process(CLICK'Transaction, RST'Transaction)
       variable v: integer := 0;
    begin
       if CLICK'Transaction then
          v := v + 1;
       end if;
       if RST'Transaction then
          VAL <= v;
       end if;
    end process Counter;

end somehow;
```

Of immediate note in this description is that an Esterel process can terminate while a VHDL process never terminates. Within the notation of Esterel it is as if the VHDL process has an implicit *do forever* loop around it. Thus the translation of Esterel back into VHDL, as was attempted in this last example, is not directly defined. This should not be a problem for the purposes here because the translation being designed is VHDL to Esterel; the second can model the first, but not vice versa. Further examples of syntactic translations between VHDL and Esterel are given in Chapter 4.

**The Treatment of Signals**

The second area of similarity between VHDL and Esterel is the use of signals and events on those signals. A difference between the two languages exists however in that events on wires in VHDL occur singly, $E = (s_1 \equiv V_1)$ while in Esterel an event on a wire is really the conjunction of all of the signals which are active at a given instant:

$$E = (s_1 \equiv V_1) \wedge (s_2 \equiv V_2) \wedge \ldots \wedge (s_n \equiv V_n)$$

This is the essential distinction between the synchronous semantics of Esterel and the discrete event semantics of VHDL.

In Esterel the event space forms a boolean algebra[25] which is interpreted as the alphabet of the finite automaton. The main idea behind the synchronous semantics of VHDL is to derive this same sort of relationship among the VHDL events by subsetting the allowable temporal behaviors that the restricted simulator will support. This will induce the boolean algebra onto the VHDL event space and define the synchronous subset.

**The Treatment of Time**

Time in VHDL is separated into two levels, as shown in Figure 20. The upper level is intended to be used in the modeling of real-world events and is measured in *seconds* (or fractions thereof). The lower level is called unit-delay or $\Delta$-delay and is intended for use by the simulator to allow events on signals to propagate until stability is reached. There is no restriction on the number of $\Delta$-delay events which can be fit between any two macro-time slices. As such, the description of Figure 21 computes an infinite amount in zero seconds.

The central idea behind the synchronous subset of VHDL is that the $\Delta$-delay level of time can be done away with under certain conditions. The use of $\Delta$-delay can be done away with if at each point the computation implied by the event-propagations is of fixed length. An example of this is shown in Figure 21. Shown in Figure 22 is a common example of a situation where the $\Delta$-delay nature of the simulation can be replaced by a simple function which relates the triggering event to the result event. In this example, the standard simulation semantics dictates that an event on **A** or **B** will trigger the first

25. See [Kur90] for more specific details on the relationships between boolean algebras and the alphabets of finite automata.

**Figure 20** The Two-Level Model of Time in VHDL

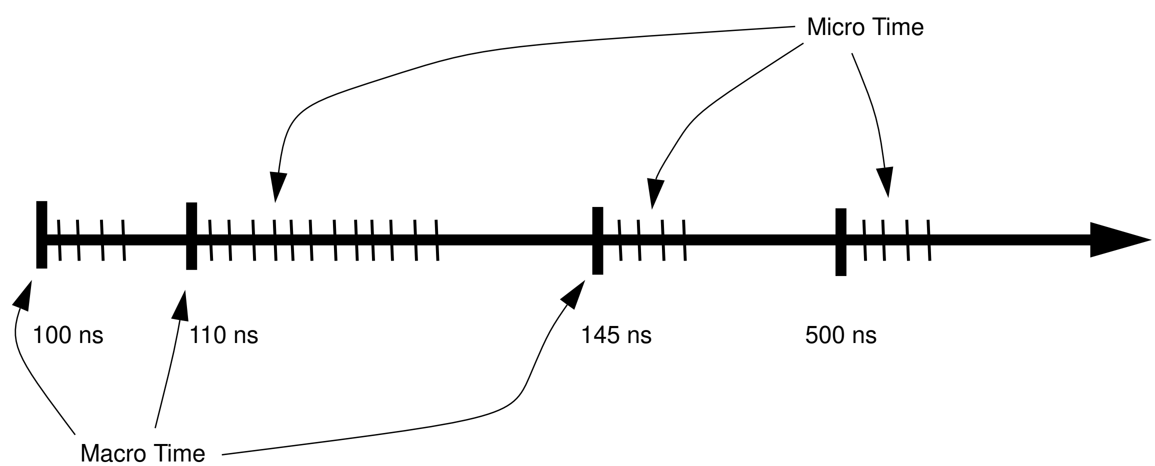



**Figure 21** A Non-Reactive Clock in VHDL

```
architecture not_reactive of CLK is
   signal clock: bit;
begin
   process
   begin
      clock <= not clock;
   end process;
end not_reactive;
```

continuous signal assignment to recompute the value on **Tmp**; this would then cause the value of **O** to be recomputed. Two $\Delta$-level iterations are required by the simulator.

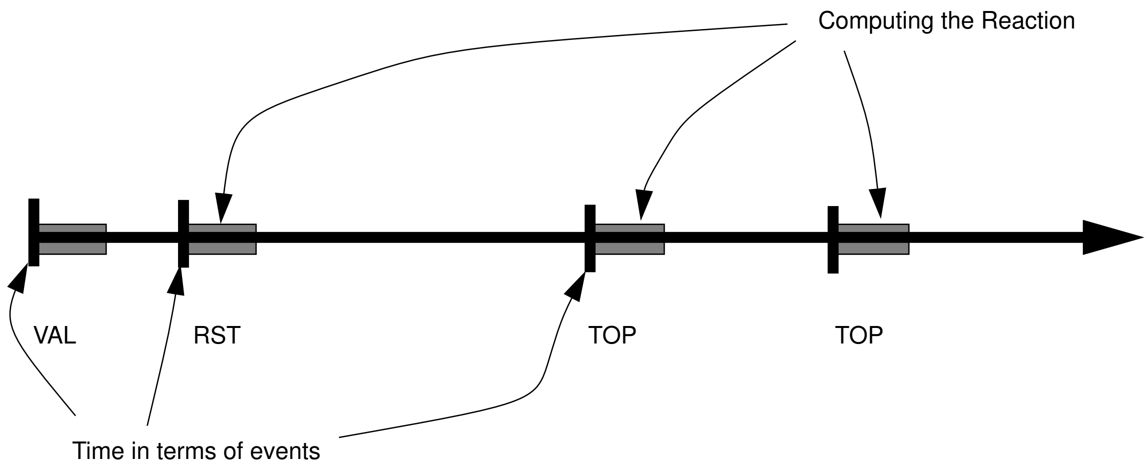
**Figure 22** An Example of Needless $\Delta$-Delay

```
entity Some_Function is
   port(A, B, C: in Bit; O: out Bit);
end Some_Function;

architecture Reactive of Some_Function is
   signal Tmp1: Bit;
begin
   Tmp1 <= A and B;
   O <= Tmp1 or C;
end Reactive;
```

Having done away with the $\Delta$-delay level of the VHDL time model, the only remaining level is the macro-time level. The synchronous model of time schematically is shown in Figure 23.[26].

**Figure 23** The One-Level Model of Time in Synchronous VHDL




26. The *strong synchrony hypothesis* says that the computation of reactions takes *no* time and the actual situation of it taking it *some* time is shown in the figure; this is done for the purposes of clarity and to allow the reader to directly relate Figure 23 to Figure 20.

# Chapter 4

# The Synchronous VHDL Subset

The argument presented earlier was that the only truly sound way to define the specification aspect of VHDL was to do it from the semantics forward to the syntax and so the restrictions on the abstract simulator will imply restrictions on the constructs in the language. The purpose of this section is to state that set of restrictions, which is surprisingly simple: there are four basic restrictions - all others are derived from this essential set.

The Synchronous VHDL subset is defined as the full VHDL-1076 language with the following four proscriptions:

1. The signal queues for managing events are restricted to be of length one.
2. The sequential code in a process must execute without a stack.
3. The use of dynamic storage allocation is disallowed.
4. Signal propagation paths must be causal.

These restrictions on the abstract simulator are very simple, but by their application, the behaviors allowed in the abstract simulator are transformed from the potentially infinite in space (memory) and time (execution time) to the finite in both dimensions. The importance of these finiteness constraints is that they allow for a strong definition of the semantics of VHDL. Processes are constrained to act as finite state machines and thus the whole VHDL program acts as a network of communicating finite state machines. In this section the Synchronous VHDL subset is defined and a meaningful way of creating implementations based on the finite-state semantics is provided.

The execution semantics of VHDL under the restrictions outlined above are the same as those of the synchronous model. Thus, the subset simulator behavior, while remaining consistent with that of the full discrete event simulator, is defined in terms of event reactions, *i.e.* processes compute only as a reaction to events on signals to which they are sensitive. The utility of defining the subset semantics in terms of event reactions is that there exists efficient techniques for deriving a finite automata-based implementation in this case. These algorithms involve computing the *event derivatives* of the program and using those derivatives as the states of an automata that recognizes the regular language defined by the program's events [Brz62] [BS86]. This semantics and implementation in terms of event derivatives is the same as that which is defined for the Esterel language [BC84]. This similarity is a connection that is used to advantage in this section in the definition of the properties of the VHDL subset, and also in Chapter 5 in the implementation of the prototype simulator.

The implications of the four basic rules defining the Synchronous VHDL subset establishes the connection between this subset and its synchronous semantics. Instead of describing the event derivatives on the VHDL language directly, a translation procedure is defined which relates syntactic structures in the VHDL subset to programs in the Esterel language. Thus, the existing Esterel compiler [CIS88] is used both to give an operational meaning to the VHDL subset and also to provide prototype simulators as is described in the next chapter. The sufficiency of the basic restrictions and their implications on the VHDL constructs that are supportable in their presence are reviewed in the following two sections, Section 4.1 and Section 4.2. The correspondences necessary to define a syntax-directed translation from VHDL to Esterel are shown in Section 4.3. This translation is, in lieu of an explicit algorithm for compiling the VHDL programs directly to automata, the definition of the Synchronous VHDL subset. The operational semantics defined for the Esterel language [BC84] in terms of event derivatives and finite automata then serves as the basic computational model with the translation procedure defined here being used to extend that definition to apply to the Synchronous VHDL subset.

## 4.1 Implications of the Restrictions

The four restrictions listed above are designed to ensure that the synchronous subset of VHDL is still consistent with the full VHDL language, yet constrain the simulator to finite memory requirements and finite execution time. We can be confident that these restrictions do not change the semantics of VHDL for they only *subtract* possible behaviors from the full abstract simulator; no new behaviors are defined. This is the necessary condition for the restrictions and the sufficient condition is that the restrictions are the smallest set of restrictions which induce the desired finite behavior.

That this set of restrictions is minimal can be seen from a quick argument involving the deletion of any one of the rules. Taking each rule in turn, it can be seen that the exclusion of any one of these rules from the definition of the synchronous subset, and thus the corresponding admission into the subset of the constructs governed by that rule, allows for non-finite behavior in the abstract simulator. The basic restrictions are reviewed below and a description is given of how each restriction aids in ensuring that the VHDL subset is reactive and finite-state and further that a removal of any restriction would allow for undesirable behavior. A description of the effect of these restrictions on the VHDL language constructs is deferred until Section 4.2.

**Finite Memory Requirements**

The first rule restricts the signal queues to be of unit length and so it ensures that the simulator operates in both finite space and finite time. In the full VHDL language, signals events are maintained on a queue of pending events. This queue's size is unrestricted since the use of waveform assignments, transport or inertial delay (the **after** clause in the signal assignment) can append an arbitrary number of events. These pending events indicate future activity of the simulator and so constitute a potential for both an unbounded amount of state and an unbounded amount of computation. The goal is to restrict the simulator to a fixed and bounded amount of state such that the occurrence of any single event completely determines the next-state of the simulator.

The second two rules restrict the other ways in which a VHDL model can create arbitrary amounts of state: implicitly through the use of stack-based computations and explicitly through the use of dynamic memory allocation. The goal of restricting the VHDL language is to ensure that the model can be treated as a specification for a finite state machine. Clearly then if this goal is to be attained, language mechanisms for creating and maintaining potentially unbounded amounts of memory must be proscribed.

**Finite Reaction Computation**

Having taken care to ensure that no VHDL description in the subset can create or maintain a non-static amount of memory, there is still one more aspect of the computations which must be constrained in order to ensure that the descriptions fall within the reactive model. Care must be taken to ensure that any computation triggered by an event will complete in a fixed amount of time. This condition summarizes one of the most important assumptions of the reactive model: the synchrony hypothesis where the system finishes computing its reaction before the next event occurs. Simple as it may seem, this requirement has a number of interesting effects on the subset.

The direct effect of restricting each reaction computation to be finite is summarized by the fourth rule: that all signal flow paths be causal. In this case causality ensures that the result of an event which causes the system to compute its reaction does not further cause the system to react: infinite oscillations are disallowed. The condition that all signal flow paths be causal simply means that there may be no cycles in the signal flow graph of the VHDL program.[27]

27. It is interesting to note that the VHDL-1076 standard does not explicitly disallow $\Delta$-time oscillations. Many VHDL models using this aspect of the language while executing on (correct) simulators will effectively infinite loop and attempt to use an infinite amount of memory for the signal queues of pending events; the simulator ultimately fails due to lack of memory resources. Thus there are examples of "correct" VHDL which are sure to cause a valid simulator to fail. It is considered by most simulator-builders that the causality condition (identifying cycles in the signal flow graph) is too time-consuming to check for large models. The condition is left unchecked and it is thus a "programmer error" to allow such cycles to occur in any VHDL model [GI90].

A second effect of the finite computation rule is that there may be no unbounded sequential executions commencing from an event. This has the effect of requiring that all potentially unbounded loops be broken by a **wait** statement. Breaking all potentially unbounded loops with a **wait** statement ensures that the computation of a reaction can be done in a fixed and finite amount of time.

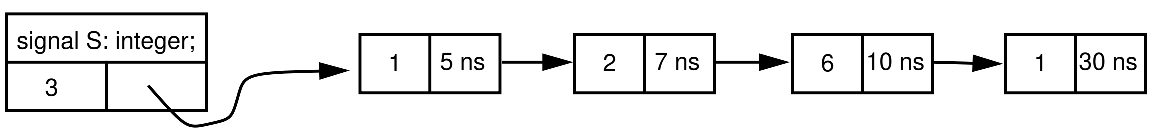
## 4.2 Implications on the Syntax

The four basic restrictions outline a broad set of rules for accepting or rejecting constructs and construct idioms from the VHDL subset. An out line of the effect of the restrictions in terms of the constructs and construct idioms which are disallowed is given in this section. The Synchronous VHDL subset is defined as all of the VHDL 1076 language subtracting off the constructs and structures which are described in this section.

### 4.2.1 Restriction 1 - Time Queues of Length One

Restricting the time queues to be of length one has the obvious effect of removing the time queues from the abstract simulator model since time queues of length one allow for a single value to be defined on the signal. The effect of this removal is that the simulator ceases to require the representation and manipulation of explicit signal queues such as the one shown in Figure 24. Instead, the signal driver values can be represented explicitly in a single variable which is updated by signal assignment. This variable, atomically updated, becomes a state variable of the final automaton under the synchronous/reactive compilation algorithm.

**Figure 24** A VHDL Signal Queue and the Process that Created It




```
signal s: integer;

process
begin
   s <= 3, 1 after 5 ns, 2 after 7 ns, 6 after 10 ns,
         1 after 30 ns;
   wait;
end process;
```

Based on this restriction, the following list of items describe constructs which are disallowed due to the lack of signal queues. Each restriction is described with a general title of the restriction, reason behind the restriction and a description of how the restriction can be implemented in the form of a static check on the VHDL source text. The important point of this last item is that the restriction can be checked at compile-time.

**Inertial Delay**

Any construct in the syntax which supports inertial delay[28] is disallowed with the most obvious one being the **after** clause in signal assignments. Inertial delay must be removed for it cannot be supported on a time queue of length one. A further point is that the preemption allows for hidden state to be stored in the signal queues.[29] The goal is to ensure that all state is explicit so that it can be identified and compiled into the final automaton.

This restriction is implemented by identifying the use of inertial delay - that is by identifying the use of the **after** keyword in signal assignments.

**Transport Delay**

While transport delay is not considered *as* intractable as inertial delay by some [AUG89], there is still an issue of the implicit state storage on the time queues. This implicit state storage must be disallowed in order to ensure reactiveness of the description, so the use of transport delay must be disallowed also.

This restriction is rather simple to implement as it requires the recognition of the **transport** keyword in a signal assignment statement. Implicit uses of transport delay as is the case in waveform signal assignments commencing with an inertial delay can be identified by recognizing and disallowing the waveform assignment.

**Signal Attributes, Part 1**

Most of the signal attributes are disallowed because they require the management of history information about the signal queue, e.g. when was it last assigned to, for how long has it *not* been assigned to, when did it last change value and what was its value when it did change and the like. These attributes are: **Delayed**, **Stable**, **Quiet**, and **Active** respectively

The restriction is implemented at compile-time by identifying the uses of the proscribed signal attributes.

**Signal Attributes, Part 2**

Other derived signal attributes are disallowed because they return the simulation time at which some effect occurred. These have no physical or specification correspondence and so are disallowed. These attributes are: **Last_Active** and **Last_Event**.

The restriction is implemented at compile-time by identifying the uses of the proscribed signal attributes.

**Signal Attributes, Part 3**

The final remaining signal attribute is disallowed too because it can be implemented at the user level and thus need not be built into the subset language. This attribute is: **Last_Value**.

The restriction is implemented at compile-time by identifying the uses of the proscribed signal attribute.

**Signal Operations**

It is possible to both implicitly and explicitly disconnect a driver from a resolved signal in VHDL. This disconnection can be accomplished implicitly through the use of guarded signal assignments or explicitly through the use of null valued transactions in the signal assignment or through the **disconnect** statement. As disconnection operation

28. There are two delay models defined in VHDL 1076: inertial delay and transport delay. Inertial delay is used to model the physical effect of gate drive; if a semiconductor device is not driven long enough with the new value then the new value does not "stick." Inertial delay has the effect of filtering out pulses which are shorter than the indicated delay. Transmission delay is used to model the physical effect of a transmission line which passes pulses of any width without modification; the delay aspect is the latency of the transmission medium. More detailed descriptions of these delay models and their uses in modeling can be found in the literature [LSU89][-Coe89][ALG+91]

29. The notion of preemption in inertial or transport delay is derived from the truncation of the signal driver queue which occurs when an earlier event is registered. See [LSU89], pages 75-82.

requires the corresponding manipulation of a runtime data structure, it is disallowed in the context of Synchronous VHDL.

This restriction is implemented by recognizing the explicit use of null valued transactions in the signal assignment, the use of the **disconnect** statement and also by recognizing the use of guarded signal assignments which implicitly use the disconnect operation. All proscribed uses are statically obvious from the source text.

### 4.2.2 Restriction 2 - No Runtime Stack

The central feature of the reactive compilation strategy is the identification of every state variable and the placement of each in a single statically-allocated piece of storage. The automaton code generated by the reactive compiler computes by identifying the event for the current instant and by computing an output event and a next state value based on that event and the current state. The requirement that the current state be stored in explicitly materialized variables stems from the requirement that there be no indirect interpretive overhead in this output and next-state computation as would be required if intervening data structures such as runtime schedulers, memory allocators, signal queues - or a runtime stack.

It is interesting to note that the lack of a runtime stack in the final executable prevents the use of *recursive* subprograms, but does not necessarily prohibit subprogram use in the general case. One can consider that all subprograms are implicitly inlined. That is they are expanded in place as if they were but semantically-defined macros. Subprograms can be considered to be *implicitly* inlined as it is not necessary to explicitly do the inlining. However, it may be beneficial to share the implementation of a given subprogram for reasons of performance or resource consumption. Within the confines of a software simulator, this implementation sharing can be easily accomplished with a simple register-linkage calling-convention for the subprograms (*c.f.* the subroutine calling conventions of older FORTRAN dialects), while in the hardware synthesis case the sharing can be accomplished by determining a schedule for the use of shared functional units.

The following items describe constructs which are disallowed because they require a runtime stack.

**Nested Functions or Procedures**

Nested subprograms, an example of which is shown in Figure 25, are disallowed because the support required for these is explicitly stack oriented. Further they require the implicit management of an internal data structure called a *display* or *static link* in order to execute them [ASU86].

**Figure 25** Example of a Nested Procedure Requiring a Static Link

```
procedure p(x: in integer; y, z: out integer) is

   procedure q(a: in boolean; b: out integer) is
      begin
         if a then
            b := x + 1;
         else
            b := x * 10;
         end if;
   end q;

begin
   q(true, y);
   q(false, z);
end p;
```

This restriction is implemented by a simple restriction on the types of objects which may be declared in the *declaration-list* of a procedure or function. Allowing only types, variables and constants, but not subprograms to be declared there ensures that nested procedures and functions cannot occur.

**Use of Recursion**

Although the use of procedures and functions can be supported within the restrictions outlined, the requirement identified earlier is that there be no stack. This essentially requires that all subprograms are be implicitly inlined and thus the use of direct or indirect recursion cannot be supported.

This restriction is easily checked by determining if there is a cycle anywhere in the call graph of the VHDL subprograms in a process. Implementing this check for all processes in the VHDL program ensures that there is no use of direct, indirect or mutual recursion in any process.

**Dynamically-Sized Objects**

It is possible in VHDL to declare that the size of an array be determined at runtime. The array size is fixed over the life of the array, but is determined only at the time of elaboration of the block containing the declaration, or the at the time of the execution of the array slice expression.

This ability to process non-statically-sized objects results in the VHDL interpreter having to support management techniques for dynamically-sized objects. An example of the use of such is shown in Figure 26 where the use of the unconstrained type indication **bit_vector** in the function **resolve** results in an unknown size for the variable **bv** at compile time. It is only when the function is invoked that the length of the array is bound.

The restriction against dynamically sized objects can be implemented in the simple case by disallowing unconstrained type declarations and slice expressions. Thus, for lack of a way to declare variable-sized objects or an

**Figure 26** A Use of a Dynamically-Sized Array

```
function resolve(bv: in bit_vector) return bit is
   variable ret: bit;
begin
   if bv'length = 0 then
      ret := '0';
   else
      ret := bv(1);
   end if;
   return ret;
end resolve;

subtype resolved_bit is resolve bit;
```

expression operator which will produce a variable-sized result, it will be possible to have explicit size information on each data value in the program, including arrays.

This restriction will solve the problem completely but will have the unfortunate side effect of making the construction of packages containing semi-generic subprograms difficult. This is because a great deal of expressive power is derived from the ability to query an array to determine its length and range at runtime. This allows one to write routines which are implicitly parameterized about the size of the array values upon which they are called. The example shown in Figure 27 is a reasonable use of this facility in the *xor* function.

**Figure 27** A Subprogram Implicitly Parameterized over the Size of an Array

```
function "xor"(bv: in bit_vector) return bit is
   variable ret: bit := '0';
begin
   for i in bv'range loop
      if bv(i) = '0' then
         ret := not ret;
      end if;
   end loop;
   return ret;
end "xor";
```

So, there is a need for the implicit genericity afforded by the export of runtime size information back into the language. On the other hand, as stated previously, this runtime size information is problematic from the standpoint of a reactive compiler. A possible solution to this problem is to push the issue back into the library management portion of the compiler; that is, treat runtime size information as implicit genericity and force the library management portion of the compiler to generate and store copies of the subprogram which are customized for the specific size values. References to these non-generic instantiations would then be used instead of the generic version. Using this suggestion, allowances can be made for the succinctness of exposition afforded by the access to object size attributes while still remaining within the confines of the reactive compilation restrictions[30]

Also, as with the case of the example shown in Figure 27 , this restriction against unconstrained type declarations if implemented exactly as stated would have the unfortunate effect of disallowing bus resolution functions - a feature which *can* be supported in a synchronous framework. Esterel supports the concept of multiply-driven signals through the use of a *composition operator* [CIS88].[31] In the case of VHDL, although the source text does not allow for the direct observation of it, each use of the bus resolution function on a multiply-driven signal is statically bound. The compiler has access to the number of drivers for the resolved signal and can determine at compile time the size of the array which represents the vector of signal drivers.

**Locally Static Objects**

Objects in VHDL which are declared in a manner which is independent of the runtime activities are called *locally static* whereas objects which require runtime activities in order to bind all their aspects are called *globally static*[32]. Only locally static declarations are allowed in the synchronous subset for the size of a globally static object is determined by the context in which it is elaborated. This is a general principle which allows for the dynamically sized arrays as described in the previous item as a special case. As an example, consider Figure 28 in which is shown a globally static subtype declaration that requires the implementation of a runtime type checking system to ensure the consistency of the type safety rules.

**Figure 28** A Locally Static and Globally Subtype Declaration

```
procedure p(low, high: in natural) is
   subtype locally_static is integer range 0 to 10;
   subtype globally_static is integer range low to high;
begin
   ...
end p;
```

It is fairly straightforward to restrict the use of globally static declarations. The analysis required to determine whether a declaration is static in the global sense or the local sense is rather sophisticated, in the general case requiring the full power of a VHDL compiler. While it is unfortunate that this check requires so much infrastructure, the key point is that the check can still be performed at compile time.

### 4.2.3 Restriction 3 - No Heap Storage

Heap storage is required in the VHDL language by the ability to dynamically create unnamed values. These values are referred to in the usual manner: via pointers, or as they are called in the VHDL type system "access types." There is no utility for pointer types and dynamic allocation in the finite-state, statically-allocated reactive computing regime. Thus, all vestiges of the dynamic allocation facilities in VHDL are proscribed in the definition of Synchronous VHDL.

The following items describe the facilities in VHDL which allow for the processing of dynamic storage:

30. See [Hil83], Chapter 2 for an analysis of this issue in the context of Ada.

31. The "bus resolution functions" in Esterel are called "the composition operator," and are always named `*`, but may be implemented by any user-defined function with the appropriate dyadic signature. The effect and intended use of the composition operator is exactly that of a bus resolution function.

32. This is exactly backwards from the intuitive sense of these words, however see pages 7-15 in the LRM [IEEE87]. The names can be remembered by understanding that they denote how much information is required to complete the definition. Locally static definitions may be fixed at analysis time because they require less information; globally static definitions must be fixed at elaboration time because they require more global information about the design in which they reside.

**Access Types**

Pointers are the only way to access items which have been allocated in the heap. No heap in the subset's abstract simulator implies no pointers.

Disallowing the use of the access type constructor is sufficient to implement this restriction. If one cannot declare pointer-valued variables or subprograms, then the use of a heap is not needed.

**The New Operator**

Again, no heap implies that there is no need to allocate in the heap. The **new** operator is disallowed as is the use of the corresponding implicitly-declared **deallocate** procedure.

### 4.2.4 Other Restrictions

There are a few other miscellaneous restrictions which are required to ensure that the simulator maintains its finite-state aspects. In addition, these restrictions ensure that the resulting simulator does not require extra data structures to manage the state of each process. It is this extra meta-state that is important to eliminate for its existence establishes the beginnings of interpreter bias. Having an abstract simulator that does not require hidden states will ensure that the effects of interpreter bias on the resulting implementation will be minimized.

**The Time Type**

The time type is simply an artifact of the full VHDL simulator. It has no specification counterpart since we are specifying a reactive system and reactive systems are defined independently of their temporal aspects.

The removal of the time type is also convenient for it ensures that other constructs that make reference to time removed. For example the removal of the time data type ensures that the **now** function, which returns the current simulator time is removed from the synchronous subset.

**The File Type**

The **file** type constructor is only relevant in the context of a full simulator and it does not have relevance for specification. As such, its use is disallowed in the Synchronous VHDL subset.

### 4.2.5 Summary

The result of these restrictions is that the simulator can execute in finite space with the minimum amount of meta-state (none). In addition, the specification aspect of VHDL is defined operationally by the execution of a description on the abstract (finite-state) simulator that is it is simply the set of inputs and outputs that the program will participate in over its lifetime. This set of input output pairing for each instant defines a symbol set and the set of pairings over time defines a set of behaviors in the form of strings of symbols: the behavior of the program is defined in terms of a regular expression defined on the symbol set composed of input output pairings. The purpose of the reactive compiler is to derive a finite automaton for this regular expression which "executes" the program by performing state transitions and producing output events in response to input events.

It is interesting to note that the Synchronous VHDL subset is completely devoid of any references to time. What this means is that the Synchronous VHDL subset operates only at the $\Delta$-delay or unit-delay level. While it was argued that this was an extremely bad thing for the purposes of modeling because zero-time modeling has no physical correspondence, the point here is exactly the opposite. The use of a unit-delay model for specification is useful because it leaves the temporal aspects of the language out of the functional specification. In order to impress the specification onto the time axis, as is required during a simulation, the synchronous VHDL program will have to be put on a test bench like the one shown in Figure 9. As the test bench will have a vector application unit and a flight recorder which are *not* reactive, the Device Under Test (DUT) will be operated with time explicitly imposed on it from outside. Thus the DUT's environment will enforce the use of metric time on the device at simulation time.

Because the DUT, as a specification, is a reactive entity, its metric time behavior only becomes defined when the DUT is connected up to its environment. This is consistent with the notion that constraints on systems really propagate from the interfaces into the center of the design. It is also consistent with the idea of the environment constraining a system's behavior as is found in language containment verification [HK90]. Thus the reactive synchronous specification semantics of VHDL fits naturally into the existing simulation paradigm with as a Device Under Test and also into the automata-theoretic verification paradigm as a system which needs to be connected to its environment in order to be completely defined.

## 4.3 VHDL to Esterel

There is a strong correspondence between Synchronous VHDL as presented in the previous section and the Esterel language. This correspondence is shown here by relating the core language concepts of VHDL to the core language of Esterel. However, only the correspondences which motivate the full translation procedure of Chapter 5 are presented here. The correspondences shown in this section are important however for they show how a control structure in a VHDL process is converted into a control structure of an Esterel process. The full translation of a VHDL process to Esterel requires a bit more mechanism than has been defined so far since VHDL has both control-handling and data-handling aspects (*i.e.* arrays, records, scalars and other user-defined additions to the type system). Esterel on the other hand focuses on describing only control aspects of computation and has rather weak data-handling capabilities requiring some extra notation to support the data-handling capabilities of VHDL.

The essential parts of the core language of Esterel were presented in Section 3.2. The essential core language construct of VHDL is the **process** with some number of **wait** statements in it. All other legal synchronous VHDL process forms, such as a process with a sensitivity list or the dataflow forms, can be de-sugared into the basic form according to the language reference manual [IEEE87]; the basic process form is shown in Figure 29..

**Figure 29** The Structure of a Reactive VHDL Process

```
process-name:
process
   vhdl-declaration-1;
   vhdl-declaration-2;
   ...
   vhdl-declaration-M;
begin
   vhdl-statement-1;
   vhdl-statement-2;
   ...
   vhdl-statement-N;
end process process-name;
```

The correspondence between the VHDL process and the Esterel process is not exact but it is indeed close. The Esterel process is defined as a sequence of statements which can terminate whereas the VHDL process is an implicit "loop forever" construct which is required to have at least one **wait** statement in it. The structure in Esterel which corresponds naturally to the VHDL process semantics is shown in Figure 30. It has an outer "loop forever" wrapped around it to emulate the semantics of the implicit loop of the VHDL process. The important details left out of these figures are the formulation of the *declaration-list* and *statement-list* in the VHDL and Esterel versions and are dealt with later..

### 4.3.1 Syntactic Correspondences

The translation of VHDL statements however are a bit less obvious but still straightforward. They can be translated mostly in a syntax-directed manner as is described in this section. This is to say that the structure of the VHDL grammar

**Figure 30** The Corresponding Esterel Process

```
var
   strl-declaration-1,
   strl-declaration-2,
   ...
   strl-declaration-M
in
   loop
      strl-statement-1;
      strl-statement-2;
      ...
      strl-statement-N
   end
end
```

can be used to drive the translation. The syntax-directed property implies that the translation of the whole program is determined by the translation of all the parts put together [ASU86]. This is an important property for it means that the translation is "simple" and it can be performed in a formal manner, without direct knowledge of the underlying semantic meaning of each construct.

### The Entity and Module Declarations

The entity declaration in VHDL declares the interface of a design unit while the module in Esterel accomplishes the same function. There is an extra layer in the VHDL design hierarchy called the architecture. There may be many architectures for a given entity interface, so for the purposes of a translation from VHDL to Esterel, the entity and architecture become synonymous. The correspondence is illustrated by the following grammatical transformation:

```
entity e-name is
   port(interface-declaration-list);
end e-name;

architecture a-name of e-name is
   signal-declaration-list
begin
   concurrent-statement-list
end a-name;
```

```
module e-name+a-name:
interface-declaration-list

signal
   signal-declaration-list
in
   [
   || concurrent-processes
   ]
end.
```

### Signal Declarations

Signals operate essentially the same way in both VHDL and Esterel. Signals may be defined at the level of the architecture in VHDL. This so that communication between the concurrent statements in the architecture can take place. A signal may be read in any number of processes but unless it is a resolved signal, it may only appear on the left-hand side of a signal assignment in one process. The same is true of signals in Esterel. The correspondence between signals in VHDL and Esterel is best illustrated in the example above describing the syntax-directed translation of the VHDL entity and architecture into an Esterel module: the signals declared locally in the VHDL architecture are translated as a local signal block in the Esterel.

**Process and Loop Constructs**

While there is no explicit process construct within Esterel, there is an implicit one which is defined by the parallelism operator ||. The notion of process in Esterel however is much different than that of VHDL in that the Esterel process is an artifact which can be repeatedly started, executed and terminated. A process in VHDL is a permanent structural fixture of the description. Further, it has is an implicit "do forever" loop and so it never terminates.[33] The VHDL idea of process can be rendered within Esterel with the following syntax-directed translation:

| | |
|---|---|
| <pre>*p-name*:<br>process<br>   *variable-declaration-list*<br>begin<br>   *sequential-statement-list*<br>end process *p-name*;</pre> | <pre>var<br>   *variable-declaration-list*<br>in<br>   loop<br>      *sequential-statement-list*<br>   end<br>end</pre> |

This correspondence should be placed in the context of the syntax-directed translation defined for the entity-to-module translation. In VHDL the architecture is filled with processes each of which operate in parallel and so what has been defined in the Esterel translation is the same sort of structure.

**Variable Declarations**

Variables are treated in the same way in VHDL and in Esterel. In both languages shared variables are not allowed and all inter-process communication must be performed via signals. The translation of VHDL variable declarations into Esterel is accomplished using the local variable declaration construct of Esterel as shown in the example above. As there is only one place within a process to declare variables, this direct translation of a single declaration block is sufficient.

**Signal Emission Statements**

Signal emission constructs in Synchronous VHDL and in Esterel are the same as the following correspondence demonstrates:

| VHDL | Esterel |
|---|---|
| `s-name <= rhs;` | `emit s-name(rhs);` |

**The Wait and Await Statements**

The correspondence of the **wait** statement in VHDL can be obtained by using the **await** statement in Esterel as the following correspondence demonstrates:

| VHDL | Esterel |
|---|---|
| <pre>wait *on*<br>      *s-name-1*,<br>      *s-name-2*,<br>      ...<br>      *s-name-N*;</pre> | <pre>await<br>   case *s-name*-1<br>   case *s-name-2*<br>   ...<br>   case *s-name-N*<br>*end*</pre> |

33. Actually a VHDL process can be terminated through the invocation of a **wait** statement with no sensitivity list. Such a process is typically interesting only in the context of simulator initialization because once halted, the process cannot be restarted - ever. An Esterel process may be started, halted or killed by another process, and restarted as often as necessary.

### The **If** and **If**/**Presence** Statements

The treatment of conditionals in VHDL is performed in a unified manner which blurs an important distinction between statically analyzable control tests and non-static data value conditions. In VHDL, one is forced to use the same control construct - the **if** statement - to test for a transaction on a signal as well as to test a data condition. In Esterel on the other hand, there is a distinction between the control part of the language which deals with inter-process events and the data part of the language which deals with intra-process data values. There are two conditional statements in Esterel: the **if** statement for data conditions and the **presence** statement for control conditions. The following figure illustrates the two Esterel constructs and the VHDL structures to which they correspond:

| VHDL | Esterel |
|---|---|
| `if` *`data-test-condition`* `then`<br>   *`statement-list`*<br>`else`<br>   *`statement-list`*<br>`end if;` | `if` *`data-test-condition`* `then`<br>   *`statement-list`*<br>`else`<br>   *`statement-list`*<br>`end` |
| `if` *`s-name`*`'transaction then`<br>   *`statement-list`*<br>`else`<br>   *`statement-list`*<br>`end if;` | `present` *`s-name`* `then`<br>   *`statement-list`*<br>`else`<br>   *`statement-list`*<br>`end` |

### The Loop Constructs

There are various looping constructs in VHDL, all of which are phrased in terms of an iteration scheme that controls a basic loop body:

```
iteration-scheme loop
   statement-1
   statement-2
   ...
   statement-N
end loop;
```

There is a *for-loop* iteration scheme which iterates over a discrete range, there is a *while-loop* iteration scheme which iterates the loop until the termination condition is met, and finally there is the *forever-loop* iteration scheme which has no termination condition - it may only be exited through an explicit use of the **exit** statement.

The Esterel language contains no direct analog of the VHDL *for-* or *while-loop* though the same behavior can be derived through the use of idioms involving **loop**, **tag**, **if** and **exit** as is outlined in the following correspondences:

| VHDL | Esterel |
|---|---|
| loop<br>   *statement-list*<br>end loop | tag *L* in<br>   loop<br>      *statement-list*<br>   end<br>end |
| for *v* in *range* loop<br>   *statement-list*<br>end loop | tag *L* in<br>   var *v* := *range'left*: *type* in<br>      loop<br>         *statement-list*<br>         if *v* = *range'right* then<br>            exit *L*<br>         end<br>      end<br>   end<br>end |
| while *condition* loop<br>   *statement-list*<br>end loop | tag *L* in<br>   loop<br>      if *condition* then<br>         exit *L*<br>      end<br>      *statement-list*<br>   end<br>end |

As mentioned previously, a loop in the reactive model must contain a delay construct (*e.g.* an **await** or a **do/watching** in Esterel) so that the body of the loop does not execute in zero time. Were this allowed occur, it would declare that an infinite amount of computation occur in zero time (between successive events). Thus all loops in Synchronous VHDL must contain some delay, which in the case of VHDL is a use of the **wait** statement. The requirement for the execution of at least one **wait** statement on all paths across the body of a loop is not reflected in the grammatical correspondences shown above, but it must be enforced in the translation phase in order for the translation be correct.

**Treatment of Procedures and Functions**

The use of subprograms, procedures or functions, is defined only within the context of a containing process. As described previously, the treatment of subprograms handled by implicit inlining. There are restrictions on the call graph structure of the subprograms to prevent recursion and to ensure that the implicit inlining is always well-defined.

What is proposed here is simple inlining where subprograms are not translated to Esterel directly but rather they are implicitly expanded in the body of the calling process. This is an important feature not for reasons of execution performance of the final automaton but because this interpretation allows for several features in VHDL to be supported in Synchronous VHDL would not be supported in a translation to Esterel which did not use implicit inlining. These features are as follows:

- Subprograms may take signals as parameter values the same way that they can take variable values. This allows one to write general signal drivers[34] which may be placed in a package for later use.

- Subprograms may execute **wait** statements as if they were at the top-level. This allows one to write general procedures for protocol management which may be placed in a package for later use.

Thus subprograms do *not* fall into the category of syntax-directed translations because of this implicit inlining.

34. The term *signal driver* here refers to a style of model where a procedure performs a set of signal assignments. It does not refer to the structure of the same name within the VHDL simulator (that structure is not visible to the user).

# Chapter 5

# A Synchronous VHDL Simulator

How the Synchronous VHDL subset might be used in the construction of a simulator is described in this chapter. The purpose of designing a simulator based on the semantic properties of the subset is twofold. First, and most obvious, the existence of the simulator allows for the demonstration that the synchronous / reactive behavior of programs in the subset is the same as the discrete event behavior defined by the standard simulator implementation. One of the major goals of the subset definition was that the subset behave in the same way on both abstract simulator models - on the finite automata-based simulator model and on the discrete-event simulator model. Second and less obvious is that the existence of the simulator shows how the reactive and finite-state properties of the subset can actually be used to interpret a VHDL program as a specification.

In Chapter 2, the difficulty of interpreting any arbitrary VHDL program as a specification of behavior was described and the reactive model using synchronous parallelism was proposed as a way of making sense out of a process-oriented event-based description of computation. Some of the central properties of an existing reactive language, Esterel, was presented in Chapter 3 with an eye towards highlighting the similarities between the imperative processes of that language and those of VHDL. In Chapter 4 the definition of the contents of the Synchronous VHDL subset was given by showing which constructs and construct idioms in the full VHDL-1076 standard could not be supported within the finite-state reactive restriction. The previous chapters thus framed the question of what a VHDL specification subset must look like and how such a subset may be identified in a rational way.

Given the subset definition, the next question is how to make use of the properties of the subset in implementation; that is the subject of the following sections. The basic architecture of the simulator is described in Section 5.1 with an explanation of how the VHDL language is converted into a finite-automata-based executable: a simple simulator. Detail about the VHDL compilation procedure and a description of some pragmatic issues involved in the translation to Esterel are provided Section 5.2 and Section 5.3. Finally, in Section 5.4 a description of the lessons learned in this research about the properties of the VHDL language with respect to the synchronous / reactive model. In a sense, this last section is the most valuable for outlined in it, in concrete terms, are the reasons why VHDL is such a “difficult” language to interpret as a specification of system behavior.

### 5.1 Simulator Architecture

The simulator designed for this work is oriented solely at demonstrating the viability of the Synchronous VHDL subset. As described earlier in Chapter 2 the full VHDL language has a wide variety of uses in simulation, ranging from describing the model itself - the device under test (DUT) - to providing ways of programming the test stimuli system and recovering and analyzing the test results. These uses are depicted again in Figure 9. The reactive properties of the Synchronous VHDL subset imply that it is suitable for describing the DUT but not for describing the whole test bench setup shown in Figure 9. Thus the simulator described in this report provides a demonstration of the feasibility of compiling the Synchronous VHDL subset into an automata-based model of the DUT. The auxiliary features shown in Figure 9 such as test vector application and recovery which would be necessary in a product are ignored here as the standard discrete event-based implementations would suffice for those aspects of a full implementation.

The main flow of compilation in the simulator is shown in Figure 32. That figure shows how the compiler translates the textual representation of the VHDL into a control flow graph representation. This is accomplished using the standard parsing techniques [ASU86]. The result of this first phase is a set of Abstract Syntax Trees (ASTs) which represents the

**Figure 31** A Schematic of a Test Bench

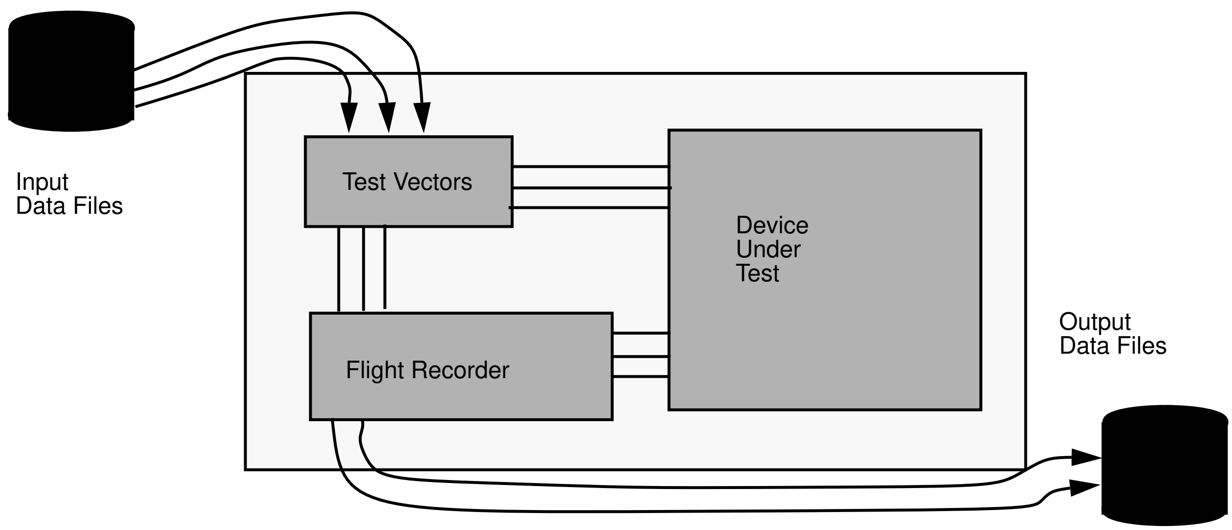
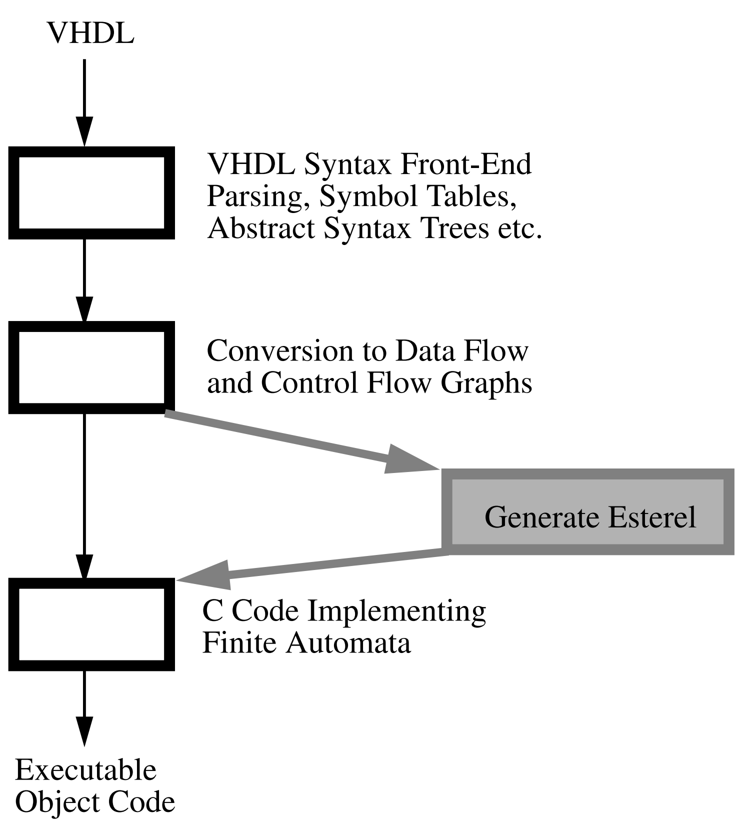



**Figure 32** Simulator Compilation Flow



VHDL source text. These tree structures are not described further in this report as they are merely a representation of the VHDL source in a data structure and are fairly uninteresting. The next phase converts the ASTs into an assembly code style intermediate form which facilitates the conversion to finite automata. From the control graph representation the goal is to extract the state graph which represents the finite-state behavior of all of the VHDL processes executing in synchrony so that the state graph can then be converted into an automata.

The process of converting from the imperative process-oriented description of the VHDL program to the state graph formulation and then to the finite automaton requires a good deal of analysis. The actual procedures for converting directly from the control flow of the intermediate representation into a finite automaton is described in Section 5.3. In lieu of implementing that procedure directly, an existing implementation of the automaton extraction procedure has been used in this work in the form of the Esterel compiler. The shaded path shown in Figure 32 indicates this use: the conversion of the intermediate representation back into the high-level form suitable for compilation by the Esterel compiler. The final result of either path is a body of C code[35] which implements the finite automaton representing the synchronous execution of the all of the VHDL processes in under the reactive model.

### 5.2 Abstract Machine Architecture

The first two phases of the compilation flow of Figure 32 is fairly standard with respect to the design of a traditional compiler [ASU86]. These phases convert grammatically structured text into a form which represents the computation as a graph of the control flow. This graph can then be used as the basis of further steps which optimize the computation or, as is the case here, extract an equivalent finite automaton. The nodes of the graph are basic blocks consisting of straight-line code with a single entry point and exit only at the end of the block. The instructions within the basic blocks represent the atomic operations that are necessary to describe a VHDL process.

The intermediate code is based on an abstract machine model that supports the operations found in VHDL programs. The textual representation of the intermediate form is that of an assembly code but this can easily be seen to be just a linear representation of a control flow graph by replacing the goto labels with pointers. The idea is that the intermediate form be a (semi) language-neutral description of a single process which has the following properties:

1. It is close enough to the high-level language that simple translation from the high-level language to it is relatively straightforward. In this case the naive code generation strategies of syntax-directed compilation are suitable [ASU86].
2. All information needed for further optimization can be represented directly in the format. That is, there should not be any need to refer to any externally defined types, variables, signals or other items in any global libraries to make sense out of the computation declared at the intermediate form level.
3. It is general enough that it can be used for multiple source languages. For example, in addition to VHDL, one might use the intermediate assembly language as an intermediate step in the compilation of Verilog or any other high-level language with the appropriate semantic properties.
4. The conversion to state machines from this format is possible. In this case, the synchronous reactive execution semantics is assumed.

#### 5.2.1 The Abstract Architecture

The abstract machine architecture is that of a number of $\infty$-register instruction processors each of which executes synchronously with respect to its environment. Computations are described as programs on these processors. Programs communicate with other programs and the external environment in general through the emission and reception of events on signals which are a broadcast medium. Thus the compilation of a network of Synchronous VHDL processes onto this

35. It should be noted here that C code is being used as the final representation of the automaton for reasons of convenience only. The popularity of the C language makes it is a very portable target language; its use here is simply to represent the next-state tables and output code of the automaton in an architecture-neutral form.

machine can be a rather direct translation from the VHDL source to the instructions using simple syntax-directed methods. Though a convenient model for compilation and optimization purposes, this abstract model is not exactly realistic; it is the role of the state graph extraction and automata generation procedure to produce a realistic representation for this abstract model on a single physical processor.

The intermediate form bears the name Non-Deterministic Abstract Machine (NDAM) though for the purposes of this work, none of the nondeterministic features were used.[36] The architecture is abstract in the sense that there is no representation of busses or other interconnect. The NDAM system describes only a set of typed registers, typed signals and computations which include transfers between the various registers and signals and synchronizations on signal events.

### 5.2.2 Describing Systems

A system is described as a number of NDAM assembly files with each assembly file describes one synchronous/reactive process. The process defines a set of registers which contain values local to the process. The process communicates with its environment through a set of formalized channels called signals. This distinction between the internal variables of the process and the externally-broadcast signal values is an important distinction because for the derivation of event patterns on the signals form the basis of the reactive compilation algorithm.

A NDAM description represents a single sequential process in exactly the same sense as a single VHDL process. The process interacts with other processes by the emission and reception of events on signals. Thus, there are instructions for emitting and awaiting events on signals. The specification of the network of processes and the linkage of their signals is not defined within the framework of a single NDAM file and so this must be implemented externally. It is expected that there be “linkage level” tools which will link multiple NDAM files and provide for scoping and renaming of signals.

### 5.2.3 The NDAM Process

Each NDAM process is described as a unit, here called a “file,” which describes the types, signals, and registers of the process as well as the instructions of the body of the process. Like a VHDL process, a NDAM process is assumed to execute forever that is it never halts.[37] A NDAM file describing a single process consists of a number of declarations which describe the value domains used in the process, the internal registers of the process, the signals with which the process communicates with other processes and of course the code body of the process.

#### Types

A type declaration defines a finite set of values which is then be used to define the domain of values that a signal or register may take. The following are examples of some simple type declaration:

```
type t(unit) 1
type t(boolean) 2
type t(int5) range -16 15
```

In this case, three types are defined: the type **unit** which has a domain of the singleton set $\{0\}$ the type **boolean** which has the domain $\{0, 1\}$ and the type **int5** which has the domain $\{-16, \ldots, 15\}$

#### Registers

Registers hold values which are local to a process. A register can be either a scalar, an array or a record. A register has a type and operations are only allowed between registers of the same type. There are register transfer operations to insert

36. In extending this work and the intermediate form on which it is based to allow for language-containment verification [Kur90] it became necessary to add constructs supporting nondeterministic control flow (goto with multiple targets) and signal value emission (signal emission with more than one value). Nondeterminism is not used in this work as VHDL does not support it; the intermediate form however allows for it due to its other uses.

37. This $\infty$-looping model of execution was chosen because it is close to the nonterminating process execution model of VHDL.

and extract values out of register and array registers. Both array and record registers are flat in the sense that there are no multi-dimensional arrays or nested records. It is expected that the high-level language compiler which produced the NDAM assembly code will have linearized all multi-dimensional arrays and flattened out all nested records. The following are some examples of register declarations:

```
register r(1) t(unit)
register r(tmp) t(boolean)
register r(value) t(int5) := 3
register r(rec) t(int5), t(boolean), t(boolean) := 3, 0, 1
register r(arr) t(boolean) 5
```

These statements declare five registers, three singleton registers, one record register and one array register.

A register declaration defines a value which persists in the process for all time. In particular, the values stored in the registers persist across **wait** instructions. The **wait** instruction in the NDAM model is analogous to the **wait** statement in VHDL and is described in further detail in the following section. It is useful in various computations however, to store values which are not needed over the lifetime of the process, but rather are needed only locally in the computation of the process. For this reason there is also a form of register declaration which indicates that the register is merely a temporary; its value will be recomputed and consumed between **wait** instructions - its value is never required to be stored across any **wait** instruction. Typically temporary registers are used to store the single-bit values required in conditionals as in the following example:

```
temporary r(vtmp) t(int5)
   r(vtmp) := r(rec).0
temporary r(tmp) t(boolean)
   r(tmp) := r(vtmp) > r(value)
   if r(tmp) goto L(1)
   goto L(2)
```

Here two temporaries are declared, one to hold the first field of the **rec** register and one to hold the single-bit condition generated by the comparison instruction.

**Signals**

Signals are used to define the external interface of a process. As with registers, signals are defined with a type that declares the domain of values which may be present on the signal. The following are some example signal declarations:

```
signal s(1) t(unit)
signal s(tmp) t(boolean)
signal s(value) t(int5) := 3
signal s(rec) t(int5), t(boolean), t(boolean) := 3, 0, 1
signal s(arr) t(boolean) 5
```

As with registers, there can be singleton signals, record signals or array signals. In this case, as with the register example there are three singleton signals, one record signal and one array signal declared.

The signal declarations do not define whether a signal is an input signal, an output signal or is used for both input and output. This information can be ascertained from the use of the signal in the instruction body of the process. A signal is an input signal if a **wait**, **present**, **presence** or **selection** instruction references it (these instructions are described in the following section). A signal is an output signal if an **emit** statement references the signal.

### Instructions

The main part of an NDAM process description is its code body. The types, signals and registers are merely declarations of the communication patterns and internal storage used by the process; the behavior of a process is described by the instructions. The instruction set consists of the usual operations which one might find on a register-register architecture:

- Assignment operations allow for the movement of values between registers. There are assignment operations to move singletons to and from the fields of record registers and also assignment operations to move singletons to and from elements of arrays. In all, there are seven variants of the basic assignment instruction:

| | |
|---|---|
| **R(*lhs*) := R(*rhs*)** | aggregate register-to-register assignment |
| **R(*lhs*) := R(*rhs*).*offset*** | record field to singleton assignment |
| **R(*lhs*).*offset* := R(*rhs*)** | record field from singleton assignment |
| **R(*lhs*) := R(*rhs*)[R(*exp*)]** | array member to singleton assignment |
| **R(*lhs*)[R(*exp*)] := R(*rhs*)** | array member from singleton assignment |
| **R(*lhs*) := R(*rhs*)[R(*low*), R(*high*)]** | array slice to array assignment |
| **R(*lhs*)[R(*low*), R(*high*)] := R(*rhs*)** | array slice from array assignment |

- Unary operations provide datapath operations involving a single source register. The unary operations supported are as follows: **abs, inc, dec, not, neg**

| | |
|---|---|
| **R (*lhs*) := *op* R(*rhs*)** | the left-hand register receives the right-hand register subjected to the ***op*** |

- Binary operations provide datapath operations involving two source registers. The binary operations supported are as follows: **and, or, nand, nor, xor, =, /=, <, <=, >, >=, +, -, *, /, mod, rem**

| | |
|---|---|
| **R(lhs) := R(*rhs-1*) *op* R(*rhs-2*)** | the left-hand register receives the value of the two right-hand registers subject to the ***op*** |

- The emit instruction transfers a value from a register onto a signal in the current instant.[38]

**emit S(*lhs*) R(*rhs-1*), R(*rhs-2*), ... R(*rhs-N*)**

- There are two attributes of a signal which can be referenced: a signal has a single value in the current instant, a signal also has a notion of presence or absence in the current instant which indicates whether or not the signal was emitted in the current instant. There are two signal reference instructions which transfer these signal attributes into a register:

| | |
|---|---|
| **R(*name*) := selection S(*name*)** | the current value of the signal is extracted[39] |
| **R(*name*) := presence S(*name*)** | a bit indicating presence or absence of the signal is transferred |

- Normal control flows sequentially through the instruction stream. There are four instructions which allow for the conditional or unconditional transfer of the control flow. The first of these is the unconditional goto instruction which

38. The emit instruction is defined in terms of nondeterministic signal emission in order to allow the NDAM intermediate form to be used in the context of language-containment verification where nondeterminism is used as a form of abstraction [Kur90]. The deterministic case of signal assigns the value of a single register to the signal in the instant.

39. The term *selection* derives from the use of this intermediate form in language-containment where the nondeterministic selection value on the signal must be resolved [Kur90]. The degenerate case of nondeterminsitic signal reference is a deterministic signal reference which has the expected behavior.

transfers the flow of control to another point in the instruction body.[40] The next two instructions provide for conditional control transfer based on the value in a register; a single-bit value in the case of the **if** instruction or a scalar value in the case of the **case** instruction. Finally, the **present** instruction effects a control-transfer based on the presence or absence of signal emission in the current instant.

**goto L(*name-1*), L(*name-2*), ..., L(*name-N*)**
**if [ not ] R(*test*) goto L(*target*)**
**case R(*key*) when *v-1-1, ... v-M-1* goto L(*target-1*)**
**...**
**when *v-1-N, ... v-M-N* goto L(*target-N*)**
**present [ not ] S(*test*) goto L(*target*)**

- The final class of instruction is the wait statement which provides for event synchronization between processes. The wait instruction, like its counterpart in the VHDL language, suspends the process until there is an event on the indicated set of signals.

**wait on S(*name-1*), S(*name-2*), ... S(*name-N*)**

With these instructions, it is possible to describe the computation of any Synchronous VHDL process in a rather simple and straightforward way. The instructions are designed to be atomic and to reference a limited number of registers. This is in much the same spirit as the restriction to three addresses in 3-address code. In this context however, it is not so much that there is a hard architectural limit on the number of values fetched, rather there is simply a desire to have an orthogonal set of primitives with which to describe the computation.

There are two important points to note in the NDAM assembly code. The first is that the compilation of the VHDL to the instruction level provides the opportunity for the traditional compiler optimizations to be performed on the process description before the extraction of the state graph. These optimizations might include constant subexpression elimination, constant folding, constant propagation and strength reduction [ASU86]. The second point to note is that the granularity of the assembly code level is very fine. Thus, the representation of a VHDL process in terms of the NDAM assembly-level instructions reduces the complexity of data manipulations such as array, record and array slice operations to simple atomic operations. As was noted previously, the Esterel language was designed to represent control and so there are correspondingly few data manipulation operations. It is expected that the data manipulations be placed in the host language and controlled by the Esterel program. Representing the VHDL process in terms of simple atomic operations, most of which can be directly represented in Esterel, greatly eases the translation process.

### 5.3 Translation of Imperative Processes To Finite Automata

The translation of synchronously parallel imperative processes into a single finite automata is defined in terms of an event-derivative semantics [BC84] where the derivative of a program with respect to an event is merely another program which behaves as the first one would have *after the event was seen.*

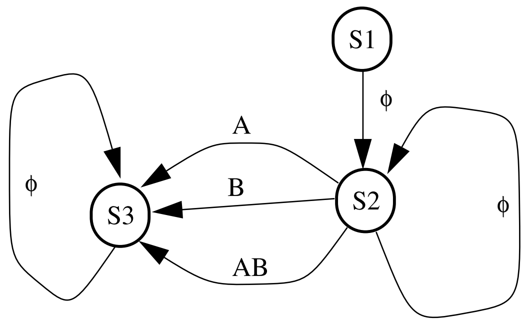
The conversion from the imperative process form to the corresponding automata merely requires transitively taking the derivative of the network of processes with respect to all possible events which can occur on the signals that are open to the outside world. Though this might seem to be a hopeless task, there are two mitigating factors: the first is that there are only a finite number of derivatives for a system with finite state [Brz62] and the second being that a derivative once seen need not be taken again for all successors of it will be the same. Thus by taking derivatives with respect to events one

40. This is the third and final case of nondeterminism in the intermediate form. The deterministic form of the goto instruction has a single target label and behaves as a jump or branch instruction.

$$\frac{\partial P}{\partial E} \Rightarrow P'$$

develops a graph of derivatives as shown in Figure 33. The conversion from the graph of derivatives to a simple state

**Figure 33** A Graph of Program Derivatives




```
module M:
input A(integer), B(integer);
output O(integer);

do
   await A;
   emit O(?A)
watching B
timeout
   emit O(?B)
end
```

graph is rather straightforward: the derivatives are each given a number *i* which becomes the name of a state. The actual derivative may then be thrown away as it no longer serves a useful purpose. The next step is the conversion from the state graph form to that of a finite automata.

The central feature which makes this whole procedure feasible is the definition of the derivative of a program with respect to an event. Unfortunately the definition of the behavior of a VHDL program with respect to an event is not so clear that one could write down a “derivative” operation directly; VHDL is defined in terms of a discrete event semantics wherein each process executes autonomously and is only awakened when there are events on the signals in its sensitivity list. The purpose though of defining the Synchronous VHDL subset was to ensure that the discrete event semantics would be restricted so that it would coincide with the synchronous semantics. So by virtue of the restrictions to finite state and reactive execution which the Synchronous VHDL subset imposes we know that there must be a derivative of a VHDL program with respect to an event

What has been achieved in this work is to use the connection between the Synchronous VHDL subset and the Esterel language to define the derivative of a VHDL program in an indirect fashion. Instead of defining the derivative of a network of VHDL processes with respect to an event directly as was done for the Esterel language [BC84], the derivative is defined indirectly through the use of the Esterel language. The translation from the Synchronous VHDL subset to the Esterel language which was outlined in Chapter 4 is the basis for this translation.

The syntax-directed translation procedure described in Chapter 4 though was described as being in the correct spirit, but as being infeasible due the limited data-manipulation facilities of the Esterel language. This limitation was circumvented through the introduction of an intermediate representation, the NDAM assembly code, which broke up the large data-manipulation operations on records, arrays and the like into simple operations that could not only be optimized using traditional compiler optimization techniques, but importantly, could be passed directly on to the Esterel language. A known algorithm is then used for recovering the high-level control structure from the control flow graph [ARZ91] and the generation of Esterel is straightforward.

## 5.4 Lessons Learned

In constructing the simulator, a number of examples were coded in the Synchronous VHDL subset and one of these, a simple key chain timer is shown in detail in Appendix A. Aside from developing an understanding of the strengths and weaknesses of the reactive computing model and synchronous parallelism, probably the most important result of this research is a set of strong reasons why the VHDL language has been so problematic as a specification language.

In this project a restriction of the VHDL language was identified under which a mathematical model of computation was seen to apply. That model, the reactive model, which assumes synchronous parallelism, allows for the specifications which are known to be useful in both the hardware and the software domain: the specification of finite state systems. There are still a number of difficulties which remain in the use of VHDL as a specification language for this class of system as described below.

### 5.4.1 VHDL Event versus VHDL Transaction

In VHDL, the notion of an event on a signal has a meaning which is especially relevant for discrete-event simulation. An event is defined to be a change in value on a signal. Unfortunately, this is a dynamic condition as a signal assignment may or may not cause an event on the signal depending on whether or not the new value assigned to the signal is the same as the old value or not. A change in value causes an event depending on the value of the variable **value** in relation to the present value of the signal **output**. A process which may or may not cause an event on its signal is shown in Figure 34. What is necessary for the derivative semantics to be soundly defined is a static condition: a condition which is true independent of the value which is assigned to the signal.

**Figure 34** May or May Not Cause an Event on **output**

```
process(value)
begin
   output <= not value;
end
```

Fortunately in VHDL, there is the notion of a VHDL *transaction* which is the execution of any signal assignment, independent of the value which is actually assigned. While the transaction is typically ignored in the context of the standard discrete-event simulation environment, it is possible to write VHDL models in terms of sensitivity to transactions instead of events. Thus Synchronous VHDL processes must focus on the transaction activity of signals instead of the event activity as is the usual case.

The lesson to be learned here is that future imperative languages which are designed to be specifications must ensure that the behaviors that they describe are statically analyzable. The event-sensitivity semantics of VHDL processes are most certainly not statically analyzable. However focusing instead on the signal transactions allows for a static analysis of the process' coordination activity is feasible.

### 5.4.2 The Flat versus Nested Process Models

A second important observation was that the flat process model of VHDL makes the description of nested behaviors particularly difficult. In particular, one often wants to describe one process as controlling one or more other processes, the master telling the slave process when to go, awaiting their completion or telling them directly when to stop. These activities are possible in VHDL's flat process scheme only with the most arduous of programming disciplines. What is required is a "go/done" protocol between the master and the slave as shown in Figure 35. Even more difficult are coordinations

**Figure 35** A Master and Slave Process Pair

```
                    signal input, output: some_data_type

master:                                slave:
process                                process(go)
begin                                     ... vars ...
   ... compute input...                begin
   go <= ping;                            output <= f(input, vars);
   wait on done;                          done <= ping;
   ... compute                         end process;
         with output...
end process;
```

between multiple processes where the master must prepare to recover control after some subset the slaves complete yet others are still computing.

The lesson to be learned here is that the specification language must allow for nested behaviors in the same way that it allows for nested structures. In VHDL an architecture can be defined structurally in terms of instances of other components, or it may be defined behaviorally in terms of a number of processes. In turn, an instance of a component is but a reference to another structure defined elsewhere; another VHDL entity/architecture pair. In contrast however, a process cannot contain another process since a process is required to be a single thread of control. This is the heart of the matter where in VHDL a structural unit can be defined in terms of other structural units or a behavioral units, but a behavioral unit is atomic.[41]

### 5.4.3 The Subtleties in the Δ-Time Models

Finally, there is the issue of the Δ-time model which has always appeared problematic from a specification point of view. The Δ-time aspect of the VHDL language has no analogy with any physical effect in the real world since nothing computes in an amount of time so small that it cannot be measured. So too the Δ-time model was problematic in the context of Synchronous VHDL but for a different reason.

Earlier in Chapter 4 it was mentioned that the Synchronous VHDL subset disallowed all references to metric time, to the use of the **after** clause in signal assignments, and to the use of the **wait for *timeval*** statement. Thus all Synchronous VHDL program are written using the Δ-delay aspect of VHDL; the idea being that the VHDL program describes the reactions that the system gives in response to events in its environment. The specification consists only of these reactions and not to any other constraints

It is interesting to note that the Esterel language effectively has a notion of Δ-delay which appears in the definition of causality. If an event *a* is said to cause event *b* in the same instant, then event *a* must have occurred slightly before event *b.* This is exactly the concept of Δ-delay - that two computations occurred at the same externally-observable time yet were ordered with respect to each other. Causality ensures that there is never a cycle in this ordering relationship. The problem with the Δ-delay system in Synchronous VHDL though is not related to causality directly for the restriction to causal signal flows ensures that there will never be a cycle of activity at the Δ-delay level. The problem is with how activity in previous deltas is *referenced*.

41. Others have observed this too [NVG91].

In this case, it is convenient to exhibit a distinction between VHDL and Esterel - the example of Figure 36 where there are two processes in each case. In both cases the first process awaits activity on the signals **S1** and **S2**. Upon seeing activity

**Figure 36** Referencing Activities in Previous Deltas

VHDL

```
process(S1, S2)
begin
   S3 <= S1 + S2;
end process;

process(S3)
begin
   if S1'transaction then
      output <= S3 * 33;
   else
      output <= S3 * 91;
   endif
end process;
```

Esterel

```
[
   every [ S1 or S2 ] do
      emit S3(?S1 + ?S2)
   end
||
   every S3 do
      present S1 then
         emit output(?S3 * 33)
      else
         emit output(?S3 * 91)
      end
   end
]
```

on either of those signals a value is computed and emitted on the signal **S3**. The distinction between the two descriptions is that the VHDL description will never see the transaction on signal **S1** because it occurred one too many deltas back; *i.e.* the VHDL **S1'transaction** only refers to signal activity in the immediately preceding delta of the current instant. The Esterel process will always see the transaction on **S1** because the presence test refers to activity in *any* preceding delta of the current instant.

Thus the lesson to be learned here is that although the notion of $\Delta$-time may seem problematic, it is actually intrinsically tied to the well-founded notion of causality. In concert with the causality however one must have a complete way of *referencing* the existence or lack of existence of previous causal events. The problem with VHDL is that the ability to reference previous causal events is restricted by the discrete-event simulator model which only allows for referencing activity in the previous simulation cycle and it does not allow for references to all activity at the current instant.

# Chapter 6

# Conclusions

The main conclusion that can be drawn from this work is that a meaningful subset of VHDL can be defined which can be interpreted in a prescriptive manner as a specification. This subset is the synchronous subset which is defined not by restricting the syntax of VHDL syntax, but rather by restricting the behavior of the simulator. The semantic restrictions then dictate what constructs and construct idioms can be used in the subset.

The simplification which reduced the VHDL simulator from a discrete-event-based paradigm to a finite automata-based paradigm imposed restrictions on the set of VHDL language constructs which could be supported within that framework. Thus the definition of synchronous VHDL in terms of its syntactic makeup is driven not by artificial constraints imposed by the capabilities of a target set of synthesis or verification tools but rather by the limits of the nature of the VHDL language semantics itself. The synchronous subset of VHDL is in this sense the largest possible subset of VHDL which can be interpreted as a specification.

The synchronous VHDL subset is reactive by virtue of the requirement that all processes in the subset must await events and respond to those events sothey may not operate autonomously as would be the case if processes were allowed to suspend themselves for specific time intervals. The reactive restriction coupled with the requirement of finite state allows for the derivative of a VHDL process with respect to an event to be defined. The derivative of a process with respect to an event is thus the behavior that the process will exhibit after seeing the event. In this way, the derivative can be thought of as a successor process.

There were fundamental restrictions imposed on VHDL processes: reactive semantics and finite state. The reactive assumption allows for the definition of the derivative of a VHDL process. Due to the fact that a process does nothing except react to events from its environment, the derivative thus becomes synonymous with the state of a process. The restriction of VHDL processes to use only finite state ensures that there are a finite number of derivatives. Thus the compilation of the synchronous VHDL subset is defined to be the extraction of these derivatives and the generation of the derivative transition graph. The derivative transition graph is synonymous with the state transition graph of a finite automata whose execution performs the computations described by the VHDL processes.

The interpretation of the synchronous VHDL subset in terms of a finite automata model is thus intrinsically tied to the definition of the derivative of a VHDL process with respect to an event. As shown in Chapter 5 the explicit definition of the derivative operation is problematic at best. Instead, in this work the derivative operation is defined indirectly through a translation from the VHDL subset to an existing synchronous language - Esterel. The derivative semantics for the Esterel language having already been described [BC84]. The final goal of this work is not a translation path from VHDL to Esterel but rather from VHDL directly to the reactive automaton. The path through Esterel used for this project was but a means to this end which was used to demonstrate the idea.

The results of this study indicate that it is indeed possible to define a meaningful subset of VHDL which is suitable for use as a specification of behavior instead of just a description of a structure containing behavioral entities. This synchronous subset of VHDL will be useful for synthesis and verification contexts where its basis in automata theory can be exploited through the use of sequential synthesis techniques and automata-theoretic verification algorithms. Further, the existence of this subset will also effect on the construction of VHDL simulators as simulators based on the execution of the finite automata models offer the potential of better performance and greater potential parallelism.

# References


ACS89 James Armstrong, Chang Cho, and Sandeep Shah, *The VHDL Validation Suite Test Development Manual*, Department of Electrical Engineering, Virginia Tech, March 1989, Revised Oct 22, 1990

ARZ91 Fran Allen, Barry K. Rosen and Kenneth Zadek, *Optimization in Compilers*, ACM Press, 1991; forthcoming.

ASU86 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1986.

Aug89 Larry M. Augustin. "Timing Models in VAL/VHDL". In I*nternational Conference on Computer-Aided Design*, 1989.

ALG+91 Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Youm Huh and Alec G. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, 1991.

BC84 Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics." In *Seminar on Concurrency*, edited by S.D. Brookes, A.W. Roscoe, and G. Winskel, pages 389-448. Springer-Verlag, 1984.

BCG86 Gérard Berry, P. Couronné and G. Gonthier. "Synchronous Programming of Reactive Systems." In *France-Japan Artificial Intelligence and Computer Science Symposium 86*, 1986.

BDS91 Fréderic Boussinot and Robert De Simone. "The Esterel Language". In *Proceedings of the IEEE*, Volume 79, No. 9, September 1991.

Ber91 Gérard Berry. "A Hardware Implementation of Pure Esterel." In *International Workshop on Formal Methods in VLSI Design*, January 9-11 1991.

Bil90 William Billowitch. *The LOGIC_SYSTEM Package - a Multivalue Logic System for VHDL*. IEEE Standard Logic Modeling Package, v2.300. Released October 24 1990.

Boo91 Booch, Grady. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, 1991.

BS86 Gérard Berry, "From Regular Expressions to Deterministic Automata," in *Theoretical Computer Science*, Volume 48, pages 117-126, Elsevier Science Publishers, 1986.

Brz62 J.A. Brzozowski. "A Survey of Regular Expressions and Their Applications." In *IRE Transactions on Electronic Computers*, Volume 11, pages 324-335, 1962.

CBH+91 R. Camposano, R.A. Bergamaschi, C.E. Haynes, M. Payer, S.M. Wu, "The IBM High-Level Synthesis System," In *High-Level VLSI Synthesis*, pages 79-104, Kluwer Academic Publishers, 1991.

Che91 Chih-Tung Chen, *VHDL2DDS: A VHDL Language to DDS Data Structure Translator*, University of Southern California, Computer Engineering Division, CEng Technical Report 91-21, July 21 1991.

CIS88 CIS Ingenierie, Agence Provence Est, Les Cardoulines B1 06560 Valbonne, France. *Esterel V3 Language Reference Manual*. This manual is available only as part of the Esterel V3 compiler system.

CJLM91 Edmund M. Clarke Jr., David E. Long, and Kenneth L. McMillan. "A Language for Compositional Specification and Verification of Finite State Hardware Controllers." In *Proceedings of the IEEE*. Volume 79, No. 9, September 1991.

Coe89 David R. Coelho. *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

GI90 Gary Imken, MCC VHDL Simulator Project, Personal Communication, November 1990.

HCD90 T. Hadley, J. Cho, and N. Dutt, *Translating BIF into VHDL: Algorithms and Examples*, U.C. Irvine, Technical Report TR 90-06, April 1990.

Hil83 Paul N. Hilfinger, *Abstraction Mechanisms and Language Design*, The MIT Press, 1983.

HK90 Zvi Har'El and Robert Kurshan. "Software for Analytical Development of Communications Protocols." In *AT&T Technical Journal*, January 1990.

HU79 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

IEEE87 *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1987.

Kur90 R. Kurshan. *Analysis of Discrete Event Coordination*. Lecture Notes in Computer Science #430. Springer-Verlag, 1990.

LiGa89 J. Lis, and D.D. Gajski, *Structured Modeling for VHDL Synthesis*, U.C. Irvine, Technical Report 89-14, June 1989.

LS75 M.E. Lesk and E. Schmidt. *LEX - A Lexical Analyzer Generator*. Computer Science Report No. 39, Bell Laboratories, Murray Hill, New Jersey, March 1975.

LSU89 Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

MCC91 The MCC VHDL Simulator, Version 3.2. *MCC VHDL System Version 3.2 General Release User's Guide*. Technical Report CAD-83-91, MCC, 1991.

NVG91 Sanjiv Narayan, Frank Vahid, Daniel Gajski, "System Specification and Synthesis with the SpecCharts Language", In *International Conference on Computer-Aided Design,* 1991.

Plo81 G.D. Plotkin. *A Structural Approach to Operational Semantics*. PhD thesis, Aarhus University, Denmark, 1981.

PO90 K.P. Parker and S. Oresjo. "A Language for Describing Boundary-Scan Devices." In *Proceedings of the 1990 International Test Conference*, pages 222-234,1990.

RoVe89 Jayanta Roy, and Ranga Vemuri, *Appropriate Usage of VHDL: The Synthesis Point of View*, University of Cincinnati, Department of Electrical and Computer Engineering, TM-DDE-89-08, 1989.

UdVe89 Kapila Udawatta and Ranga Vemuri, *A VHDL Subset for Synthesis: Reference Manual*, University of Cincinnati, Department of Electrical and Computer Engineering, TM-DDE-89-09, 1989.

Vel90 Beverly Lynn Vellandi. *Parallelism Extraction and Program Restructuring for Parallel Simulation of Digital Systems*. PhD thesis, University of Colorado, 1990.

Ver90 *Verilog Hardware Description Language Reference Manual (LRM) Version 1.0*, November 1991, Open Verilog International, Suite 408, 1016 East El Camino Real, Sunnyvale CA 94087.

VRMK91 Ranga Vemuri, Jay Roy, Paddy Mantora and Nand Kumar. *Benchmarks for High Level Synthesis*. Technical Report ECE-DDE-91-11, University of Cincinnati, June 1991. Revised November 1991.

WAV90 WAVES Analysis and Standardization Group of the Design Automation Standard Subcommittee and the Standards Coordinating Committee, Number 20 of the IEEE. *A User's Guide to the WAVES - Waveform and Vector Exchange Specification*, December 10 1990. Draft Document Version 4.4

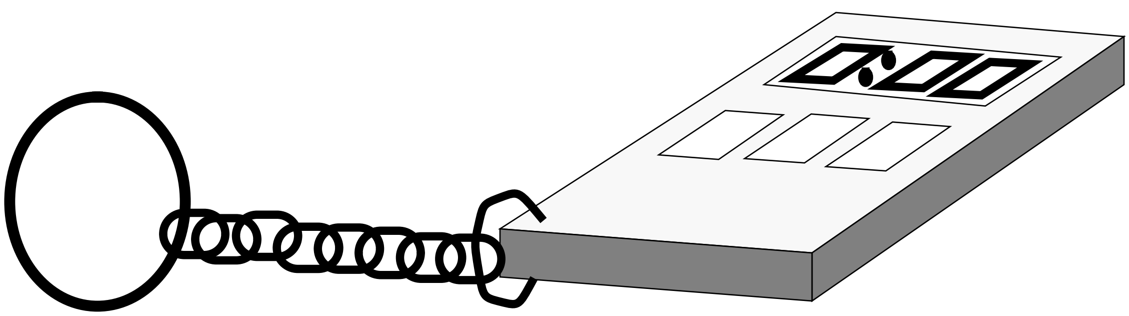
Wil90 John Willis. “Analysis, Elaboration and Formation of Directed Graphs Representing Large VHDL Designs.” In *Proceedings of the Fall 1990 VHDL User’s Group*, October 14-17 1990.

# Appendix A

# The Key-Chain Example

The Synchronous VHDL subset was tested on a number of small examples. This chapter details the use of the subset on one such example. The example is a small timer which is attached to a keychain. The idea being that when the owner parks a car at a parking meter, the timer is set for the duration of the meter. At the duration of the interval, the alarm rings to indicate that the owner is out of time and should refill the meter, or equivalently that the owner is now illegally parked and is receiving a citation. The physical appearance of the object is shown in he drawing of Figure 37.

**Figure 37** The Keychain Timer



The salient points of the part is that there are three keys labeled left-to-right as S1, S2 and S3 and there is a three and a half digit display which shows the hours and minutes.

### A.1 Specification

The specification of the part is rather simple and is given in the "user manual" is as follows:

**To Set Timer**

1. To set Hours: press S1
2. To set Minutes: press S2

**To Start Timer**

1. Once desired time is set press S3 - timer will start counting down
2. Buzzer will sound when zero is reached

**To Discontinue or Reset Timer**

1. To turn off buzzer: press S1
2. To stop timer (in mid-cycle): press S3; to restart (in mid cycle): press S3 again
3. To clear timer (during count down cycle): press S3, then press S1

**4.** To clear timer (before count down cycle has started): press S3 twice, then press S1

### A.2 Synchronous VHDL

The following VHDL design implements the keychain timer specification. The design is broken down into three entities, **countdown**, **setup** and **timer**. Each of the entities has a single architecture named **synch** which contains the processes implementing the entity.

The entity **countdown** implements the down counting behavior of the counter and consists of four independent processes: **p1**, **p2**, **p3** and **p4**. The entity setup implements the setup phase of the counter and consists of three independent processes: **p1**, **p2** and **p3**. The timer entity is the top level of the design and it consists of an instance of the **countdown** and an instance of the **setup** as well as three other processes: **master**, **beeper** and **blinker**. In the following sections, Section A.3 and Section A.4, the translation of each of these processes is given in NDAM code and Esterel respectively.

```
-- The Key Chain Timer
package types is
 type peep is (ping);
end types;

use Work.types.all;
entity countdown is
   port(go: in peep;
         decrement: in peep;
         abort: in peep;
         init_hours: in natural;
         init_minutes: in natural;
         init_seconds: in natural;
         show_hours: out natural;
         show_minutes: out natural;
         show_seconds: out natural;
         zero: out peep);
end countdown;

architecture synch of countdown is
   signal dec_min, dec_hour: peep;
   signal z_seconds, z_minutes, z_hours: boolean;
begin

   p1:
   process
      variable seconds: natural;
   begin
      wait on go'transaction;
      seconds := init_seconds;
      show_seconds <= seconds;
      z_seconds <= seconds = 0;
      loop
         wait on decrement'transaction, abort'transaction;
         if abort'transaction'event then
```

```
            exit;
         end if;
         if seconds = 0 then
            seconds := 59;
            dec_min <= ping;
         else
            seconds := seconds - 1;
         end if;
         show_seconds <= seconds;
         z_seconds <= seconds = 0;
      end loop;
   end process p1;

   p2:
   process
      variable minutes: natural;
   begin
      wait on go'transaction;
      minutes := init_minutes;
      show_minutes <= minutes;
      loop
         wait on dec_min'transaction, abort'transaction;
         if abort'transaction'event then
            exit;
         end if;
         if minutes = 0 then
            minutes := 59;
            dec_hour <= ping;
          else
            minutes := minutes - 1;
         end if;
         show_minutes <= minutes;
         z_minutes <= minutes = 0;
      end loop;
   end process p2;

   p3:
   process
      variable hours: natural;
   begin
      wait on go'transaction;
      hours := init_hours;
      show_hours <= hours;
      loop
         wait on dec_hour'transaction, abort'transaction;
         if abort'transaction'event then
            exit;
         end if;
         if hours > 0 then
```

```
            hours := hours - 1;
         end if;
         show_hours <= hours;
         z_hours <= hours = 0;
      end loop;
   end process p3;

   p4:
   process(z_seconds'transaction,
           z_minutes'transaction,
           z_hours'transaction)
   begin
      if z_seconds and z_minutes and z_hours then
         zero <= ping;
      end if;
   end process p4;

end synch;

use Work.types.all;
entity setup is
   port(go: in peep;
         s1: in peep;
         s2: in peep;
         s3: in peep;
         init_hours: in natural;
         init_minutes: in natural;
         set_hours: out natural;
         set_minutes: out natural;
         done: out peep);
begin
   assert
         not (s1'transaction'event and s2'transaction'event)
      and
         not (s2'transaction'event and s3'transaction'event)
      report "s1, s2, and s3 are not all mutually disjoint"
      severity error;
end setup;

architecture synch of setup is
   signal rollover: peep;
begin

   p1:
   process
      variable hours: natural;
   begin
      wait on go'transaction;
      hours := init_hours;
```

```
      set_hours <= hours;
      loop
         wait on s2'transaction, rollover'transaction,
            s3'transaction;
         if s3'transaction'event then
            exit;
         end if;
         hours := hours + 1;
         set_hours <= hours;
      end loop;
   end process p1;

   p2:
   process
      variable minutes: natural;
   begin
      wait on go'transaction;
      minutes := init_minutes;
      set_minutes <= minutes;
      loop
         wait on s1'transaction, s3'transaction;
         if s3'transaction'event then
            exit;
         end if;
         if minutes < 59 then
            minutes := minutes + 1;
            set_minutes <= minutes;
         else
            minutes := 0;
            rollover <= ping;
            set_minutes <= minutes;
         end if;
      end loop;
    end process p2;

   p3:
   process
   begin
      wait on go'transaction;
      wait on s3'transaction;
      done <= ping;
   end process p3;

end synch;

use Work.types.all;
entity timer is
   port(per_second: in peep;
         s1: in peep;
```

```
         s2: in peep;
         s3: in peep;
         show_hours: out natural;
         show_minutes: out natural;
         is_beeping: out boolean;
         colon_showing: out boolean);
begin
   assert
         not (s1'transaction'event and s2'transaction'event)
      and
         not (s2'transaction'event and s3'transaction'event)
      report "s1, s2, and s3 are not all mutually disjoint"
      severity error;
end timer;

architecture synch of timer is
   component setup
      port(go: in peep;
            s1: in peep;
            s2: in peep;
            s3: in peep;
            init_hours: in natural;
            init_minutes: in natural;
            set_hours: out natural;
            set_minutes: out natural;
            done: out peep);
   end component;

   for setup_stage: setup
      use entity Work.setup(synch);

   component countdown
      port(go: in peep;
            decrement: in peep;
            abort: in peep;
            init_hours: in natural;
            init_minutes: in natural;
            init_seconds: in natural;
            show_hours: out natural;
            show_minutes: out natural;
            show_seconds: out natural;
            zero: out peep);
   end component;

   for countdown_stage: countdown
      use entity Work.countdown(synch);

   signal setup_go, setup_done: peep;
   signal setup_hours, setup_minutes: natural;
```

```
    signal countdown_go, countdown_done: peep;
    signal countdown_abort, countdown_zero: peep;
    signal hours_left, minutes_left, seconds_left: natural;

    signal noshow_seconds: natural;
    signal blink_go, blink_stop: peep;
begin

    setup_stage:
    setup
       port map(go => setup_go,
                s1 => s1,
                s2 => s2,
                s3 => s3,
                init_hours => setup_hours,
                init_minutes => setup_minutes,
                set_hours => hours_left,
                set_minutes => minutes_left,
                done => setup_done);

    countdown_stage:
    countdown
       port map(go => countdown_go,
                decrement => per_second,
                abort => countdown_abort,
                init_seconds => seconds_left,
                init_minutes => minutes_left,
                init_hours => hours_left,
                show_hours => show_hours,
                show_minutes => show_minutes,
                show_seconds => noshow_seconds,
                zero => countdown_zero);

    master:
    process
    begin
       setup_hours <= 0;
       setup_minutes <= 0;
       setup_go <= ping;
       wait on setup_done;
       seconds_left <= 0;
       loop
          countdown_go <= ping;
          blink_go <= ping;
          wait on countdown_done'transaction;
          blink_stop <= ping;
          wait on s3'transaction, countdown_zero'transaction;
          -- we're now stopped awaiting further instructions
```

```
            wait on s2’transaction, s3’transaction;
            if s2’transaction’event then
               countdown_abort <= ping;
               exit;
            end if;
         end loop;
      end process master;

      beeper:
      process
      begin
         is_beeping <= FALSE;
         loop
            wait on countdown_zero’transaction;
            is_beeping <= TRUE;
            wait on s1’transaction, s2’transaction;
            is_beeping <= FALSE;
         end loop;
      end process beeper;

      blinker:
      process
         variable colon: boolean;
      begin
         colon := TRUE;
         colon_showing <= colon;
         loop
            wait on blink_go’transaction;
            loop
               wait on per_second’transaction, blink_stop’transaction;
               if per_second’transaction’event then
                  exit;
               end if;
               colon := not colon;
               colon_showing <= colon;
            end loop;
         end loop;
      end process blinker;

   end synch;
```

### A.3 NDAM Intermediate Code

The translation of each of the keychain timer’s processes into the Nondeterministic Abstract Machine Code (NDAM) assembly code representation is given in this section. Each process is described separately with each description importing the set of signals which are visible to the process in the original VHDL. The NDAM intermediate code representation in this instance is fully deterministic as VHDL does not support nondeterminism; the nondeterministic features of the NDAM intermediate code is unused.

As each of the processes is described in a self-contained manner, there is an assumption that there is a final-linkage phase which will aggregate all of the processes. In the current implementation, the Esterel compiler is used to do this, though one could envision a special-purpose "linker" tool which would perform the same function.

A careful reader will observe that most processes declare a far larger set of signals than are actually used by that process. This is acceptable given the broadcast model of communication which is implicit in the synchronous model of computation. In the following section, the translation to Esterel of each of these processes is included and only the signals which are actually used by a process are shown.

### A.3.1 Entity: Countdown, Architecture: Synch, Process: P1

```
-- The Countdown Unit (process p1)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(decrement) t(peep)
signal s(abort) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(init_seconds) t(natural60)
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
signal s(show_seconds) t(natural60)
signal s(zero) t(peep)
signal s(dec_min) t(peep)
signal s(dec_hour) t(peep)
type t(boolean) 2
signal s(z_seconds) t(boolean)
signal s(z_minutes) t(boolean)
signal s(z_hours) t(boolean)
register r(seconds) t(natural60)
constant r(natural60_0) t(natural60) := 0
constant r(natural60_59) t(natural60) := 59

L(start):
   wait on s(go)
   r(seconds) := selection s(init_seconds)
   emit s(show_seconds) r(seconds)
temporary r(tmp) t(boolean)
   r(tmp) := r(seconds) = r(natural60_0)
   emit s(z_seconds) r(tmp)
   wait on s(decrement), s(abort)
L(loop):
   present s(abort) goto L(out)
temporary r(tmp2) t(boolean)
   r(tmp2) := r(seconds) = r(natural60_0)
```

```
   if not r(tmp2) goto L(else)
L(then):
   r(seconds) := r(natural60_59)
   emit s(dec_min) r(peep)
   goto L(endif)
L(else):
   r(seconds) := dec r(seconds)
   goto L(endif)
L(endif):
   emit s(show_seconds) r(seconds)
temporary r(tmp3) t(boolean)
   r(tmp3) := r(seconds) = r(natural60_0)
   emit s(z_seconds) r(tmp3)
   wait on s(decrement), s(abort)
   goto L(loop)
L(out):
   goto L(start)
```

### A.3.2 Entity: Countdown, Architecture: Synch, Process P2

```
-- The Countdown Unit (process p2)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(decrement) t(peep)
signal s(abort) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(init_seconds) t(natural60)
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
signal s(show_seconds) t(natural20)
signal s(zero) t(peep)
signal s(dec_min) t(peep)
signal s(dec_hour) t(peep)
type t(boolean) 2
signal s(z_seconds) t(boolean)
signal s(z_minutes) t(boolean)
signal s(z_hours) t(boolean)
register r(minutes) t(natural60)
constant r(natural60_0) t(natural60) := 0
constant r(natural60_59) t(natural60) := 59

L(start):
   wait on s(go)
   r(minutes) := selection s(init_minutes)
```

```
   emit s(show_minutes) r(minutes)
   wait on s(dec_min), s(abort)
L(loop):
   present s(abort) goto L(out)
temporary r(tmp1) t(boolean)
   r(tmp1) := r(minutes) = r(natural60_0)
   if not r(tmp1) goto L(else)
L(then):
   r(minutes) := r(natural60_59)
   emit s(dec_hour) r(peep)
   goto L(endif)
L(else):
   r(minutes) := dec r(minutes)
   goto L(endif)
L(endif):
   emit s(show_minutes) r(minutes)
temporary r(tmp2) t(boolean)
   r(tmp2) := r(minutes) = r(natural60_0)
   emit s(z_minutes) r(tmp2)
   wait on s(dec_min), s(abort)
   goto L(loop)
L(out):
   goto L(start)
```

### A.3.3 Entity: Countdown, Architecture: Synch, Process P3

```
-- The Countdown Unit (process p3)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(decrement) t(peep)
signal s(abort) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(init_seconds) t(natural60)
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
signal s(show_seconds) t(natural60)
signal s(zero) t(peep)
signal s(dec_min) t(peep)
signal s(dec_hour) t(peep)
type t(boolean) 2
signal s(z_seconds) t(boolean)
signal s(z_minutes) t(boolean)
signal s(z_hours) t(boolean)
register r(hours) t(natural20)
```

```
constant r(natural20_0) t(natural20) := 0
constant r(natural20_59) t(natural20) := 19

L(start):
   wait on s(go)
   r(hours) := selection s(init_hours)
   emit s(show_hours) r(hours)
   wait on s(dec_hour), s(abort)
L(loop):
   present s(abort) goto L(out)
temporary r(tmp1) t(boolean)
   r(tmp1) := r(hours) > r(natural20_0)
   if not r(tmp1) goto L(endif)
   r(hours) := dec r(hours)
L(endif):
   emit s(show_hours) r(hours)
temporary r(tmp2) t(boolean)
   r(tmp2) := r(hours) = r(natural20_0)
   emit s(z_hours) r(tmp2)
   wait on s(dec_hour), s(abort)
   goto L(loop)
L(out):
   goto L(start)
```

### A.3.4 Entity: Countdown, Architecture: Synch, Process: P4

```
-- The Countdown Unit (process p4)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(decrement) t(peep)
signal s(abort) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(init_seconds) t(natural60)
signal s(show_hours) t(natural60)
signal s(show_minutes) t(natural60)
signal s(show_seconds) t(natural60)
signal s(zero) t(peep)
signal s(dec_min) t(peep)
signal s(dec_hour) t(peep)
type t(boolean) 2
signal s(z_seconds) t(boolean)
signal s(z_minutes) t(boolean)
signal s(z_hours) t(boolean)
```

```
L(start):
   wait on s(z_seconds), s(z_minutes), s(z_hours)
temporary r(z_seconds) t(boolean)
temporary r(z_minutes) t(boolean)
temporary r(z_hours) t(boolean)
   r(z_seconds) := selection s(z_seconds)
   r(z_minutes) := selection s(z_minutes)
   r(z_hours) := selection s(z_hours)
temporary r(tmp1) t(boolean)
temporary r(tmp2) t(boolean)
   r(tmp1) := r(z_seconds) and r(z_minutes)
   r(tmp2) := r(tmp1) and r(z_hours)
   if not r(tmp2) goto L(endif)
   emit s(zero) r(peep)
L(endif):
   goto L(start)
```

### A.3.5 Entity: Setup, Architecture: Synch, Process: P1

```
-- Setup Unit (process p1)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(set_hours) t(natural20)
signal s(set_minutes) t(natural60)
signal s(rollover) t(peep)
signal s(done) t(peep)
register r(hours) t(natural20)

L(start):
   wait on s(go)
   r(hours) := selection s(init_hours)
   emit s(set_hours) r(hours)
   wait on s(s2), s(rollover), s(s3)
L(loop):
   present s(s3) goto L(out)
   r(hours) := inc r(hours)
   emit s(set_hours) r(hours)
   wait on s(s2), s(rollover), s(s3)
   goto L(loop)
```

```
L(out):
   goto L(start)
```

### A.3.6 Entity: Setup, Architecture: Synch, Process: P2

```
-- Setup Unit (process p2)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(set_hours) t(natural20)
signal s(set_minutes) t(natural60)
signal s(rollover) t(peep)
signal s(done) t(peep)
register r(minutes) t(natural60)
constant r(natural60_0) t(natural60) := 0
constant r(natural60_59) t(natural60) := 59

L(start):
   wait on s(go)
   r(minutes) := selection s(init_minutes)
   emit s(set_minutes) r(minutes)
   wait on s(s1), s(s3)
L(loop):
   present s(s3) goto L(out)
type t(boolean) 2
temporary r(tmp) t(boolean)
   r(tmp) := r(minutes) < r(natural60_59)
   if not r(tmp) goto L(else)
L(then):
   r(minutes) := inc r(minutes)
   emit s(set_minutes) r(minutes)
   goto L(endif)
L(else):
   r(minutes) := r(natural60_0)
   emit s(rollover) r(peep)
   emit s(set_minutes) r(minutes)
   goto L(endif)
L(endif):
   wait on s(s1), s(s3)
   goto L(loop)
```

```
L(out):
   goto L(start)
```

### A.3.7 Entity: Setup, Architecture: Synch, Process: P3

```
-- Setup Unit (process p3)

type t(peep) 1
constant r(peep) t(peep) := 0
signal s(go) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(init_hours) t(natural20)
signal s(init_minutes) t(natural60)
signal s(set_hours) t(natural20)
signal s(set_minutes) t(natural60)
signal s(rollover) t(peep)
signal s(done) t(peep)

L(start):
   wait on s(go)
   wait on s(s3)
   emit s(done) r(peep)
   goto L(start)
```

### A.3.8 Entity: Timer, Architecture: Synch, Process: Master

```
-- Timer Unit (process master)

type t(peep) 1
constant r(peep) t(peep) :=0
signal s(per_second) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
type t(boolean) 2
signal s(is_beeping) t(boolean)
signal s(colon_showing) t(boolean)
signal s(setup_go) t(peep)
signal s(setup_done) t(peep)
signal s(setup_hours) t(natural20)
signal s(setup_minutes) t(natural60)
```

```
signal s(countdown_go) t(peep)
signal s(countdown_done) t(peep)
signal s(countdown_abort) t(peep)
signal s(countdown_zero) t(peep)
signal s(hours_left) t(natural20)
signal s(minutes_left) t(natural60)
signal s(seconds_left) t(natural60)
signal s(noshow_seconds) t(natural60)
signal s(blink_go) t(peep)
signal s(blink_stop) t(peep)
constant r(natural60_0) t(natural60) := 0
constant r(natural20_0) t(natural20) := 0

L(start):
   emit s(setup_hours) r(natural20_0)
   emit s(setup_minutes) r(natural60_0)
   emit s(setup_go) r(peep)
   wait on s(setup_done)
   emit s(seconds_left) r(natural60_0)
   emit s(countdown_go) r(peep)
   emit s(blink_go) r(peep)
   wait on s(countdown_done)
   emit s(blink_stop) r(peep)
   wait on s(s3), s(countdown_zero)
   wait on s(s2), s(s3)
L(loop):
   present s(s2) goto L(out)
   emit s(countdown_go) r(peep)
   emit s(blink_go) r(peep)
   wait on s(countdown_done)
   emit s(blink_stop) r(peep)
   wait on s(s3), s(countdown_zero)
   wait on s(s2), s(s3)
   goto L(loop)
L(out):
   emit s(countdown_abort) r(peep)
   goto L(start)
```

### A.3.9 Entity: Timer, Architecture: Synch, Process: Beeper

```
-- Timer Unit (process beeper)

type t(peep) 1
constant r(peep) t(peep) :=0
signal s(per_second) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
```

```
type t(natural60) 60
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
type t(boolean) 2
signal s(is_beeping) t(boolean)
signal s(colon_showing) t(boolean)
signal s(setup_go) t(peep)
signal s(setup_done) t(peep)
signal s(setup_hours) t(natural20)
signal s(setup_minutes) t(natural60)
signal s(countdown_go) t(peep)
signal s(countdown_done) t(peep)
signal s(countdown_abort) t(peep)
signal s(countdown_zero) t(peep)
signal s(hours_left) t(natural20)
signal s(minutes_left) t(natural60)
signal s(seconds_left) t(natural60)
signal s(noshow_seconds) t(natural60)
signal s(blink_go) t(peep)
signal s(blink_stop) t(peep)
constant r(TRUE) t(boolean) := 1
constant r(FALSE) t(boolean) := 0

L(start):
   emit s(is_beeping) r(FALSE)
L(loop):
   wait on s(countdown_zero)
   emit s(is_beeping) r(TRUE)
   wait on s(s1), s(s2)
   emit s(is_beeping) r(FALSE)
   goto L(loop)
```

### A.3.10 Entity: Timer, Architecture: Synch, Process: Blinker

```
-- Timer Unit (process blinker)

type t(peep) 1
constant r(peep) t(peep) :=0
signal s(per_second) t(peep)
signal s(s1) t(peep)
signal s(s2) t(peep)
signal s(s3) t(peep)
type t(natural20) 20
type t(natural60) 60
signal s(show_hours) t(natural20)
signal s(show_minutes) t(natural60)
type t(boolean) 2
signal s(is_beeping) t(boolean)
signal s(colon_showing) t(boolean)
```

```
signal s(setup_go) t(peep)
signal s(setup_done) t(peep)
signal s(setup_hours) t(natural20)
signal s(setup_minutes) t(natural60)
signal s(countdown_go) t(peep)
signal s(countdown_done) t(peep)
signal s(countdown_abort) t(peep)
signal s(countdown_zero) t(peep)
signal s(hours_left) t(natural20)
signal s(minutes_left) t(natural60)
signal s(seconds_left) t(natural60)
signal s(noshow_seconds) t(natural60)
signal s(blink_go) t(peep)
signal s(blink_stop) t(peep)
register r(colon) t(boolean)
constant r(TRUE) t(boolean) := 1
constant r(FALSE) t(boolean) := 0

L(start):
   r(colon) := r(TRUE)
   emit s(colon_showing) r(colon)
L(loop):
   wait on s(blink_go)
   emit s(is_beeping) r(TRUE)
   wait on s(per_second), s(blink_stop)
L(loop1):
   present s(per_second) goto L(out1)
   r(colon) := not r(colon)
   emit s(colon_showing) r(colon)
   wait on s(per_second), s(blink_stop)
   goto L(loop1)
L(out1):
   goto L(loop)
```

### A.4 Esterel

This section contains the final Esterel translation of the original VHDL process. With the VHDL processes translated to Esterel, all that remains to do is to link together in parallel the Esterel modules given here to form the final top-level Esterel module. This top-level module is compiled with the -**simul** option of the Esterel V.3 compiler to form the final simulator.

#### A.4.1 Entity: Countdown, Architecture: Synch, Process: P1

```
module countdown_p1:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
```

```
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
constant natural60_0: integer;
% type domain is a singleton set
% constant peep: peep;
constant natural60_59: integer;

input
   go,
   abort,
   decrement,
   init_seconds(integer);

output
   show_seconds(integer),
   z_seconds(boolean),
   dec_min;

% declarations of registers
var
   seconds := 0: integer
in
   % declarations of temporaries
   var
      tmp3 := false: boolean,
      tmp2 := false: boolean,
      tmp := false: boolean
   in
      % the process body itself
      loop
         await go;
         seconds := ?init_seconds;
         emit show_seconds(seconds);
         tmp := seconds = natural60_0;
         emit z_seconds(tmp);
          await
            case abort
            case decrement
         end;
         trap WHILE in
            loop
               present abort then
                  exit WHILE
               end;
               tmp2 := seconds = natural60_0;
               if not tmp2 then
```

```
                  seconds := seconds-1;
                  % an orphaned goto
                  nothing
               else
                  seconds := natural60_59;
                  emit dec_min;
                  % an orphaned goto
                  nothing
               end;
               emit show_seconds(seconds);
               tmp3 := seconds = natural60_0;
               emit z_seconds(tmp3);
               await
                  case abort
                  case decrement
               end;
               % an orphaned goto
               nothing
            end
         end;
         % an orphaned goto
         nothing
      end
   end
end.
```

#### A.4.2 Entity: Countdown, Architecture: Synch, Process: P2

```
module countdown_p2:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
constant natural60_0: integer;
% type domain is a singleton set
% constant peep: peep;
constant natural60_59: integer;

input
   go,
   abort,
   dec_min,
   init_minutes(integer);
```

```
output
   show_minutes(integer),
   dec_hour,
   z_minutes(boolean);

% declarations of registers
var
   minutes := 0: integer
in
   % declarations of temporaries
   var
      tmp1 := false: boolean,
      tmp2 := false: boolean
   in
      % the process body itself
      loop
         await go;
         minutes := ?init_minutes;
         emit show_minutes(minutes);
         await
            case abort
            case dec_min
         end;
         trap WHILE in
            loop
               present abort then
                  exit WHILE
               end;
               tmp1 := minutes = natural60_0;
               if not tmp1 then
                  minutes := minutes-1;
                  % an orphaned goto
                  nothing
               else
                  minutes := natural60_59;
                  emit dec_hour;
                  % an orphaned goto
                  nothing
               end;
               emit show_minutes(minutes);
               tmp2 := minutes = natural60_0;
               emit z_minutes(tmp2);
               await
                  case abort
                  case dec_min
               end;
               % an orphaned goto
               nothing
```

```
              end
          end;
          % an orphaned goto
          nothing
       end
    end
end.
```

### A.4.3 Entity: Countdown, Architecture: Synch, Process: P3

```
module countdown_p3:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
constant natural20_0: integer;
% type domain is a singleton set
% constant peep: peep;

constant natural20_59: integer;

input
   go,
   dec_hour,
   abort,
   init_hours(integer);

output
   z_hours(boolean),
   show_hours(integer);

% declarations of registers
var
   hours := 0: integer
in
   % declarations of temporaries
   var
      tmp1 := false: boolean,
      tmp2 := false: boolean
   in
      % the process body itself
      loop
         await go;
```

```
          hours := ?init_hours;
          emit show_hours(hours);
          await
             case abort
             case dec_hour
          end;
          trap WHILE in
             loop
                present abort then
                   exit WHILE
                end;
                tmp1 := hours > natural20_0;
                if not tmp1 else
                   hours := hours-1
                end;
                emit show_hours(hours);
                tmp2 := hours = natural20_0;
                emit z_hours(tmp2);
                await
                   case abort
                   case dec_hour
                end;
                % an orphaned goto
                nothing
             end
          end;
          % an orphaned goto
          nothing
       end
    end
 end.
```

### A.4.4 Entity: Countdown, Architecture: Synch, Process: P4

```
module countdown_p4:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
% type domain is a singleton set
% constant peep: peep;

input
```

```
    z_hours(boolean),
    z_minutes(boolean),
    z_seconds(boolean);

output
    zero;

% declarations of temporaries
var
    z_hours := false: boolean,
    tmp1 := false: boolean,
    z_minutes := false: boolean,
    tmp2 := false: boolean,
    z_seconds := false: boolean
in
    % the process body itself
    loop
       await
          case z_hours
          case z_minutes
          case z_seconds
       end;
       z_seconds := ?z_seconds;
       z_minutes := ?z_minutes;
       z_hours := ?z_hours;
       tmp1 := z_seconds and z_minutes;
       tmp2 := tmp1 and z_hours;
       if not tmp2 else
          emit zero
       end;
       % an orphaned goto
       nothing
    end
end.
```

### A.4.5 Entity: Setup, Architecture: Synch, Process: P1

```
module setup_p1:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% type domain is a singleton set
% type peep;

% constant declarations
% type domain is a singleton set
% constant peep: peep;
```

```
input
   go,
   s2,
   s3,
   rollover,
   init_hours(integer);

output
   set_hours(integer);

% declarations of registers
var
   hours := 0: integer
in
   % the process body itself
   loop
      await go;
      hours := ?init_hours;
      emit set_hours(hours);
      await
         case rollover
         case s2
         case s3
      end;
      trap WHILE in
         loop
            present s3 then
               exit WHILE
            end;
            hours := hours+1;
            emit set_hours(hours);
            await
               case rollover
               case s2
               case s3
            end;
            % an orphaned goto
            nothing
         end
      end;
      % an orphaned goto
      nothing
   end
end.
```

### A.4.6 Entity: Setup, Architecture; Synch, Process: P2

```
module setup_p2:
% type declarations
```

```
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
constant natural60_0: integer;
% type domain is a singleton set
% constant peep: peep;
constant natural60_59: integer;

input
   s1,
    go,
   s3,
   init_minutes(integer);

output
   set_minutes(integer),
   rollover;

% declarations of registers
var
   minutes := 0: integer
in
   % declarations of temporaries
   var
      tmp := false: boolean
   in
      % the process body itself
      loop
         await go;
         minutes := ?init_minutes;
         emit set_minutes(minutes);
         await
            case s1
            case s3
         end;
         trap WHILE in
            loop
               present s3 then
                  exit WHILE
               end;
               tmp := minutes < natural60_59;
               if not tmp then
                  minutes := natural60_0;
                  emit rollover;
```

```
                    emit set_minutes(minutes);
                    % an orphaned goto
                    nothing
                 else
                    minutes := minutes+1;
                    emit set_minutes(minutes);
                    % an orphaned goto
                    nothing
                 end;
                 await
                    case s1
                    case s3
                 end;
                 % an orphaned goto
                 nothing
              end
           end;
           % an orphaned goto
           nothing
        end
     end
  end.
```

### A.4.7 Entity: Setup, Architecture: Synch, Process: P3

```
module setup_p3:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% type domain is a singleton set
% type peep;

% constant declarations
% type domain is a singleton set
% constant peep: peep;

input
   go,
   s3;

output
   done;

% the process body itself
loop
   await go;
   await s3;
   emit done;
```

```
   % an orphaned goto
   nothing
end.
```

### A.4.8 Entity: Timer, Architecture: Synch, Process: Master

```
module timer_master:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
constant natural20_0: integer;
constant natural60_0: integer;
% type domain is a singleton set
% constant peep: peep;

input
   s2,
   s3,
   countdown_zero,
    setup_done,
   countdown_done;

output
   countdown_go,
   blink_stop,
   blink_go,
   countdown_abort,
   setup_hours(integer),
   setup_minutes(integer),
   setup_go,
   seconds_left(integer);

% the process body itself
loop
   emit setup_hours(natural20_0);
   emit setup_minutes(natural60_0);
   emit setup_go;
   await setup_done;
   emit seconds_left(natural60_0);
   emit countdown_go;
   emit blink_go;
   await countdown_done;
```

```
    emit blink_stop;
    await
       case countdown_zero
       case s3
    end;
    await
       case s2
       case s3
    end;
    trap WHILE in
       loop
          present s2 then
             exit WHILE
          end;
          emit countdown_go;
          emit blink_go;
          await countdown_done;
          emit blink_stop;
          await
             case countdown_zero
             case s3
          end;
          await
             case s2
             case s3
          end;
          % an orphaned goto
          nothing
       end
    end;
    emit countdown_abort;
    % an orphaned goto
    nothing
end.
```

### A.4.9 Entity: Timer, Architecture: Synch, Process: Beeper

```
module timer_beeper:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
% subsumed by the predefined Esterel constant
```

```
% constant false: boolean;
% type domain is a singleton set
% constant peep: peep;
% subsumed by the predefined Esterel constant
% constant true: boolean;

input
   s1,
   s2,
   countdown_zero;

output
   is_beeping(boolean);

% the process body itself
loop
   emit is_beeping(false);
   loop
      await countdown_zero;
      emit is_beeping(true);
      await
         case s1
         case s2
      end;
      emit is_beeping(false);
      % an orphaned goto
      nothing
   end
end.
```

### A.4.10 Entity: Timer, Architecture: Synch, Process: Blinker

```
module timer_blinker:

% type declarations
% subsumed by the predefined Esterel type
% type integer;
% subsumed by the predefined Esterel type
% type boolean;
% type domain is a singleton set
% type peep;

% constant declarations
% subsumed by the predefined Esterel constant
% constant false: boolean;
% type domain is a singleton set
% constant peep: peep;
% subsumed by the predefined Esterel constant
% constant true: boolean;
```

```
input
   per_second,
   blink_stop,
   blink_go;

output
   is_beeping(boolean),
   colon_showing(boolean);

% declarations of registers
var
   colon := false: boolean
in
   % the process body itself
   colon := true;
   emit colon_showing(colon);
   loop
      await blink_go;
      emit is_beeping(true);
      await
         case blink_stop
         case per_second
      end;
      trap WHILE in
         loop
            present per_second then
               exit WHILE
            end;
            colon := not colon;
            emit colon_showing(colon);
            await
               case blink_stop
               case per_second
            end;
            % an orphaned goto
            nothing
         end
      end;
      % an orphaned goto
      nothing
   end
end.
```

# Appendix B

# The Synchronous VHDL Subset

This appendix documents the grammatical aspects of the Synchronous VHDL subset. There are two sections: the first section, Section B.1, lists the constructs which are included in the VHDL subset while the second section, Section B.2 describes the constructs which are not supported in the subset.

The grammatical structure of the subset is defined without reference to the static correctness issues mentioned in Chapter 4. It should be noted that the grammatical structure documented in this appendix is much less important than the consistency issues described in the main body. This is because even within the grammatical structure shown in Section B.1 one can write nonsense programs through the use of non-causal signal assignments, recursion within subprograms, uses of unconstrained array references and the like. The following sections then should be understood as a starting point for the analyses described in Chapter 4. That is all legal Synchronous VHDL descriptions must fall within the grammatical structure described in Section B.1 *and* they must not use any of the constructs described in Section B.2. In addition however, legal programs just also obey the further restrictions documented in Chapter 4.

## B.1 Supported Constructs

The following BNF describes the grammatical structure of Synchronous VHDL Subset:[42]

```
abstract_literal
   :  decimal_literal
   |  based_literal
   ;

actual_designator
   :  expression
   |  signal_name
   |  variable_name
   |  OPEN
   ;

actual_parameter_part
   :  parameter_association_list
   ;

actual_part
   :  actual_designator
```

42. The BNF notation used here is a modified form of that which is found in Appendix A of VHDL-1076 Language Reference Manual (LRM) [IEEE87]. The notation used there documents the structure of VHDL language but is not in and of itself useful because its loose notation makes it unsuitable for use in an LR(1) parser-generator such as **yacc**. The description presented here is a modified form of the Appendix A presentation, modified to be accepted by LR(1) parser generators. It is interesting to note that Appendix A of the LRM is actually *not* part of the 1076 standard itself; it is provided for informational purposes only ([IEEE87] page A-1). No part of the standard actually documents the grammatical structure of the VHDL language.

```
    | function_name '(' actual_designatbor ')'
    ;

adding_operator
    :  '+'
    |  '-'
    |  '&'
    ;

aggregate
    :  '(' _element_association_list ')'
    ;

_element_association_list
    :  element_association
    |  _element_association_list ',' element_association
    ;

alias_declaration
    :  ALIAS identifier ':' subtype_indication IS name
    ;

architecture_body
    :  ARCHITECTURE identifier OF entity_name IS
       architecture_declarative_part
       BEGIN
       architecture_statement_part
       END architecture_optional_simple_name ';'

architecture_declarative_part
    :   /* NULL */
    |   architecture_declarative_part block_declarative_item
    ;

architecture_statement_part
    :  /* NULL */
    |  architecture_statement_part concurrent_statement
    ;

array_type_definition
    :  unconstrained_array_definition
    |  constrained_array_definition
    ;

assertion_statement
    :  ASSERT condition
       _optional_report
       _optional_severity ';'
    ;
```

```
association_element
    :  formal_part EQGR actual_part
    |  actual_part
    ;

association_list
    :  association_element
    |  association_list ',' association_element
    ;

attribute_declaration
    :  ATTRIBUTE identifier ':' type_mark ';'
    ;

attribute_designator
    :  attribute_simple_name
    ;

attribute_name
    :  prefix '\'' attribute_designator
         static_optional_paren_expression
    ;

attribute_specification
    :  ATTRIBUTE attribute_designator OF
       entity_specification IS expression ';'
    ;

base
    :  integer
    ;

base_specifier
    :  'B'
    |  'O'
    |  'X'
    ;

base_unit_declaration
    :  identifier ';'
    ;

/*
 * These are handled in the scanner directly
 *
 * based_integer: extended_digit { [ underline ] extended_digit }
 * based_literal: base '#' based_integer
 *                    [ '.' based_integer ] '#' exponent
 * basic_character: basic_graphic_character | format_effector
```

```
 * basic_graphic_character: upper_case_letter | digit |
 *                            special_character | space_character
 */

binding_indication
   :  entity_aspect
      _optional_generic_map_aspect
            _optional_port_map_aspect
   ;

/*
 * These are handled in the scanner
 *
 * bit_string_literal: base_specifier '"' bit_value '"'
 * bit_value: extended_digit {[ underline ] extended_digit }
 */

block_configuration
   :  FOR block_specification
      _use_clause_list
      _configuration_item_list
      END FOR ';'

block_declarative_item
   :  subprogram_declaration
   |  subprogram_body
   |  type_declaration
   |  subtype_declaration
   |  constant_declaration
   |  signal_declaration
   |  file_declaration
   |  alias_declaration
   |  component_declaration
   |  attribute_declaration
   |  attribute_specification
   |  configuration_specification
   |  disconnection_specification
   |  use_clause
   ;

block_declarative_part
   :  block_declarative_item
   |  block_declarative_part block_declarative_item
   ;

block_header
   :  optional_generic_clause_generic_map
      optional_port_clause_port_map
   ;
```

```
block_specification
   :  name
   |  label _optional_index_specification
   ;

block_statement
   :  label ‘:’
      BLOCK optional_paren_expression
      block_header
      block_declarative_part
      BEGIN
      block_statement_part
      END BLOCK label ‘;’
   ;

block_statement_part
   :  concurrent_statement
   |  block_statement_part concurrent_statement
   ;

case_statement
   :  CASE expression IS
      _case_statement_alternative_list
      END  CASE ‘;’
   ;

case_statement_alternative
   :  WHEN choices EQGR sequence_of_statements
   ;

_case_statement_alternative_list
   :  case_statement_alternative
   |  _case_statement_alternative_list case_statement_alternative
   ;

/*
 * These are handled by the scanner
 *
 * character_literal: ‘\’’ graphic_character ‘\’’
 */

choice
   :  simple_expression
   |  discrete_range
   |  simple_name
   |  OTHERS
   ;
```

```
choices
   :  choice
   |  choices '|' choice
   ;

component_configuration
   :  FOR component_specification
      _optional_use_binding_indication
      _optional_block_configuration
      END FOR ';'
   ;

component_declaration
    : COMPONENT identifier
      _optional_generic_clause
      _optional_port_clause
      END  COMPONENT ';'
    ;

component_instantiation_statement
   :  label ':'
      name
      _optional_generic_map_aspect
      _optional_port_map_aspect ';'
   ;

component_specification
   :  instantiation_list ':' name
   ;

composite_type_definition
   :  array_type_definition
   |  record_type_definition
   ;

concurrent_assertion_statement
   :  _optional_label_colon assertion_statement
   ;

concurrent_procedure_call
   :  _optional_label_colon procedure_call_statement
   ;

concurrent_signal_assignment_statement
   :  _optional_label_colon conditional_signal_assignment
   |  _optional_label_colon selected_signal_assignment
   ;
```

```
concurrent_statement
    :  block_statement
    |  process_statement
    |  concurrent_procedure_call
    |  concurrent_assertion_statement
    |  concurrent_signal_assignment_statement
    |  component_instantiation_statement
    |  generate_statement
    ;

condition
    :  expression
    ;

condition_clause
    :  UNTIL condition
    ;

conditional_signal_assignment
    :  target LTEQ options conditional_waveforms ';'
    ;

conditional_waveforms
    :  _waveform_when_condition_else_list
       waveform
    ;

configuration_declaration
    :  CONFIGURATION identifier OF name IS
       configuration_declarative_part
       block_configuration
       END _optional_simple_name ';'
    ;

configuration_declarative_part
    :  use_clause
    |  attribute_specification
    ;

configuration_declarative_part
    :  configuration_declarative_item
    |  configuration_declarative_part configuration_declarative_item
    ;

configuration_item
    :  block_configuration
    |  component_configuration
    ;
```

```
configuration_specification
   :  FOR component_specification USE binding_indication ';'
   ;

constant_declaration
   :  CONSTANT identifier_list ':'
      subtype_indication _optional_initial_value ';'
   ;

constrained_array_definition
   :  ARRAY index_constraint OF subtype_indication
   ;

constraint
   :  range_constraint
   |  index_constraint
   ;

context_clause
   :  context_item
   |  context_clause context_item
   ;

context_item
   :  library_clause
   |  use_clause
   ;

/*
 * This is handled in the scanner
 *
 * decimal_literal: integer [ . integer ] [ exponent ]
 */

declaration
   :  type_declaration
   |  subtype_declaration
   |  object_declaration
   |  file_declaration
   |  interface_declaration
   |  alias_declaration
   |  attribute_declaration
   |  component_declaration
   |  entity_declaration
   |  configuration_declaration
   |  subprogram_declaration
   |  package_declaration
   ;
```

```
design_file
    :  design_unit
    |  design_file design_unit
    ;

design_unit
    :  context_clause library_unit
    ;

designator
    :  identifier
    |  operator_symbol
    ;

direction
    :  TO
    |  DOWNTO
    ;

discrete_range
    :  subtype_indication
    |  range
    ;

element_association
    :  _optional_choices_eqgr expression
    ;

element_declaration
    :  identifier_list ':' element_subtype_definition ';'
    ;

element_subtype_definition
    :  subtype_indication
    ;

entity_aspect
    :  ENTITY name _optional_identifier
    |  CONFIGURATION name
    |  OPEN
    ;

entity_class
    :  ENTITY
    |  ARCHITECTURE
    |  CONFIGURATION
    |  PROCEDURE
    |  FUNCTION
    |  PACKAGE
```

```
    |  TYPE
    |  SUBTYPE
    |  CONSTANT
    |  SIGNAL
    |  VARIABLE
    |  COMPONENT
    |  LABEL
    ;

entity_declaration
    :  ENTITY identifier IS
       entity_header
       entity_declarative_part
       _optional_entity_body
       END simple_name ‘;’
    ;

entity_declarative_item
    :  subprogram_declaration
    |  subprogram_body
    |  type_declaration
    |  subtype_declaration
    |  constant_declaration
    |  signal_declaration
    |  file_declaration
    |  alias_declaration
    |  attribute_declaration
    |  attribute_specification
    |  disconnection_specification
    |  use_clause
    ;

entity_declarative_part
    :  entity_declarative_item
    |  entity_declarative_part entity_declarative_item
    ;

entity_designator
    :  simple_name
    |  operator_symbol
    ;

entity_header
    :  _optional_generic_clause _optional_port_clause
    ;

entity_name_list
    :  entity_designator entity_designator_list
    |  OTHERS
```

```
    | ALL
    ;

entity_designator_list
    :  entity_designator
    |  entity_designator_list ‘,’ entity_designator
    ;

entity_specification
    :  entity_name_list ‘:’ entity_class
    ;

entity_statement
    :  concurrent_assertion_statement
    |  concurrent_procedure_call
    |  process_statement
    ;

entity_statement_part
    :  entity_statement
    |  entity_statement_part entity_statement
    ;

enumeration_literal
    :  identifier
    |  character_literal
    ;

enumeration_type_definition
    :  ‘(‘ enumeration_literal_list ‘)’
    ;

enumeration_literal_list
    :  enumeration_literal
    |  enumeration_literal_list ‘,’ enumeration_literal
    ;

exit_statement
    :  EXIT _optional_label _optional_when_condition ‘;’
    ;

/*
 * These will be taken care of by the scanner
 *
 * exponent: E [ + ] integer | E - integer
 */

expression
    :  relation _and_relation_list
```

```
    |  relation _or_relation_list
    |  relation _xor_relation_list
    |  relation NAND relation
    |  relation NOR relation
    ;

_and_relation_list
    :  /* NULL */
    |  AND relation _and_relation_list
    ;

_or_relation_list
    :  /* NULL */
    |  OR relation _or_relation_list
    ;

_xor_relation_list
    :  /* NULL */
    |  XOR relation _xor_relation_list
    ;

/*
 * These will be taken care of by the scanner
 *
 * extended_digit: digit | letter
 */

factor
    :  primary _optional_exponentiation
    |  ABS primary
    |  NOT primary
    ;

floating_type_definition
    :  range_constraint
    ;

formal_designator
    :  name
    ;

formal_parameter_list
    :  interface_list
    ;

formal_part
    :  formal_designator
    |  name '(' formal_designator ')'
    ;
```

```
full_type_declaration
   :  TYPE identifier IS type_definition
   ;

function_call
   :  name _optional_paren_actual_parameter_part
   ;

generate_statement
   :  label ‘:’
      generate_scheme GENERATE
      _concurrent_statement_list
      END GENERATE _optional_label ‘;’
   ;

concurrent_statement_list
   :  /* NULL */
   |  _concurrent_statement_list concurrent_statement
   ;

generation_scheme
   :  FOR parameter_specification
   |  IF condition
   ;

generic_clause
   :  GENERIC ‘(‘ generic_list ‘)’ ‘;’
   ;

generic_list
   :  interface_list
   ;

generic_map_aspect
   :  GENERIC MAP ‘(‘ association_list ‘)’
   ;



/*
 * The scanner will take care of this
 *
 * graphic_character: basic_graphic_character |
 *             lower_case_character | other_special_character
 */

guarded_signal_specification
   :  signal_list ‘:’ type_mark
   ;
```

```
/*
 * The scanner will take care of this
 *
 * identifier: letter { [ underline ] letter_or_digit }
 */

identifier_list
   :  identifier
   |  identifier_list ',' identifier
   ;

if_statement
   :  IF condition THEN
      sequence_of_statements
      _elsif_list
      _optional_else
      END IF ';'
   ;

_elsif_list
   :  /* NULL */
   |  _elsif_list ELSIF condition THEN
      sequence_of_statements
   ;

incomplete_type_declaration
   :  TYPE identifier ';'
   ;

index_constraint
   :  '(' discrete_range_list ')'
   ;

discrete_range_list
   :  discrete_range
   |  discrete_range_list ',' discrete_range
   ;

index_specification
   :  discrete_range
   |  expression
   ;

index_subtype_definition
   :  type_mark RANGE LEGR
   ;
```

```
indexed_name
   :  prefix `(` expression_list `)'
   ;

expression_list
   :  expression
   |  expression_list `,' expression
   ;

instantiation_list
   :  label_list
   |  OTHERS
   |  ALL
   ;

label_list
   :  label
   |  label_list `,' label
   ;

/*
 * These will be taken care of by the scanner
 *
 * integer: digit { [ underline ] digit }
 */

integer_type_definition
   :  range_constraint
   ;

interface_constant_declaration
   :  _optional_constant identifier_list `:'
      _optional_in subtype_indication
      _optional_initial_value
   ;

interface_declaration
   :  interface_constant_declaration
   |  interface_signal_declaration
   |  interface_variable_declaration
   ;

interface_element
   :  interface_declaration
   ;

interface_list
   :  interface_element
```

```
    |  interface_list ';' interface_element
    ;

interface_signal_declaration
    :  _optional_signal identifier_list ':'
       _optional_mode subtype_indication
       _optional_bus _optional_initial_value
    ;

interface_variable_declaration
    :  _optional_variable identifier_list ':'
       _optional_mode subtype_indication
       _optional_intial_value
    ;

iteration_scheme
    :  WHILE condition
    |  FOR parameter_specification
    ;

label
    :  identifier
    ;

/*
 * The scanner will take care of these
 *
 * letter: upper_case_letter | lower_case_letter
 * letter_or_digit: letter | digit
 */

library_clause
    :  LIBRARY logical_name_list ';'
    ;

library_unit
    :  primary_unit
    |  secondary_unit
    ;

literal
    :  numeric_literal
    |  enumeration_literal
    |  string_literal
    |  bit_string_literal
    |  NULL
    ;
```

```
logical_name
    :  identifier
    ;

logical_name_list
    :  logical_name
    |  logical_name_list ',' logical_name
    ;

logical_operator
    :  AND
    |  OR
    |  NAND
    |  NOR
    |  XOR
    ;

loop_statement
    :  _optional_label_colon
       _optional_iteration_scheme LOOP
       sequence_of_statements
       END LOOP _optional_label ';'
    ;

miscellaneous_operator
    :  STARSTAR
    |  ABS
    |  NOT
    ;

mode
    :  IN
    |  OUT
    |  INOUT
    |  BUFFER
    |  LINKAGE
    ;

multiplying_operator
    :  '*'
    |  '/'
    |  MOD
    |  REM
    ;

name
    :  simple_name
    |  operator_symbol
    |  selected_name
```

```
    | indexed_name
    | slice_name
    | attribute_name
    ;

next_statement
    : NEXT _optional_label _optional_when_condition
    ;

null_statement
    : NULL
    ;

numeric_literal
    : abstract_literal
    | physical_literal
    ;

object_declaration
    : constant_declaration
    | signal_declaration
    | variable_declaration
    ;

operator_symbol
    : string_literal
    ;

options
    : _optional_guarded _optional_transport
    ;

package_body
    : PACKAGE BODY simple_name IS
      package_body_declarative_part
      END _optional_simple_name ';'
    ;

package_body_declarative_item
    : subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | file_declaration
    | alias_declaration
    | use_clause
    ;
```

```
package_body_declarative_part
    :  /* NULL */
    |  package_body_declarative_part package_body_declarative_item
    ;

package_declaration
    :  PACKAGE identifier IS
       package_declarative_part
       END _optional_simple_name ';'
    ;

package_declarative_item
    :  subprogram_declaration
    |  type_declaration
    |  subtype_declaration
    |  constant_declaration
    |  signal_declaration
    |  file_declaration
    |  alias_declaration
    |  component_declaration
    |  attribute_declaration
    |  attribute_specification
    |  disconnection_specification
    |  use_clause
    ;

package_declarative_part
    :  package_declarative_item
    |  package_declarative_part package_declarative_item
    ;

parameter_specification
    :  identifier IN discrete_range
    ;

physical_literal
    :  _optional_abstract_literal name
    ;

physical_type_definition
    :  range_constraint
       UNITS
       _unit_declaration_list
       END  UNITS
    ;

_unit_declaration_list
    :  base_unit_declaration
```

```
    |  _unit_declaration_list secondary_unit_declaration
    ;

port_clause
    :  PORT '(' port_list ')'
    ;

port_list
    :  interface_list
    ;

port_map_aspect
    :  PORT MAP '(' association_list ')'
    ;

prefix
    :  name
    |  function_call
    ;

primary
    :  name
    |  literal
    |  aggregate
    |  function_call
    |  qualified_expression
    |  type_conversion
    |  allocator
    |  '(' expression ')'
    ;

primary_unit
    :  entity_declaration
    |  configuration_declaration
    |  package_declaration
    ;

procedure_call_statement
    :  name _optional_paren_actual_parameter_part
    ;

process_declarative_item
    :  subprogram_declaration
    |  subprogram_body
    |  type_declaration
    |  subtype_declaration
    |  constant_declaration
    |  variable_declaration
    |  file_declaration
```

```
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    ;

process_declarative_part
    : process_declarative_item
    | process_declarative_part process_declarative_item
    ;

process_statement
    : _optional_label_colon
      PROCESS _optional_sensitivity_list
      process_declarative_part
      BEGIN
      process_statement_part
      END _optional_label ';'
    ;

process_statement_part
    : sequential_statement
    | process_statement_part sequential_statement
    ;

qualified_expression
    : type_mark '\'' '(' expression ')'
    | type_mark '\'' aggregate
    ;

range
    : attribute_name
    | simple_expression direction simple_expression
    ;

range_constraint
    : RANGE range
    ;

record_type_definition
    : RECORD
      _element_declaration_list
      END RECORD
    ;

_element_declaration_list
    : element_declaration
    | _element_declaration_list element_declaration
    ;
```

```
relation
   :  simple_expression
      _optional_relational_operator_simple_expression
   ;

relational_operator
   :  '='
   |  NOTEQ
   |  '<'
   |  LEQ
   |  '>'
   |  GEQ
   ;

return_statement
   :  RETURN _optional_expression
   ;

scalar_type_definition
   :  enumeration_type_definition
   |  integer_type_definition
   |  floating_type_definition
   |  physical_type_definition
   ;

secondary_unit
   :  architecture_body
   |  package_body
   ;

secondary_unit_declaration
   :  identifier '=' physical_literal ';'
   ;

selected_name
   :  prefix '.' suffix
   ;

selected_signal_assignment
   :  WITH expression SELECT
      target LE options
      selected_waveforms ';'
   ;

selected_waveforms
   :  _waveform_when_choices_list
      _waveform_when_choices
   ;
```

```
_waveform_when_choices
    :  /* NULL */
    |  _waveform_when_choices _waveform_when_choices ‘,’
    ;

_waveform_when_choices
    :  waveform WHEN choices
    ;

sensitivity_clause
    :  ON sensitivity_list
    ;

sensitivity_list
    :  _name_list
    ;

_name_list
    :  name
    |  _name_list ‘,’ name
    ;

sequence_of_statements
    :  sequential_statement
    |  sequential_of_statements sequential_statement
    ;

sequential_statement
    :  wait_statement
    |  assertion_statement
    |  signal_assignment_statement
    |  variable_assignment_statement
    |  procedure_call_statement
    |  if_statement
    |  case_statement
    |  loop_statement
    |  next_statement
    |  exit_statement
    |  return_statement
    |  null_statement
    ;

sign
    :  ‘+’
    |  ‘-’
    ;
```

```
signal_assignment_statement
   :  target GEQ _optional_transport waveform ';'
   ;

signal_declaration
   :  SIGNAL identifier_list ':'
      subtype_indication _optional_signal_kind
      _optional_initial_value ';'
   ;

signal_kind
   :  REGISTER
   |  BUS
   ;

signal_list
   :  _name_list
   |  OTHERS
   |  ALL
   ;

simple_expression
   :  _optional_sign term _adding_operator_term_list
   ;

_adding_operator_term_list
   :  /* NULL */
   |  _adding_operator_term_list adding_operator term
   ;

simple_name
   :  identifier
   ;

slice_name
   :  prefix '(' dixcrete_range ')'
   ;

/*
 * The scanner is going to take care of these
 *
 * string_literal: '"' { graphic_character } '"'
 */

subprogram_body
   :  subprogram_specification IS
      subprogram_declarative_part
      BEGIN
      subprogram_statement_part
```

```
        END _optional_designator ';'
    ;

subprogram_declaration
    :   subprogram_specification ';'
    ;

subprogram_declarative_item
    :   subprogram_declaration
    |   subprogram_body
    |   type_declaration
    |   subtype_declaration
    |   constant_declaration
    |   variable_declaration
    |   file_declaration
    |   alias_declaration
    |   attribute_declaration
    |   attribute_specification
    |   use_clause
    ;

subprogram_declarative_part
    :   subprogram_declarative_item
    |   subprogram_declarative_part subprogram_declarative_item
    ;

subprogram_specification
    :   PROCEDURE designator
         _optional_paren_formal_parameter_list
    |   FUNCTION designator
         _optional_paren_formal_parameter_list RETURN type_mark
    ;

subprogram_statement_part
    :   _sequential_statement_list
    ;

subtype_declaration
    :   SUBTYPE identifier IS subtype_indication ';'
    ;

subtype_indication
    :   _optional_name type_mark _optional_constraint
    ;

suffix
    :   simple_name
    |   character_literal
    |   operator_symbol
```

```
    |  ALL
    ;

target
    :  name
    |  aggregate
    ;

term
    :  factor _multiplying_operator_factor_list
    ;

_multiplying_operator_factor_list
    :  /* NULL */
    |  _multiplying_operator_factor_list multiplying_operator_factor
    ;

timeout_clause
    :  FOR expression
    ;

type_conversion
    :  type_mark `(` expression `)'
    ;

type_declaration
    :  full_type_declaration
    |  incomplete_type_declaration
    ;

type_definition
    :  scalar_type_definition
    |  composite_type_definition
    |  access_type_definition
    |  file_type_definition
    ;

type_mark
    :  name
    ;

unconstrained_array_definition
    :  ARRAY `(` _index_subtype_definition_list `)'
       OF subtype_indication
    ;

_index_subtype_definition_list
    :  index_subtype_definition
```

```
    |  index_subtype_definition_list ',' index_subtype_definition
    ;

use_clause
    :  USE _selected_name_list
    ;

_selected_name_list
    :  selected_name
    |  _selected_name_list ',' selected_name
    ;

variable_assignment_statement
    :  target COLONEQ expression ';'
    ;

variable_declaration
    :  VARIABLE identifier_list ':' subtype_indication
        _optional_initial_value ';'
    ;

wait_statement
    :  WAIT _optional_sensitivity_clause
       _optional_condition_clause
       _optional_timeout_clause ';'
    ;

_optional_abstract_literal
    :   /* NULL */
    |   abstract_literal
    ;

_optional_block_configuration
    :  /* NULL */
    |  block_configuration
    ;

_optional_bus
    :  /* NULL */
    |  BUS
    ;

_optional_choices_eqgr
    :  choices EQGR
    ;

_optional_condition_clause
    :  /* NULL */
```

```
    |  condition_clause
    ;

_optional_constant
    :  /* NULL */
    |  CONSTANT
    ;

_optional_constraint
    :  /* NULL */
    |  constraint
    ;

_optional_designator
    :  /* NULL */
    |  designator
    ;

_optional_else
    :  /* NULL */
    |  ELSE sequence_of_statements
    ;

_optional_entity_body
    :  /* NULL */
    |  BEGIN entity_statement_part
    ;

_optional_exponentiation
    :  /* NULL */
    |  STARSTAR primary
    ;

_optional_generic_clause
    :  /* NULL */
    |  generic_clause
    ;

_optional_generic_clause_generic_map_aspect
    :  /* NULL */
    |  generic_clause generic_map_aspect ';'
    ;

_optional_generic_map_aspect
    :  /* NULL */
    |  generic_map_aspect
    ;
```

```
_optional_generic_map_aspect
    :  /* NULL */
    |  generic_map_aspect
    ;

_optional_guarded
    :  /* NULL */
    |  GUARDED
    ;

_optional_identifier
    :  /* NULL */
    |  identifier
    ;

_optional_in
    :  /* NULL */
    |  IN
    ;

_optional_index_specification
    :  /* NULL */
    |  ‘(‘ index_specification ‘)’
    ;

_optional_initial_value
    :  /* NULL*/
    |  COLONEQ expression
    ;

    _optional_iteration_scheme
    :  /* NULL*/
    |  iteration_scheme
    ;

_optional_label_colon
    :  /* NULL */
    |  label ‘:’
    ;

_optional_label
    :  /* NULL */
    |  label
    ;

_optional_mode
    :  /* NULL */
    |  mode
    ;
```

```
_optional_name
   :  /* NULL */
   |  name
   ;

_optional_paren_actual_parameter_part
   :  /* NULL */
   |  `(` actual_parameter_part `)'
   ;

_optional_paren_expression
   :  /* NULL */
   |  `(` expression `)'
   ;

_optional_paren_formal_parameter_list
   :  /* NULL */
   |  `(` formal_parameter_list `)'
   ;

_optional_port_clause
   :  /* NULL */
   |  port_clause
   ;

_optional_port_clause_port_map_aspect
   :  /* NULL */
   |  port_clause port_map_aspect `;'
   ;

_optional_port_map_aspect
   :  /* NULL */
   |  port_map_aspect
   ;

_optional_port_map_aspect
   :  /* NULL */
   |  port_map_aspect
   ;

_optional_relational_operator_simple_expression
   :  /* NULL */
   |  relational_operator simple_expression
   ;

_optional_report
   :  /* NULL */
   |  REPORT expression
   ;
```

```
_optional_sensitivity_clause
    :  /* NULL */
    |  sensitivity_clause
    ;

_optional_sensitivity_list
    :  /* NULL */
    |  sensitivity_list
    ;

_optional_severity
    :  /* NULL */
    |  SEVERITY expression
    ;

_optional_sign
    :  /* NULL */
    |  sign
    ;

_optional_signal
    :  /* NULL */
    |  SIGNAL
    ;

_optional_signal_kind
    :  /* NULL */
    |  signal_kind
    ;

_optional_simple_name
    :  /* NULL */
    |  simple_name
    ;

_optional_timeout_clause
    :  /* NULL */
    |  timeout_clause
    ;

_optional_use_binding_indication
    :  /* NULL */
    |  USE binding_indication ';'
    ;

_optional_when_condition
    :  /* NULL */
    |  WHEN condition
    ;
```

```
_optional_variable
   :  /* NULL */
   |  VARIABLE
   ;

_use_clause_list
   :  /* NULL */
   |  _use_clause_list
   ;

_waveform_when_condition_else_list
   :  /* NULL */
   |  _waveform_when_condition_else_list waveform
      WHEN condition ELSE
```

## B.2 Unsupported Constructs

The constructs described in this section are defined in the full VHDL 1076 language standard but cannot be supported in the Synchronous VHDL Subset. Each of the constructs is shown along with a simple statement describing why the construct cannot be supported. Chapter 4 describes these reasons in more detail.

```
access_type_definition
   :  ACCESS subtype_indication
   ;

allocator
   :  NEW subtype_indication
   |  NEW qualified_expression
   ;
```

These two constructs cannot be supported because they are used provide dynamic memory allocation

```
disconnect_specification
   :  DISCONNECT guarded_signal_specification
      AFTER expression ';'
   ;
```

This construct cannot be supported because it manipulates the number of drivers attached to a signal. It represents a manipulation of the runtime data structures in the simulator.

```
_optional_transport
   :  /* NULL */
   |  TRANSPORT
   ;
```

The transport delay model is not supported in the subset

```
_optional_after_expression
   :  /* NULL */
```

```
    |  AFTER expression
    ;
```

The after clause is not supported because it is only relevant for the inertial and transport delay models, neither of which are supported in the Synchronous VHDL subset.

```
file_declaration
    :  FILE identifier ':' subtype_indication IS
          _optional_mode file_logical_name ';'
    ;

file_logical_name
    :  expression
    ;

file_type_definition
    :  FILE OF type_mark
    ;
```

The three constructs above cannot be supported because they provide an interface from the simulator to the operating system.

```
signal_assignment_statement
    :  target GEQ _optional_transport waveform ';'
    ;

selected_waveforms
    :  _waveform_when_choices_list
       _waveform_when_choices
    ;

_waveform_when_choices
    :  /* NULL */
    |  _waveform_when_choices _waveform_when_choices ','
    ;

_waveform_when_choices
    :  waveform WHEN choices
    ;

waveform
    :  waveform_element
    |  waveform ',' waveform_element
    ;

waveform_element
    :  expression _optional_after_expression
    |  NULL _optional_after_expression
    ;
```

```
_waveform_when_condition_else_list
    :  /* NULL */
    |  _waveform_when_condition_else_list waveform
       WHEN condition ELSE
```

The use of waveform signal assignments is not supported in the Synchronous VHDL subset because the waveform assignment implicitly uses the transport delay model. The transport delay model is not supported in the subset.

```
unconstrained_array_definition
    :  ARRAY `(` _index_subtype_definition_list `)'
       OF subtype_indication
    ;

_index_subtype_definition_list
    :  index_subtype_definition
    |  index_subtype_definition_list `,' index_subtype_definition
    ;
```

Unconstrained arrays are not supported in the Synchronous VHDL subset unless there is a way to determine the bounds which will actually be applied to the array. So, while not all unconstrained arrays are illegal, any use of an unconstrained array which cannot be assigned a fixed bound at compile time is not legal. This restriction is due the requirement for finite state.

```
    :  WAIT _optional_sensitivity_clause
       _optional_condition_clause
       _optional_timeout_clause `;'
    ;
```

The use of the timeout clause in the wait statement is not supported in the subset. The timeout clause refers to a time limit and therefore is not a reactive construct; the behavior of the wait statement is not defined in terms of a reaction to events external to the process.