On Interfacing Existing Hardware Description Languages to State-Space Exploration-Based Verification

Wendell Baker

June 1, 1993

This report extends the ideas presented in [Bak93] towards state-space exploration-based verification. In that work a subset of the VHDL [IEEE87] hardware description language, called Synchronous VHDL (S-VHDL) with reactive properties was identified. The focus in the previous work was the identification of subset for use in a simulator, thus demonstrating by example the efficacy of the subset. The work reported here is directed at building upon the properties of the S-VHDL subset to allow state-space exploration-based verification methods such as model-checking [CES86] or language-containment [Kur90] to be used on hardware systems described in terms of a hardware description language such as VHDL.

Ultimately this work is directed at the development of a verification-based debugger environment as shown in Figure 1. Shown there is a much larger project than is being reported here. Only the content of the parts shown in bold are described in this report.

1 A Verification-Based Debugger System

The big-picture idea is that there will be a state-exploration-based verification systems that will allow one to write programs in some sort of high-level language such as VHDL, Verilog [Ver90] or Esterel [CIS88] and then verify properties of that design using techniques such as model-checking or language-containment. The system will consist of a compiler for the high-level language, a verification algorithm and a user interface to allow for the visualization of the errors identified by the verification algorithm.

1.1 A Language-Neutral Intermediate Representation for Verification

The job of the compiler as shown in Figure 1 is to transform one of the commonly-used hardware description languages into an internal form that can be manipulated automatically. This report concentrates on an extension to S-VHDL called V-VHDL as an example of such a hardware description language. The internal form which is common to all high-level languages is the Non-Deterministic Abstract Machine (NDAM). This form provides a convenient intermediate level of description between the high-level language and the automata models that are used as inputs to the verification algorithms.

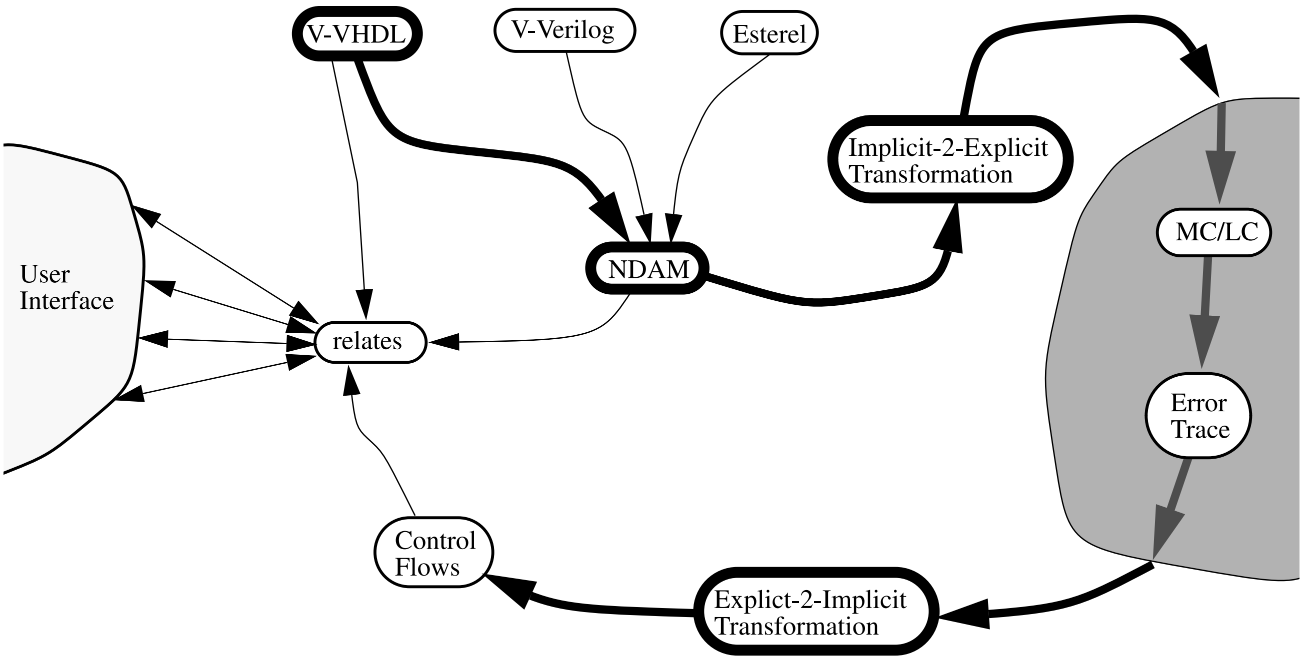


Figure 1 A Verification-Based Debugger System

The definition of an early version of NDAM was presented in [Bak93] but was described there only as an aid to the transformation of S-VHDL into a legal, compilable Esterel program. This report gives the formal definition of a synchronous semantics for networks of reactive NDAM processes directly in terms of event-derivatives. The notion of the derivative of a NDAM-network with respect to its environment ($\partial Network / \partial Env$) is defined in terms of the events E_i which can occur from the environment; the network-derivative is thus defined in terms of the derivative of processes ($\partial Proc / \partial E_i$) by a chain rule that takes into account causality [Halb93]. The total behavior of the network is thus the graph induced by the transitive closure of its derivatives. This is presented in Section 3 and is the substantive contribution of this report.

The design of the NDAM intermediate representation is important for it is at that level where the most powerful automatic abstraction techniques will be applied to the system description in order to simplify the verification problems relative to the properties to be proved. Automated abstraction of large state spaces is considered to be of fundamental importance for any practical verification system. Verification algorithms that have been published to date all suffer from one fundamental weakness which is that they are extremely unstable relative to the ordering of the input variables and the internal structure of the state-spaces. This is due to the use of Binary Decision Diagrams (BDDs) [Bry86] to make the verification algorithms efficient. BDDs while making feasible many previously-infeasible techniques also have the property of having highly non-intuitive performance degradation behavior. The control abstractions proposed here, called *control trees*, were first developed for optimizing compilers [ARZ91] as an aid to reducing the complexity involved in the dataflow analysis done there; they are described in this report in Section 3.4.

1.2 Explicit-State versus Implicit-State System Descriptions

The larger problem addressed by this work is the difference between hardware description languages such as VHDL, Verilog or Esterel and the existing body of automata-theoretic work on which state-exploration verification is based [CJM91] [Hal93] [Kur90] [Kur92] [McM92]. While the former are imperative languages where there is a distinction between program (control) and variables (data) latter are simple automata models where the description presents a graph of states and edges between the states. This fundamental difference between the two sorts of descriptions can is termed the *implicit-state versus explicit-state distinction* in this work.[1]

1.2.1 Explicit-State System Descriptions

This distinction can be best illustrated in the context of the S/R language which is accepted by the COSPAN verification system [HK87] [HK90]. In that system the language-containment method of verification is based on the exploration of a state-graph that represents the possible states and state transitions through which the system under study can evolve. As such, the front-end or user language,[2] of such systems have been oriented towards describing the state-graph in a fairly explicit way.

The S/R language is oriented at providing a convenient way for the programmer to describe the state transition graph of the system in an explicit manner. A simple example of a system described in this way is shown in Figure 2 with its accompanying S/R description. The important observation to be made about this description is that the S/R language describes the *explicit* state of the design in terms of specific *named* states and the transitions between those named states. The whole design is then the concurrent execution of all the individual state machines.

The fact that all of the states of the design are explicitly named, and the transitions between those states are explicitly indicated has important ramifications for the use of language-containment based verification methods. Not only is the specific structure of the state graph made explicit to the programmer, but it is also very obvious to the programmer which states are important and should be executed infinitely often. Properties to be verified are then phased in terms of cycles of states or edges in this graph. Thus both the design and the properties to be verified about the design relate directly to the tangible graph representation with which the user has direct access.

The tangibility property of the user language has the effect that it is relatively easy to identify important states in the design by virtue of the fact that all states are named. On the other hand, this simplicity has a cost in terms of the expression of the design properties: it is imperative that the user language provide a succinct way of generating large sets of state names for inclusion in the design property expressions. For example, in the case of the S/R language, it is imperative that there be easy ways of generating the names of pause states to allow them to be excluded from consideration in a *cyset* declaration. In fact in S/R, there is a fair bit of language mechanism dedicated to allowing the user to describe these large numbers of states in a succinct manner; the power of these mechanisms is tremendously understated in descriptions of that system. An example of this, taken from [KH87] is shown in Figure 3.

1.2.2 Implicit-State System Descriptions

Where existing verification systems [HK87] [HK90] [McM92] can be said to take an explicit-state approach to system description. On the other hand, imperative hardware description languages can be said to describe systems using implicit-state as the state of a process described in any of these languages consists of not only the values in the variables but also the latent value of the program counter.

1. A distinction should be drawn here between use of the terms *explicit-state* and *implicit-state* in this work and its use in the context of [CM90] wherein the next-state relation of a sequential circuit is implicitly represented as a function (usually represented in a BDD data structure) that is true if-and-only-if the inputs and the outputs, treated as inputs to the function, are part of the next-state relation.

2. There is a need here to distinguish the programming language by which the user interacts with the verification system and the language-theoretic set which is being verified by means of the containment check. In this work the terms *user language*, or equivalently, *design description language*, are be used for the former; references to the latter should be clear from the context.


```
proc PING
   import PONG
   selvar #: (0, 1)
   stvar $: (0, 1)
   init 1
   trans
   0     { 0 }
      ->1: (PONG: 1)
      ->$: else;
   1     { 0, 1 }
      ->0: (# = 1)*(PONG: 0)
      ->$: else;
end PING
```

```
proc PONG
   import PING
   selvar #: (0, 1)
   stvar $: (0, 1)
   init 1
   trans
   0     { 0 }
      ->1: (PING: 1)
      ->$: else;
   1     { 0, 1 }
      ->0: (# = 1)*(PING: 0)
      ->$: else;
end PONG
```

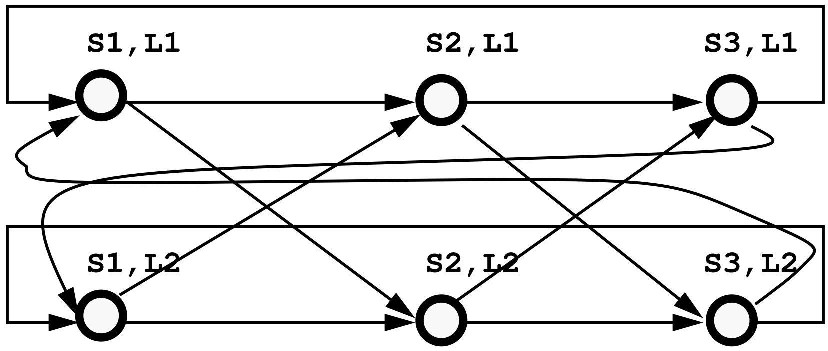
Figure 2 A Simple “Ping Pong” System

```
cyset { 1@, 2@, 3@, 4@, 5@, 6@, 7@ }

cyset { [i in 1..7] {i@} }

cyset { *@ }
```

Figure 3 Equivalent Expressions for a Certain Cycle Set

The syntactic facilities for design expression in the S/R language are important to understand because it is highly desirable that these same features be reproducible in another design description language. These facilities should be available or there should be obvious reasons why they are not needed. In the case of this work, which uses S-VHDL as a basis, the imperative semantics provides a much different set of facilities for the expression of the design and so there is a different set of expressivity requirements on properties to be verified.

In contrast with the explicit state graph representation of the S/R language, the imperative semantics of the VHDL language implies that there is an *implicit* component of the state of a computation. The state of a VHDL process is not directly represented by a node in a state graph, but is implicitly represented as a pair consisting of a position in the code body of the process and a binding of the variables. As can be seen in the trivial example of Figure 4, there are six possible states in the process P1. Each state corresponds to a pairing of a *wait* statement and a binding of the single process-local variable. In this case there are two possible wait statements: `L1` and `L2`, at each of there are three possibilities for a binding of the variable `S`. Thus there are six states to the process and they are described by the following six predicates:




```
P1:
process
   type state is (S1, S2, S3);
   variable S: state := S1s;
begin
         loop
            S := state'succ(S);
            if ping then exit end if;
   L1:      wait on all;
         end loop;
         loop
            S := state'succ(S);
            if pong then exit end if;
   L2:      wait on all;
         end loop;
end process P1;
```

Figure 4 A Simple Six-State Process

$$(pc \Leftrightarrow L1) \wedge (S \Leftrightarrow S1) \qquad (pc \Leftrightarrow L2) \wedge (S \Leftrightarrow S1)$$

$$(pc \Leftrightarrow L1) \wedge (S \Leftrightarrow S2) \qquad (pc \Leftrightarrow L2) \wedge (S \Leftrightarrow S2)$$

$$(pc \Leftrightarrow L1) \wedge (S \Leftrightarrow S3) \qquad (pc \Leftrightarrow L2) \wedge (S \Leftrightarrow S3)$$

Thus, in order to interpret a VHDL process in a manner that allows for the verification problem to be posed, a state transition graph must be derived from the imperative control-flow of the process. This transformation is shown in Figure 1 performing the conversion from implicit-state language-neutral NDAM representation to the explicit state-graph form for use in the verifier. The inverse transformation is shown converting the explicit-state error trace back into control-flows in the implicit-state process form of the NDAM. The *relates* bubble is shown taking these control flows and identifying the relevant structures in the original V-VHDL, associating them with registers, signals and instructions in the intermediate NDAM representation and presenting this to the user interface component of the system.

This report describes a method by which any language that can be meaningfully compiled into a network of communicating NDAM processes can be converted into the explicit state-graph form that is required for state-space exploration based verification. For the purposes of this work, a VHDL-like[1] language, V-VHDL, is used as an example of such a language though one could fruitfully envision using a language such as Esterel or an extension of Verilog being used in the same manner.

1.3 The Visualization Component

Current thinking on verification-based debuggers indicates that the user interface component is extremely important. Despite this perceived importance very little research has been conducted to determine what the visualization component of the system should contain. Some of the options which have been proposed to date include:

- Simple animations of the system being verified using an error-trace that contains single values. The error-trace consists of a stream of (explicit) states through which the system evolved on its path to oblivion. One could envision a window-based display showing each process and being able to watch as the system stepped forward (or backwards) in time.
- More complex displays of the system would use an error-trace that consisted not of a stream of single values but rather would consist of a stream of sets of values. The display mechanism would then somehow render a graphical map of the all the invalid behaviors of the system. Such a map could involve scribbling lines over the system description to show all of the possible the paths that the system could evolve through as things went wrong.

The quantity of traces displayed by the visualization component is one aspect of the information that can be displayed in an error trace. Another aspect is the treatment of time in the visualization of the error trace. It may be useful to treat time in the error trace the way the programmer lives it - sequentially through a film-like sequence of snapshots of the system. On the other hand it may be more useful to

- Using the scribbling-approach wherein the control flows through which the system evolves is drawn on top of the code itself as one would with a colored pencil one can show the evolution of time through changes in color from red through violet. Bright red for example could represent the initialization of the system. The control-flow line would then progressively tend towards the violet end of the spectrum as the line represented movement towards a bad state (for safety properties) or towards infinity (for eventuality properties). An approximation to this sort display is shown in Figure 5 for the simple six-state process of Figure 4.
- Alternatively, it may be feasible and convenient to represent the explicit state-graph of the process directly. This could require some amount of placement and routing skill in order to render a graph which is readable and which provides some sort of intuition onto the problem. An approximation to this sort of display is shown in again for the six-state process. Unfortunately this approximation does not come off so well in the monochromatic format of this report; the thinking however is that with a spectrum of color to maintain the continuity across the overlapping scribbles, this might be a very useful form of visualization.
- Finally, one of the more graphically aggressive ways of displaying the control aspect of an error trace over time involves using three-dimensional graphics. Consider the standard display of a single process as is given in Figure 4 for example. It is fairly straightforward to show the currently-active statement by highlighting it in a different color. Now, consider replicating the snapshot of the process back into the page as if on a deck of cards. The idea is to allow the user to browse the error trace by slicing the behavior along instants of reactive time and providing a simple interface by which an instant can be selected and browsed, variable values examined and the like. Moving forward or backwards in time along the stack-of-cards is thus intuitively obvious. An approximation to this sort of graphical display is shown in Figure 7.

1. As described in the following sections, some extensions to the basic Synchronous VHDL framework is necessary in order to represent nondeterministic execution as is supported in other system description languages (*e.g.* S/R or SMV [McM92]); this extended language is referred to as V-VHDL.

```
P1:
process
   type state is (S1, S2, S3);
   variable S: state := S1s;
begin
         loop
            S := state'succ(S);
            if ping then exit end if;
   L1:      wait on all;
         end loop;
         loop
            S := state'succ(S);
            if pong then exit end if;
   L2:      wait on all;
         end loop;
end process P1;
```

Figure 5 An Error Trace in the Simple Six-State Process

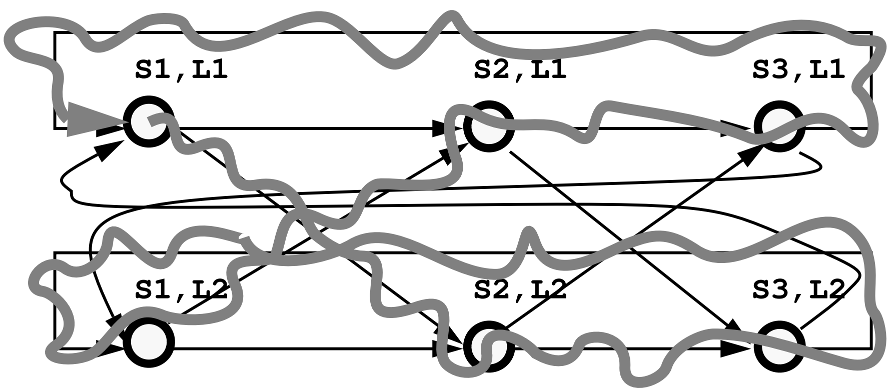


Figure 6 Another Error Trace in the Simple Six-State Process

Interestingly the form of the error trace and the amount of information that can be displayed there is not simply a function of the imagination of the visualization component developer. There is a strong interaction between the verification algorithms with respect to the granularity of the error trace and the speed with which an error trace containing a bad cycle can be generated. Previous experiments [HTKB92] have shown that the speed of the language-containment algorithm can be vastly improved over that presented in [HK90] if a only a simple single-state trace is desired. The original language-containment algorithm used a transitive-closure-based computation to produce cycles of strongly-connected components. It is thus is able to produce an error trace consisting of sets of states, any one trace of which is an invalid trace. The improved language-containment algorithm uses the so-called Emerson-Lei technique and avoids the transitive-closure computation. The result however is a single-state error trace.[1] The current thinking is that there are possibly other interactions which involve algorithm features such as early bad-cycle detection [HTKB92] and the debugger which have yet to be discov-

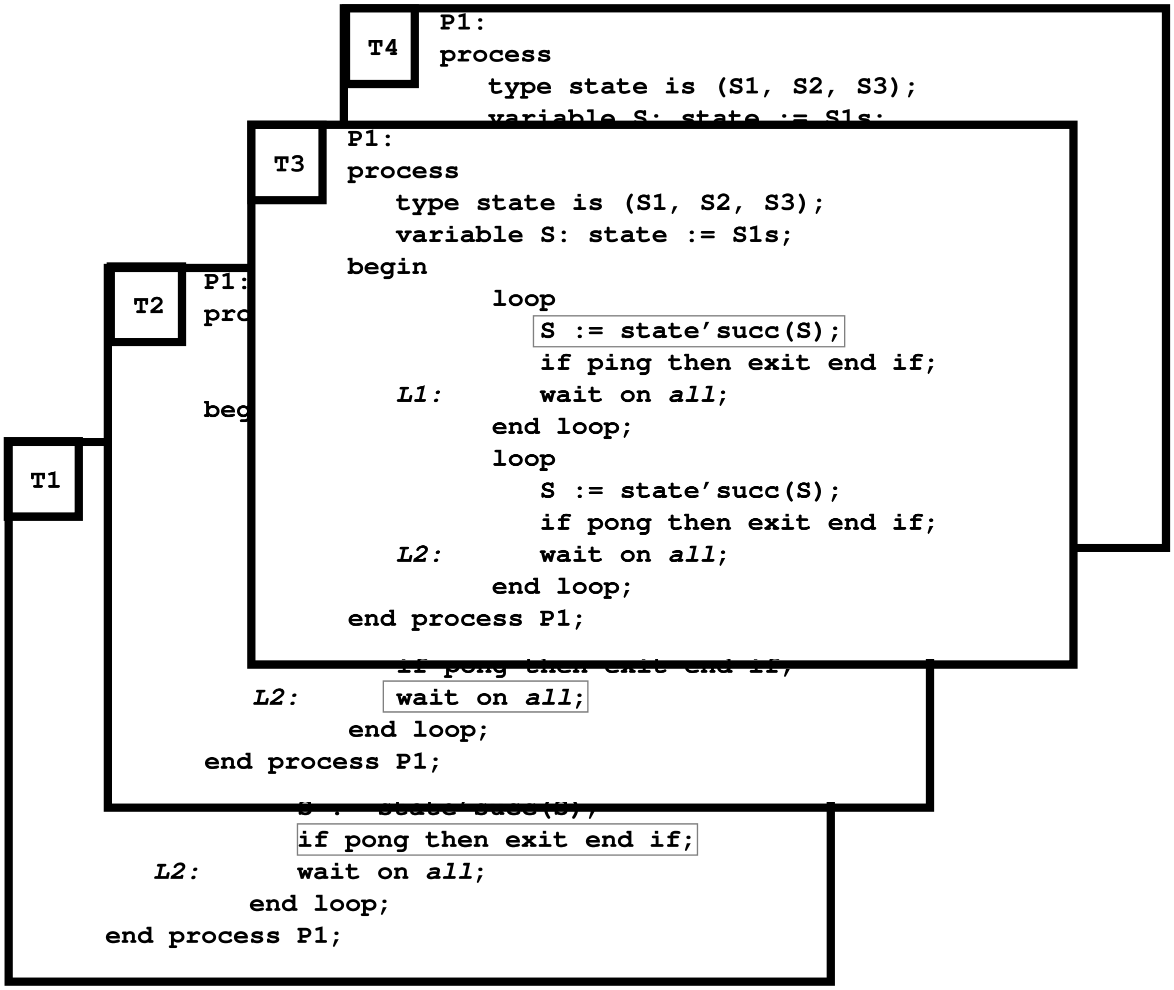


Figure 7 A Deck-of-Cards Trace in the Simple Six-State Process

ered. More work needs to be done in the area of the interactions between the needs of the visualization component and the verification algorithms.

1.4 Summary

The work presented here is oriented at addressing the larger question of the design of a verification-based debugger environment. There are many levels to the system as it is described here. Those levels and the representations and transformations required to make it work are shown schematically in Figure 1. This report specifically treats the highlighted path in that diagram. That particular flow traces the following path:

- A design is described in the V-VHDL extension to the Synchronous VHDL subset.

1. Unfortunately I don't quite see how this conclusion must be drawn from the material that is presented in [HTKB92]. However one of the authors of that work assures me that this is the conclusion that should be drawn.

- A compiler (not shown in the figure) transforms each V-VHDL process, including those that describe design properties to be verified, into an NDAM process. The whole design is composed as a network of the NDAM processes.
- A hardware/software synthesis step could then transform the NDAM process network into an implementation in either hardware or software some mixture of the two as desired. This step is not shown in the figure but it would be the synthesis component of a synthesis and verification environment.
- Focusing on design verification however, the NDAM network is converted from its imperative implicit-state form into an explicit-state graph for use in a verification algorithm such as model-checking or language-containment. Any of a number of choices are available for the actual verification algorithm so long as an explicit-state graph describing the verification problem instance is accepted and an error-trace is produced.
- The verification algorithm produces an explicit-state error-trace which must then be inverted back into a form suitable for display. This step inverts the *implicit-2-explicit* state transformation that was performed earlier. This step extracts the control-flows that caused the bad cycle and allows for the relationship between the control flows in the instruction-level NDAM and the statement-level high-level language to be established.
- The control-flows driving to an error condition can then be viewed using any of a number of graphical and temporal visualization techniques. A few were sketched earlier in this section.

The following sections provide more detail on various aspects of the design of this system. The essential features of the Synchronous VHDL subset is reviewed in Section 2. That section is mostly review as the full definition of S-VHDL can be found in [Bak93]. The following section, Section 3 contains a description of the Non-Deterministic Abstract Machine representation of synchronous computations. In that section, the basic NDAM-1 language as described in [Bak93] is reviewed. Beyond that review however, the new extensions (NDAM-2) which were prompted by the lessons learned in the design and evaluation of Synchronous VHDL is given in Section 3.2.[1] This is followed by a novel development of the event-derivative semantics for NDAM in Section 3.3. Based on the event-derivative semantics a scheme that allows for the automatic recovery and representation of control abstractions using control trees and non-determinism is given in Section 3.4. Having defined the underlying NDAM-2 language, its formal event-derivative semantics and a method for abstracting complex control descriptions of NDAM networks, the question of the capabilities of this representation which can be reproduced in the source-level language is addressed. To that end an extension to Synchronous VHDL called V-VHDL is sketched in Section 4 which extends S-VHDL with non-determinism and design property declarations.

2 S-VHDL - Synchronous VHDL[2]

The only truly sound way to define the state-transition aspect of VHDL is to do it starting from the semantics that can be supported, finite-state state-transition semantics, and move forward to the identify what aspects of the syntax can be supported under these restrictions. The meaning of a VHDL-1076 program is defined in terms of the operations on an abstract simulator in the style of Plotkin [Plo81] so we have a situation where the restrictions on the abstract simulator will imply restrictions on the constructs in the language. The idea then in the definition of the largest subset of VHDL-1076 that has synchronous semantics is to start with a restriction on the abstract simulator model and work forwards to identify the restrictions that are implied on the language. The fundamental restriction on the abstract simulator model is straightforward: it must be finite state. There are four basic restrictions on the simulator that enforce finite-state behavior - all other restrictions 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.

1. The lessons-learned are described in Section 5.4 of [Bak93]. They are reviewed in Section 3.3 here for completeness.
2. This section is adapted in its entirety from Chapter 4 of [Bak93]. It is presented here only to maintain the flow of the report.

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 the VHDL subset. In this subset, processes are constrained to act as finite state machines and thus the whole VHDL program acts as a network of communicating finite state machines.

The execution semantics of a VHDL program under the restrictions outlined above are the same as those of any other programming language operating under the restriction of synchrony. 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 exist 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 was a connection that was used to advantage in earlier presentations of S-VHDL. In this report however, the understanding is that structures in the S-VHDL language can be converted to the register-transfer-level formalism of the NDAM using standard optimizing-compiler code-generation techniques [ASU86] [ARZ91]. The formal event-derivative semantics of NDAM is presented in Section 3.3 and thus it is sufficient at this point to focus only on the restrictions imposed by S-VHDL. This can be done safely given the knowledge that any compilation of a Synchronous VHDL program into the NDAM form will preserve its behavior. This is a very mild assumption on the implementation of any S-VHDL compiler.

3 NDAM - Non-Deterministic Abstract Machine

The Non-Deterministic Abstract Machine (NDAM) was originally introduced in [Bak93] as notational aid used in the definition of a translation of S-VHDL to Esterel. Such a circuitous path was convenient at that point for the intent was to demonstrate the existence of synchronous subset of the VHDL-1076 language. Rather than attempt to define the meaning of S-VHDL in terms of event-derivatives on S-VHDL programs directly, it was more efficient and demonstrative to exhibit a translation path that rendered the S-VHDL subset in a synchronous language which has a known and well-published semantics: Esterel.[1]

The presentation of the original NDAM code (NDAM-1) is given here again in order to provide a formal definition of the event-derivative semantics of NDAM. This more in-depth presentation is important as it gives a what is believed to be a novel formulation of the synchronous event-derivative semantics for a computation described on an instruction-set processor. The event-derivative formulation is given for both the case where all processes are always active as well as the case where some processes can become idle (subprocesses). This latter case is significant as an NDAM network consists of a set interacting processes, some of which are subprocesses that are idle until invoked by their parents.

1. The semantics of Esterel has evolved over its life and so has been well published: [BC84] [BCG86] [BDS91] [Ber91] [CIS88] and [Hal93] were the references used in this work.

3.1 NDAM-1 - The Basic NDAM[1]

The Non-Deterministic Abstract Machine (NDAM) provides a representation for computing systems that is somewhere in complexity between high-level languages such as VHDL and lower levels of representation such BLIF-MV.[2] The major design goal of this assembly code-level format is that it be a 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 possible [ASU86]. The idea is that the transformation from the high-level language be straightforward enough that there can be no fundamental disagreement on how it should be done. This is important for reasons of clarity but also because it in some sense defines the meaning of the high-level language within the synchronous/reactive model.
2. All information needed for further compilation be represented directly in the description. 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 assembly 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 NDAM assembly language as an intermediate step in the compilation of Verilog, Esterel or any other imperative high-level language with the appropriate synchronous semantics.
4. It is designed so that compiler optimization techniques can be made will work using it [ARZ91]. These would include constant propagation, common subexpression elimination, dead code elimination and the like (driven by dataflow analysis and other techniques). The use of control trees as a representation for control abstraction is presented along these lines in Section 3.4.
5. The conversion to state machines from this format is possible. Defining the semantics of NDAM directly in terms of event-derivatives is the main contribution of this report.

3.1.1 A NDAM Process

A NDAM-1 description (a file) represented a single sequential process in exactly the same sense as a single VHDL process. Under that definition, the process interacted with other processes by the emission and reception of events on signals. The specification of the network of processes and the linkage of their signals was not defined within the framework of a single NDAM file; such was expected to occur externally. In [Bak93] the existence of a "linkage level" tool that would will link multiple NDAM files was postulated but not explicitly defined.[3] Extensions defined for NDAM-2 make up for this deficiency by explicitly allowing for multiple processes and a top-level network declaration; these extensions are described in Section 3.3.

3.1.2 The Execution Model

The execution semantics behind the NDAM architecture is synchronous execution under the reactive model. This model of computation enforces a discipline of synchronous parallel execution where each process in the system only reacts to events sent to it by its environment. There is no autonomous execution and thus there is no dynamic synchronization implicit in the model. All synchronization activities are explicit in the action / reaction of the parallel processes [BC84].

The reactive model requires the following properties to be true:

---

1. A distinction in this report is made between the two languages NDAM-1 and NDAM-2. This naming convention is designed solely to illustrate that part of NDAM which was defined in [Bak93] and is simply being reviewed here and that part which is being presented here for the first time. NDAM-1 is the name given to the former and NDAM-2 is the name given to the latter.

2. Brayton, et al. *BLIF-MV: An Interchange Format for Design Verification and Synthesis*, November 1 1991. Available from the authors.

3. In practice the Esterel compiler was used as a linker. This occurred because in [Bak93] the NDAM code was converted *back* to Esterel in order to use the Esterel compiler to extract the automata and produce executable object files.

1. Time is measured in specific *instants* which are measured relative to *events* from the environment between which nothing of interest occurs. Time may therefore be measured in units of seconds, meters or button-presses. Thus, the computations of a process' reaction to events from its environment are said occur *instantaneously* with respect to the next event. A process is required to complete its reaction before the next event occurs. The condition required for this to be believable in the real world is called the *synchrony hypothesis*.[1]
2. There is a distinction between the process and its environment. In the NDAM, this takes the explicit form of a distinction between signals through which communication with the environment is performed and registers which contain state not visible to the environment or to other processes.
3. Each process only reacts to events received on its signals. A reaction may cause the modification of the process' local variables and the emission of other signal events. The key point though is that processes do not evolve with respect to time of their own accord; only events from the environment may cause them to progress.

One can note that these restrictions are the same sorts as are imposed on clocked digital systems. In that case, the system clock can be thought of as the event which causes each of the processes to compute their reaction. Here however, the conditions of synchrony, causality and reactivity are phrased in general terms to allow for the possibility of synchronous reactive software systems.

Each NDAM-1 assembly file represents a single such reactive process. Thus a complete system is composed of many processes that are explicitly communicating with each other. The reactive model implies that each process can be transformed into an equivalent finite state machine. Thus the system as a whole is a set of communicating finite state machines.

3.1.3 The Abstract Machine Model

The model on which the NDAM assembly code is built is an abstract register-transfer architecture. 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.

A NDAM assembly code file describes one process which is assumed to execute in a synchronous, reactive manner relative to its environment. The process contains 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 the derivation of the system behavior subject to the event patterns on its signals forms the basis of both reactive compilation and of language-containment verification.[2] Though termed "registers" and "signals" for the purposes of the NDAM, objects with the same properties can be found under other names in a number of languages, including Verilog, VHDL, and Esterel.

3.1.4 The Major Artifacts

The following are the major artifacts in an NDAM-1 description:

types — The NDAM architecture consists of typed registers and typed signals. A type declaration introduces a name denoting a set of values. Once defined, the type name is attached to one or more register or signal declarations to describe the set of values that the register or signal can take on.

A type defines a set of values that is denoted by two integer bounds: *low* and *high*. For example the set of integers between -1 and 10. The range of values which can be given for the low and high bounds is arbitrary; types may be declared assumed to be of arbitrary size. This is to say that there is no restriction on the

1. This term was first used in [BC84].

2. Such a distinction is not a requirement for model-checking. Under that regime design properties are phrased in terms of the trajectories of predicates on states of the system. However, maintaining a system description in terms of independent but interacting processes seems to be a good engineering decision with respect to model-checking (*c.f.* the SMV `module` construct [McM92]).

bounds to, say 32-bit integers; the bounds may be of arbitrary magnitude. No implementation of a NDAM assembler is allowed to enforce limits on the magnitude of type range boundaries.[1]

Another important point to make is that while types are declared with integer literals, this does not imply that a type declaration must be interpreted as the declaration of integer subrange. It is fully expected that NDAM data types may be reencoded by synthesis or verification tools in the course of their operation. Though the type declaration describes a bounded integer range, this can best be thought of as the declaration a multi-valued variable. A reencoding of the multi-valued variable is valid so long as all of the operations on that multi-valued variable are respected. The most convenient encoding for certain operations (*e.g.* addition or multiplication) may in fact be the 2s-complement integeer representation.

registers Registers hold values which are local to a process. A register can be either a scalar, an array or a record. It is most useful to consider that a scalar is a degenerate array - an array with only one member; or a degenerate record - a record with only one field. A register has a type and operations are only allowed between registers of the same type. There are register transfer operations to insert and extract values out of register and array registers.

As mentioned, registers may be of storage class plain, array or record. 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.

signals Signals are used to broadcast values between processes. Thus a process can be either sensitive to a signal or it can drive a signal or both. A process is said to be sensitive to a signal if the signal is referenced in a wait instruction. A process is said to drive a signal if the signal is assigned to in a signal emission instruction. The same assignment compatibility rules that exist between registers also exist between registers and signals.

As with registers, a signal may have a storage class of either plain, array or record. And as with registers, the array and record are only linear; it is expected that the high-level language compiler which produced the assembly code will have linearized and flattened the multi-dimensional arrays and records respectively.

3.1.5 Names

Names are important for at the assembly-level of representation, association is done strictly by name and there is very little scoping of names. The visibility of names of various classes in NDAM is patterned after the name visibilty in modern compilers, linkers and libraries.[2] In those systems the various high-level language compilers manage complex and nested name scopes at a levels appropriate to their respective high-level languages. The compilers transform their names down to a form where they can be represented at the inter-file linkage level. In such systems, there is an assembler and linker which is common to all the high-level language compilers. The assembler and linker provide support for two scopes: *internal* linkage and *external* linkage relative to the unit of assembly or linking which is the file.

1. This lack of limitation at the NDAM assembler level does not guarantee that the verification algorithms envisioned for the debugger system of Figure 1 will actually be able to handle the huge state-spaces implied by large type domains. Synthesis (of software, hardware or mixed hardware/software systems) on the other hand does benefit from the lack of such restrictions as synthesis typically focuses on the control-plane aspects and leaves the data-plane manipulation to more special-purpose structures than state graphs. The contrast here is with representations where the use of large-domain multi-valued variables is problematic at best (*c.f.* BLIF-MV, Footnote 2. on page 11).

2. For example, see the discussion of such in Bjarne Stoustroup's *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991 or the ANSI C++ draft standard.

In the case of NDAM the names are patterned in the same way: there are two levels of scoping *internal* and *external* which are relative to a *process.* Any more intricate name relationships than this simple two-level scheme must be managed by the high-level language compilers which produce NDAM. At the NDAM assembly-level for example, signal names have external linkage as signals in different processes are associated by name directly. Two signals are connected in different processes if and only if they have the same name. Registers, types and instruction labels on the other hand have internal linkage and so are private to their respective processes.

There are a variety of name spaces in NDAM and each is distinct from the others. Each name space is kept distinct by virtue of a tag of the form **?** as in **?(*NAME*)**. Within the ***NAME***, there may be any sort of printable character except for parenthesis. This includes of course alphabetic characters, numeric characters and underscores in any combination as is the usual case for names. The tag **?** and the parenthesis are used to tag the various name classes; thus a type can have the same ***NAME*** as a register or a signal or a label. The four name spaces of NDAM-1 are:

T(*NAME*) Type names are denoted by the tag **T** (or equivalently **t**). An example of a type name would be **T(int)** or **T(std.character)**.

R(*NAME*) Register names are denoted by the tag **R** (or equivalently **r**). An example of a register name would be **R(0)** or **R(controller.synchronous.P1.reset)**. Constant register names are denoted the same way that register names are denoted for constant registers are but registers which cannot appear as the target of an assignment context.

S(*NAME*) Signal names are denoted by the tag **S** (or equivalently **s**). An example of a signal name would be **S(reset)** or **S(alu.carryout.bit9)**.

L(*NAME*) Instruction names, or labels, are denoted by the tag **L** (or equivalently **l**). An example of a label would be **L(start)**, **L(1)**, **L(2)** or the like.

The name spaces are all explicitly tagged in order to allow as wide a latitude as possible in the ***NAME*** part of the name. Additions to the namespaces for NDAM-2 have included exception-names of the form **E(*NAME*),** sensor names of the form **X(*NAME*)**, process names of the form **P(*NAME*)** and network names of the form **N(*NAME*)**.

3.1.6 Assembly-Code Syntax

The full set of instructions is given in Appendix A, however a summary of the instructions and their informal meanings is given in Figure 8.

In lieu of an exhaustive description of the major features of the NDAM instructions which is presented in Appendix A in any case, an example of the well-known the traffic light controller is shown in Figure 9. This example was originally found in [Kur92]. The control-flow graph for that example is shown in Figure 10; some obvious notational reductions have been performed on the instructions of Figure 9 have been used to make that graph presentable in such a small space. Some further elaborations of the traffic light controller example both in terms of the original S/R and the proposed V-VHDL language are given in Appendix C.

The important point of these instructions as shown in example of Figure 9 and in the flow graph of Figure 10 is that they are patterned after the 3-address code form proposed for use in compiler development [ASU86] [ARZ91]. As such, a wide variety of optimizations such as constant-folding, constant-propagation, common sub-expression elimination, strength reduction and the like may be performed at this level of representation directly.

In addition though the utility of this representation at this level of granularity is that it allows for a very direct bottom-up development of the event-derivative of a process ($\partial P / \partial E_i$) in terms of the event-derivative of a series of instructions

Class	Operation	Description
Declarations	**register R type, type, type** ... **= value, value, value** ...	
		Declare register **R** to hold the **types** and contain the **values**; scalars are declared with a single **type = value** while records have multiple such
	register R type dim, dim, dim ... **= value, value, value** ...	
		Declare register **R** to hold an array of **type** with the indicated dimensions and containing the indicated **values**
	constant R type, type, type ... **= value, value, value** ...	
	constant R type dim, dim, dim ... **= value, value, value** ...	
		Declare register **R** as before, except as a constant, **R** cannot be assigned to
Assignments	**L := R**	Assignment of register **R** to register **L**
	L := R.f	Assignment of field **f** in register **R** to register **L**
	L.f := R	Assignment of register **R** to field **f** in register **L**
	L := R[E]	Assignment of array member at **E** of array in **R** to register **L**
	L[E] := R	Assignment of register **R** to array member **E** of array in **L**
	L := R[E1, E2]	Assignment of array members from **E1** to **E2** of array in **R** to register **L**
	L[E1, E2] := R	Assignment of register **R** to array members from **E1** to **E2** of array in **L**
Operations	**L := R1** *op* **R2**	Primitive binary operations (not overloaded operations) **and**, **or**, **nand**, **nor**, **xor**, **=**, **/=**, **<**, **>**, **=<**, **>=**, **+**, **−**, *****, **/**, **mod**, **rem**
	L := *op* **R**	Unary operations (not overloaded operations) **not**, **−**, **dec**, **inc**, **abs**
Control	**goto B1, B2** ...	Unconditionally go to one of the blocks **Bi** nondeterministically. The base case of **goto B** is a deterministic control transfer
	if R goto B	If **R** has the value true then go to block **B**
	if not R goto B	If **R** has the value false then go to block **B**
	case R when *set1* **goto B1; when** *set2* **goto B2, ..., when** *setN* **goto BN**	
		Determine which *set* the value of **R** belongs in and go to that block **Bi**
Signals	**emit S R1, R2** ...	Emit the value of one **Ri** onto signal **S** nondeterminstically. The base case of **emit S R** is a deterministic signal assignment
	R := selection S	Assign the nondeterministic selection on **S** to register **R**
	R := presence S	Assign the presence of **S** to register **R**
	wait on all	Wait for an event on any signal referenced in a selection
Constraints	**range V, L, H**	Range check register **V** relative to registers **L** and **H** as $L <= V <= H$
	range V.f, L, H	Range check field **f** in register **V** relative to registers **L** and **H**
	range V[E], L, H	Range check element **E** in register **V** relative to registers **L** and **H**
Misc	**report M, S**	Report an error with register **M** as the message, register **S** as the severity
	exit	Terminate execution of the process

Figure 8 Summary Presentation of the NDAM-1 Instructions

```
type t(who_can_go) 2
constant r(A_can_go) t(who_can_go) := 0
constant r(B_can_go) t(who_can_go) := 1
register r(w) t(who_can_go)

L(start):
       goto L(1), L(2)
L(1):
       r(w) := r(A_can_go)
       goto L(3)
L(2):
       r(w) := r(B_can_go)
       goto L(3)
L(3):
       null
L(4):
       case r(w)
       when R(A_can_go) goto L(5);
       when R(B_can_go) goto L(6)

type t(controller_state) 3
constant r(pause) t(controller_state) := 0
constant r(go_A) t(controller_state) := 1
constant r(go_B) t(controller_state) := 2

signal s(sel_var) t(controller_state)

L(5):
       emit s(sel_var) r(pause), r(go_A)
       goto L(8), L(9)

type t(boolean) 2
register r(tmp1) t(controller_state)
register r(tmp2) t(boolean)
L(8):
       r(tmp1) := selection s(sel_var)
       r(tmp2) := r(tmp1) = r(go_A)
       if not r(tmp1) goto L(10)
       r(w) := r(B_can_go)
L(10): goto L(7)

register r(tmp3) t(controller_state)
register r(tmp4) t(boolean)

L(9):  r(tmp3) := selection s(sel_var)
       r(tmp4) := r(tmp3) = r(go_A)
       if r(tmp4) goto L(11)

register r(tmp5) t(controller_state)
register r(tmp6) t(boolean)

       r(tmp5) := selection s(sel_var)
       r(tmp6) := r(tmp6) = r(pause)
       if r(tmp6) goto L(11)
       goto L(12)
L(11): r(w) := r(A_can_go)
L(12): goto L(7)
L(6):  emit s(sel_var) r(pause), r(go_B)

register r(tmp7) t(controller_state)
register r(tmp8) t(boolean)

       r(tmp7) := selection s(sel_var)
       r(tmp8) := r(tmp7) = r(go_B)
       if not r(tmp8) goto L(13)

type t(avenue_state) 3
constant r(no_cars) t(avenue_state) := 0
constant r(cars_waiting) t(avenue_state) := 1
constant r(cars_going) t(avenue_state) := 2

register r(tmp9) t(avenue_state)
register r(tmp10) t(boolean)

       r(tmp9) := selection r(avenue_A)
       r(tmp10) := r(tmp9) = r(cars_waiting)
       if not r(tmp10) goto L(13)
       r(w) := r(A_can_go)
       goto L(14)
L(13): r(w) := r(B_can_go)
L(14): null
L(7):  wait on all
       goto L(4)
```

Figure 9 The Traffic Light Controller

($\partial P / \partial E_i \partial (I_1 I_2 \ldots I_N)$). In total, $\partial P / \partial E_i$ defines the computations that P may perform in transitioning from one *wait* instruction to another. The set of possible event-derivatives of a process is thus the set of possible instruction paths that the process may use to get from one *wait* statement to another. The event-derivative set thus defines the transition relation between all-pairs of *wait* statements. The control-flow graph form graphically represents the possible control-flow paths directly. This point is elaborated further in Section 3.3 in the development of $\partial P / \partial E_i$.

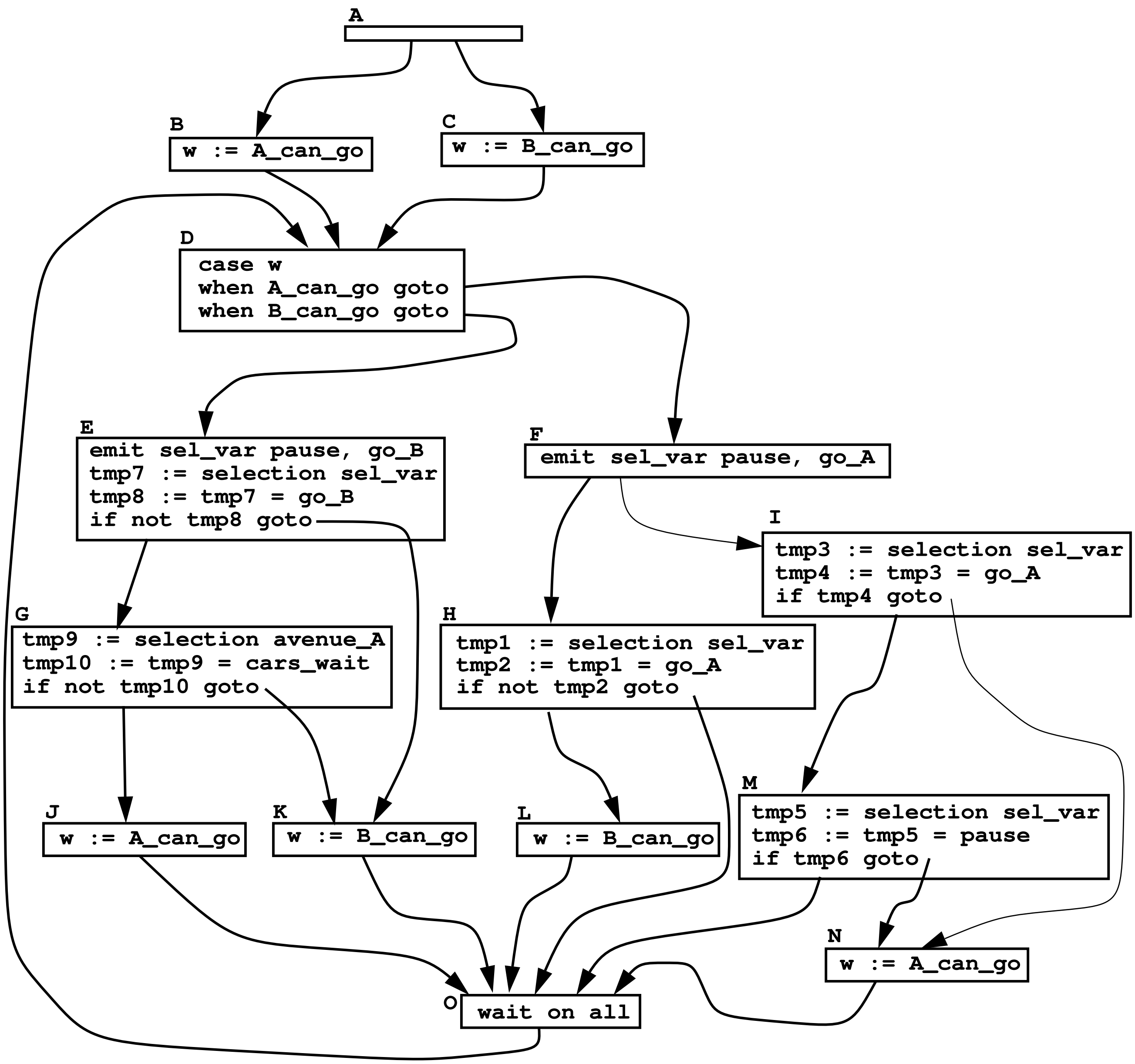


Figure 10 The Control-Flow Graph of Figure 9

3.2 NDAM-2 - Extensions For Software and Verification

The extensions to the NDAM-1 language used are presented in this section. In order to motivate those extensions however, Section 3.2.1 is devoted to a description of the problems that exist in NDAM-1 and how they appear in the use of NDAM-1 as the intermediate-form of a S-VHDL compler. The extensions, the NDAM-2 language, are then given in Section 3.2.2.

3.2.1 Lessons Learned from NDAM-1[1]

In constructing the S-VHDL simulator, a number of examples were coded in Synchronous VHDL. That experience led to some observations on the weakness of the VHDL computing model with respect to specification. As NDAM was originally designed to represent a subset of the VHDL language, this implied a weakness in the original NDAM-1 language too. These weaknesses are described in terms of VHDL and S-VHDL in particular, however the effects described are related to the underlying NDAM-1 language as NDAM-1 is the intermediate representation of an S-VHDL program. The problems in S-VHDL thus show up as problems in NDAM-1 and its use as a language-neutral representation for synchronous programming languages. The following sections outline the three weaknesses that were empirically uncovered in [Bak93]. The additions in NDAM-2 used to address these weaknesses is presented in summary in Section 3.2.2 and in detail in Appendix A.

3.2.1.1 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. An example of this phenomenon is shown in Figure 11. There, a change in the value of the signal **input** causes an event on the signal **output** depending on the value of the signal **input** in relation to the present value of the signal **output**.

```
signal input: bit := '0';

INVERTER:
process(input)
begin
   output <= not input;
end
```

Figure 11 May or May Not Cause an Event on **output**

The condition necessary for the definition of the derivative $\partial P/\partial E_i$ is a static condition - a condition which is true independent of the value that is associated with the signal at the point that the point that $\partial/\partial E_i$ is taken.

Fortunately for the case of VHDL-1076, there is the notion of a VHDL *transaction* which is formally exported to the source-language level though a signal-attribute of the same name. The execution of any signal assignment, independent of the value which is actually assigned causes the transaction-attribute of the signal to toggle. One can make a process be sensitive to signal transactions explicitly by referencing this signal attribute in a sensitivity-list situation. A modification of the example of Figure 11 that uses transaction-sensitivity instead of event-sensitivity is shown in Figure 12.

Signal transactions are typically not used as process triggers in the context of a discrete-event simulation environment as it is less efficient to focus on transactions than on events. This efficiency loss can be seen from a the definition of a VHDL-event as a *change in value on a signal*. A change in a signal-value will always require a recomputation to update the simulator state. A transaction on the other hand, which need not represent any change in value at all, will not always require a recomputation to reestablish consistency in the simulator's state.

The synchronous semantics of event-derivatives requires that NDAM be defined in terms of signal transactions. This is because $\partial/\partial E_i$ must be defined in terms of static conditions, not dynamic ones. Thus the NDAM instruction that provides

1. This section is adapted from Section 5.4 of [Bak93]. It is presented here to motivate need for the extensions to NDAM that are presented in the next section.

```
signal input: bit := '0';

INVERTER:
process
begin
   output <= not input;
   wait on input'transaction;
end
```

Figure 12 The INVERTER Defined With Transaction Sensitivity

for signal sensitivity, the *wait* instruction, is relative to reactive events which corresponds to a VHDL transaction. This distinction between a *reactive event* which corresponds to a *VHDL transaction* and a *VHDL event* which corresponds to no mathematical notion but is rather an artifact of the discrete event simulation model must be kept firmly in mind. NDAM events are reactive events as is required to define $\partial P/\partial E_i$

The lesson to be learned here with respect to language design is that imperative languages that are designed for use as specifications must ensure that the behaviors that they describe are statically analyzable with respect to synthesis and verification as well as being computationally efficient for simulation. The event-sensitivity semantics of VHDL processes is efficient for simulation because recomputations only occur when it is absolutely necessary to update the simulator. Dynamic event-based semantics are most certainly not statically analyzable in the sense required by an event-derivative semantics. Fortunately in VHDL-1076, one can focus the sensitivity of descriptions wholly on signal transactions instead of on signal events. This allows for a static analysis of the process' coordination activities via their event-derivatives.

3.2.1.2 The Flat versus Nested Process Models

A second important observation of [Bak93] was that the flat process model of VHDL makes the description of nested behaviors difficult. In particular, one often wants to describe a "master" process as controlling one or more "slave" processes with the master telling the slave process when to go, awaiting their normal completion or telling them directly when to stop by preempting their computation. 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 13. Even more difficult are coordinations between multiple processes where the master must prepare to recover control after some of the slaves complete yet await the others which are still computing.

```
                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;
```

Figure 13 A Master and Slave Process Pair

The lesson to be learned here is that the specification language must allow for nested behavioral descriptions in the same way that it allows for nested structural descriptions. In VHDL-1076 this problem shows up in the architecture which 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 the structural case, an instance of a component is but a reference to another entity/architecture pair (a structure) which defined elsewhere. In contrast however, a VHDL process cannot contain or invoke 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.[1] This effect shows up in NDAM-1 as a flat process model that directly mimic's the one in VHDL.

3.2.1.3 The Subtleties in the Δ-Time Models

Finally, there is the issue of the two-level *macro*-time / Δ-time model in VHDL 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 Section 1.0 it was mentioned that all references to metric time (seconds) are disallowed in S-VHDL. Thus the use of the **after** clause in signal assignments and to the use of the **wait for *timeval*** statement are proscribed. Thus all S-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 synchronous languages such as Esterel or Argos [Hal93] effectively have a notion of Δ-delay which appears in the definition of *causality*. In the reactive model, if an event *a* is said to cause event *b* in the same instant, then event *a* must have occurred slightly before event *b* but at the same macro-time step. 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 internally. The causality condition ensures that there is never a cycle in this ordering relationship.[2] The problem with the Δ-delay system with respect to S-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 Δ-time steps is *referenced*.

In this case, it is useful to motivate this problem through a distinction between VHDL-1076 which has discrete-event semantics and Esterel which has reactive semantics. The example of Figure 14 shows the case of two processes recomputing a value **output** in a single instant. Though similar in form, there is a distinct difference in the behavior that will be exhibited by each example. In both cases the first process awaits activity on the signals **S1** or **S2**. Upon seeing activity on either of those signals a value is computed and emitted on the signal **S3**. The distinction between the two descriptions is that in the VHDL description the second process will never see the transaction on signal **S1** because it occurred one too many Δ's back; *i.e.* the VHDL **S1'transaction** only refers to signal activity in the *immediately* preceding Δ of the current macro-time step. 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.

The lesson to be learned here is that although the notion of Δ-time may seem problematic, it is actually intrinsically tied to the well-founded notion of causality. In concert with this abstract notion of causality however one must have a complete way of *referencing* the existence or lack of existence of previous causal events at the source language level. The problem with VHDL is that the ability to reference previous causal events is restricted by the discrete-event simulator model which

1. Others have observed this too [NVG91].

2. VHDL-1076 does not prohibit such "non-causal" cycles in simulation models. Some may work and compute meaningful results and some may not work and cause the simulator to infinite-loop or make signal-queues of infinite length. It is considered a programmer error if these non-causal cycles do not work; the LRM [IEEE87] does not prohibit them and few simulators have any static checks for them [GI90].

only allows for references to activity in the previous simulation cycle and it does not allow for references to all activity at the current macro-time step.

This subtlety of the Δ-time model is important when defining a synchronous semantics for subsets of languages such as VHDL which were not originally designed with such in mind. While VHDL has a restrictive notion of presence or absence of a signal transaction, the underlying NDAM semantics for signal presence or absence are correct with respect to the reactive model (*c.f.* the *present* branch instruction or *presence* signal reference instruction). Being able to reference the presence or absence of all "mini" events in the current instant is important for the definition of the derivative of a sequence of instructions $\partial P / \partial E_i \partial(I_1 I_2 \ldots I_N)$ in terms of a chain rule of individual instruction derivatives $\partial P / \partial E_i \partial I_j$

3.2.2 NDAM-2 - Additions to NDAM

The lessons learned in the development of S-VHDL and of the underling NDAM-1 language have prompted a reevaluation of the constructs in NDAM-1. The previous section outlined three points against which NDAM-1 must be reevaluated. There is a fourth point which was not mentioned in the previous analysis but important nonetheless: if NDAM is to attain the goal of being a language-neutral representation for compiler-level manipulations then NDAM must also be able to represent the computations describable in Esterel. This point was described in point 3 of the design goals of NDAM-1 in Section 3.1. Some further analysis of the design of Esterel relative to NDAM-1 shows there is still some disparity. The following sections outline the additions made in NDAM-2 to allow the NDAM representation to take into account the deficiencies outlined in the previous section as well as to address the disparity between the NDAM and Esterel. In order to maintain the flow of this report and not get bogged down in the minute details of instruction syntax, the new additions to NDAM are summarized at a high level here and the details are presented in Appendix A.

3.2.2.1 The Network and Processes

In NDAM-1 a *file* as a unit of compilation contained a complete NDAM process; it was assumed that some sort of external linker-tool would be defined that would string together multiple NDAM processes. The addition in NDAM-2 is the definition of a *process* and a *network* of processes as entities within a compilable unit. The process structure is simply a wrapping around the declarations and instructions which composed a process in NDAM-1. This collection is given a name within the new namespace **P(*NAME*)**. The new construct is as follows:

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
]
```

Figure 14 Referencing Activities in Previous Deltas

```
process P(NAME) is
declarations-and-instructions
end process P(NAME)
```

The network is an analogous structure which names the network in the new namespace **N(*NAME*)**. The network construct collects the declarations of types and signal declarations of the external interface to the network. The *process-list* references the names of the processes that make up the network.

```
network N(NAME) is
type-and-signal-declarations
comma-separated-process-list
end network N(NAME)
```

This construct has the utility of formally defining the interface of a network to the outside world. The types at the network level are used to define the signals; the signals that are determined to be input signals are ordered to form the basis of the event E_i. While there may be many more signals internal to the network, the signals declared at the top-level are the only ones which will be considered for inclusion in the E_i. Without this declaration of the interface, there would be no obvious way for the linker and derivative-generator to determine which signals were external inputs and which were merely internal signals.

The addition of these two constructs to NDAM do not change the fundamental event-derivative semantics of the language. What it does do however, is allow for multiple parts of a system to be described in a single compilation unit and it allows for the explicit declaration of the interface to the system. Thus the assembler-design problem for NDAM-2 is simplified through the addition of these constructs, but the semantics is not changed.

3.2.2.2 Sub-Processes and Sub-Process Calling

The previous section mentioned that one of the most obvious needs shown up in the definition of S-VHDL was the lack of the ability to define control-transfer and control-recovery operations *between* processes. This could only be done in the most rudimentary way in S-VHDL and thus in the NDAM-1 language. There are three fundamental primitives needed for this capability:

- a means by which to define a process that is not always running,[1]
- a means by which the stationary process can be started on demand from another parent process, suspending the parent until the completion of its children,
- a means by which a running process can suspend its own execution and transfer control back to its parent.

The third primitive already exists in NDAM-1 as the *exit* instruction. Using the first two fundamental primitive capabilities as inspiration, two new constructs have been added to NDAM. They are the *subprocess* and the parallel *call* instruction. The subprocess is an extension of the process construct defined earlier and has the following form:

```
subprocess P(NAME) of P(NAME-1) of P(NAME-2) ... is
declarations-and-instructions
end subprocess P(NAME)
```

The subprocess **P(*NAME*)** is associated with a single parent process **P(*NAME-1*)** whose parents and grandparents **P(*NAME-2*)**, **P(*NAME-3*)** *etc*. are also mentioned.[2] The semantics of the subprocess is that it is initially not running, and

1. Running in the reactive sense means being sensitive to some input event. There was no way in NDAM-1 to declare a process that was sensitive to nothing but could be started on demand.

must be invoked by a control transfer from the parent process by a *call* instruction. The *call* instruction allows for the invocation of multiple subprocesses in parallel and has the following form:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-1);
   ...
   call P(NAME-N) at L(NAME-1)
```

The parallel call instruction shown here is a specific case of a more general call structure that takes into account exception handling and watchdogs both of which are non-local control-transfer operations that operate across subprocess boundaries. Unlike the *network* and the *process* constructs, the addition of the *subprocess* and the *call* instruction to NDAM change its semantics. The new semantics is that of unactivated subprocesses, control transfers to the subprocesses to make them active and control transfers back to the parent upon their completion. The event-derivative semantics must take this new execution principle into account.

3.2.2.3 Preemption and Watchdogs

A second fundamental control primitive that was identified as missing in NDAM-1 was the ability of one process to preempt the execution of another. There is no mechanism in the flat process model of NDAM-1 for one process to unilaterally force another process to cease computing. At best there must be some sort of cooperation involved, and at worst it cannot be done. To address this deficiency the concept of a watchdog has been added to NDAM as an attribute to the *call* instruction:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-1);
   ...
   call P(NAME-N) at L(NAME-1)
watching S(NAME-1), S(NAME-2), ..., S(NAME-M)
```

Esterel users will recognize this variant of the call instruction as an adaptation of the *do/watching* primitive of that language. While the utility of both instructions to the compiler and to the programmer is similar as they provide for preemption at the level of processes, there is a difference between Esterel and NDAM in the use of preemption in the formal event-derivative semantics. In the case of Esterel, preemption is a a very fundamental concept. Most of the composite instructions in Esterel are defined in terms of other more primitive operations coupled with a use of preemption (*e.g.* ***do-upto***, ***loop-each***, ***every-do***, ***await-case***, *etc.*). In the case of NDAM-2, while preemption is a required feature, it is not the core mechanism by which other the meaning of more exotic control structures are defined. The core feature of NDAM is the implicit assignment to the program counter.

The issue of the utility of preemption in the formal semantics is more a distinction of focus rather than a distinction of suitability or appropriateness. The five basic goals in the design of NDAM listed in Section 3.1, indicate that NDAM is designed to be used as the language-neutral intermediate representation of a system based on ideas used in optimizing-compilers [ASU86] [ARZ91]. As such, the NDAM instructions all need to be roughly the same *weight* where weight in this case is interpreted relative to the complexity of the operations they perform on abstract register-transfer machine. Thus each NDAM instruction should do roughly the same amount of work on the abstract machine; no instruction should do grossly more than any other. Were this otherwise then it would be hard to perform many of the common compiler opti-

2. The utility of mentioning all parent processes in the subprocess declaration will become apparent in Section 3.2.2.4 with the presentation of the *raise* instruction.

mizations such as register allocation, common subexpression elimination and the like. Further, it will be difficult to define the event-derivative semantics in a manner which is computationally useful for state-space-based verification.

3.2.2.4 Exceptions and Exception Handlers

The NDAM system is designed to represent imperative languages and as such it preserves most if not all of the control versus data distinction that is visible in those systems. For example, NDAM is phrased in terms of an abstract instruction-set processor and the instructions are designed along the same lines as the instructions of a RISC processor: simple, orthogonal operations on a register-transfer model. The control-versus-data distinction is interesting for it brings to light an currions effect: if there is an operation that can occur in one plane (say the control-plane) then there ought to be the same sorts of operations in the other plane (the data plane). For lack of the effect in one or the other planes of the machine experience shows that one often must resort to intricate and obscure tricks to produce it the other. A case in point is non-determinism itself which is fundamental to the NDAM representation. It is available both in the form of the non-deterministic signal assignment (the data plane) as well as the non-deterministic *goto* (the control-plane) which can be fruitfully thought of as a non-deterministic assignment to the program-counter.

In the previous section the watching clause of the *call* instruction was introduced to allow for preemption. Preemption can be thought of as a non-local "goto" where the control transfer is initiated by the target of the transfer as opposed to being initiated by the source of the transfer as is the usual case. Preemption is also initiated through an event on a signal (from the data-plane). Thus there is a need for an analogous operation in control plane in order to make the NDAM "symmetric." The analogous operation is of course the notion of the *exception* where an exception is considered to be a non-local transfer of control.

Raising an exception is a control-transfer operation that manipulates the control plane by modifying the program counters of one or more sibling and parent processes. Unlike the watchdog where the transfer is initiated by the destination, the control transfer is initiated by the source and allows for the atomic communication of a unit data item (a register) from the transferor to the transferee (this is the exception value).

The exception name can be thought of abstractly as a temporary register that can be assigned to from the context of the subprocess; normally subprocesses cannot read or assign the registers of its parent processes. The exception value register is abstractly a temporary register in the sense that it does not contribute to the data-space of the parent process at any other time other than just after the *raise-catch* of an exception. This is an important subtlety in the definition of exceptions for it affects the definition of $\partial P / \partial E_i$

Exceptions have been added in NDAM-2 with exactly the semantics described in the previous paragraphs. Exceptions exist in their own name space **E(*NAME*)** that can be referenced from subprocesses through a *raise* instruction. The exception is considered to be a sort of special temporary register value that can only be accessed in the current instant on control paths flowing away from a *handle* subclause of a catching *call* instruction. Exceptions are provided with the following four additions to NDAM:

```
exception E(NAME) T(NAME)
```

The exception **E(*NAME*)** is declared to have type **T(*NAME*)**; this type must then match the register values passed via a *raise* instruction and the register values used in the *caught* assignment instruction.

```
raise E(NAME) of P(NAME) with R(NAME)
```

The exception **E(*NAME*)** of the parent process **P(*NAME*)**[1] is raised with the value in **R(*NAME*)** assigned to the exception register **E(*NAME*)**. Reference to this value can be made in the parent process through the *caught* assignment instruction which has the following form:

```
R(LHS) := caught E(NAME-i)
```

The *caught* instruction is only valid on a control-path from a *handle*-subclause in the current instant when the exception is raised. This is to say that one cannot use the *caught* instruction after one or more *wait* instructions have occurred after the *raise* instruction.[2] Handlers for exceptions are instantiated using a variant of the parallel *call* instruction. In this variant the parallel call is made as before, but the instruction is suffixed with a set of *handle*-subclauses.

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-1);
   ...
   call P(NAME-N) at L(NAME-1)
catch
   handle E(NAME-1) goto L(NAME-1);
   handle E(NAME-2) goto L(NAME-2);
   ...
   handle E(NAME-M) goto L(NAME-M)
```

It is useful at this point to step back and observe that the perceived size of the *call* instruction has been growing as watchdog and exception semantics has been built into it. The set of trailing *handle*-clauses makes the parallel *call* seem quite large and thus in danger of violating a design goal of the NDAM language: that the call instruction might become larger than any of the other instructions in terms of its operational weight. The semantics relative to $\partial P/\partial E_i$ is actually smaller than it seems. The best way to visualize this is to think of the *catch*-clause as sort of a "case" keying on the termination status of the parallel *call* instruction. If the call instruction finishes normally which is to say that all of the called subprocesses terminal normally, then control flows to the next instruction. If an exception was raised then the control flows through the *handle*-clause indicated by the exception name.

3.2.2.5 Summary

The extensions to NDAM-1 that were described in the previous sections were motivated by experiences observed in the use of NDAM-1 to define S-VHDL and to emulate the high-level control constructs available in another imperative language with synchronous semantics - Esterel. The preceding sections have outlined the new instructions and explained the reasoning behind their design; more complete descriptions of these instructions and their features is given in Appendix A.

With the addition of these new instructions in NDAM-2 it should be possible to do two things:

1. Define a set of extensions to the S-VHDL language to allow for the convenient use of a VHDL-based language for system description. The extensions should be convenient in terms of the control-primitives available and also amenable to use in the verification-based debugger environment presented in Section 1. The additions to NDAM of subprocesses, watchdogs and exceptions as described in this section should make the former feasible. The latter was always in mind in the design of NDAM-1 with the inclusion of non-deterministic control (goto) and data (signal assignment) primitives. A very preliminary proposal to this end is presented in Section 4.

---

1. The **P(*NAME*)** is one of the members of the *of-list* in the subprocess declaration.

2. Though complicated to express, this is a simple statically-checkable condition on the control-flows emanating from the *handle-goto* subclause of the *call* instruction.

2. It should be possible to derive a compiler for the Esterel language using the features of the NDAM-2 instructions. This would demonstrate in some sense the sufficiency of the instruction primitives and the efficacy of the instruction-level event-derivative semantics presented here.[1]

3.3 Semantics through Event-Derivatives

The fundamental idea behind semantics through event derivatives is to define what will occur as the reaction to some event from the environment. This is exactly the information that is needed in the case of a system operating according to the reactive model. Such a system is said to obey the *synchrony hypothesis* [BC84] which in simple terms says that the system operates faster than the environment is sending it events. Thus the behavior of the system can be meaningfully modeled as a state-transition system with each event from the environment being said to *cause* a state transition in the system. With respect to digital design the synchrony hypothesis usually translates to "is the combinational logic operating faster than the clock speed;" there is an analogous phrasing for embedded software systems.

Given the reactive model of execution and there are two interesting questions with respect to system description languages. The first and most fundamental question is by what means the set of events that the environment can send to the system is described. The second question is of course how, given the definition of the set of events that the environment can produce, the system's description should be given to make the computation of its reactions simple to describe and easy to transform into an automata.

The following section reviews the basics about *regular expressions* [HU79] as a demonstration of the first use of an abstract derivative operator. The derivative of a regular expression is used definition of a transformation from the pattern itself to a finite automata that recognizes the pattern. The more complex case of the Esterel language and the program-derivative operator defined for that language is reviewed in Section 3.3.2. Finally the process-derivative for the cases of all processes active in the network and some processes in active is given for NDAM-2 in Section 3.3.3 and Section 3.3.4 respectively.

3.3.1 The Case of Regular Expressions

The definition of so-called regular expressions is in terms of the derivative of the expression R relative to a character c as $\partial R/\partial c$ The following is a very simplified account of the procedure used to convert a regular expression into a finite automaton. The account is by example as the procedure is straightforward and well-known. Given an alphabet Σ of symbols, then the regular expressions can be defined recursively as shown in Figure 15.

The goal is an algorithm that will take a regular expression R and convert it into a finite automata that will recognize strings and return true or false as to whether the string belongs to the set described by the regular expression. The algorithm of [Brz62] is defined in terms of the derivative of the regular expression R relative to a symbol a_i, in terms of the derivative $\partial R/\partial a_i$. The derivative is taken for all symbols a_i in the alphabet Σ until there are no more. It is a theorem that there are a finite number of derivatives of a regular expression. The example of Figure 16 shows how this works in practice for a small example.

3.3.2 The Case of Esterel

In the case of regular expressions the symbol-set, Σ, was given explicitly in the definition of the problem. In the case of Esterel[2] the definition of the symbol-set is given indirectly by defining a set of *signals* through which the program P will communicate with its environment. An event is understood to be the conjunction of the occurrences of a activity on the

1. Granted there is already an Esterel V.3 compiler [CIS88] with later versions are under development and there is a state-based verify cation system called auto [Hal93]. The question posed here is one of the sufficiency of NDAM-2. At present there does not seem to be a need for any more instructions; no NDAM-3 is envisioned.

2. The presentation here is adapted from the presentation in [BC84] which defines the program-derivative operator in a syntax-directed manner, building up the derivative operator on compound constructs in terms of the derivative of their components.

The alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$

The Regular Expressions

$$R = a_i$$
$$R = R_1 R_2$$
$$R = R_1 | R_2$$
$$R = R^*$$

Figure 15 The Regular Expressions

$$R_0 = \text{ab*c}$$

$$\frac{\partial R_0}{\partial a} = \text{b*c} = R_1$$

$$\frac{\partial R_1}{\partial b} = \text{b*c} = R_1$$

$$\frac{\partial R_1}{\partial c} = \text{""} = R_2$$

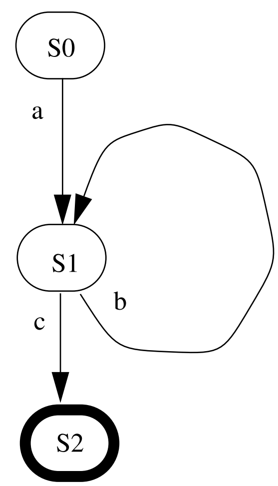


Figure 16 Regular Expression to Finite Automata Conversion

signal in the instant. The set of possible events that the environment can generate is thus the power set of signals. This relationship between signals and events is shown graphically for a small example in Figure 17.

The meaning of an Esterel program P is the set of derivatives of that program with respect to all possible events that it can ever see. The notation which has traditionally been used for these derivations is $(P, E_i) \Rightarrow (P', O)$ to indicate the output produced in addition to the new state P'. In the case of Esterel, the result of the program-derivative operator is another program in the same style as the derivative of a regular expression is another regular expression. Defining the derivative operator directly on the program text is convenient for it allows the definition of that operator to be phrased in a syntax-directed way: the derivative of compound constructs being defined in terms of the derivatives of their parts in a meaningful way. The job of the Esterel compiler is thus to ensure that the derivative is always meaningful according to the rules of causality and given that the program is causal, to take all possible derivatives. An simple example of how the program derivative operator is applied to a simple program is shown in Figure 18.

Figure 17 An Esterel System and its Event Set

```
module M:
input A(integer), B(integer);
output O(integer);
do
   await A;
   emit O(?A)
watching B
timeout
   emit O(?B)
end.
```

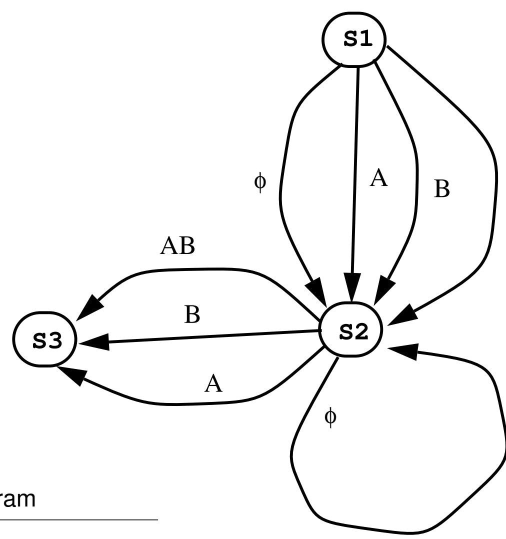


Figure 18 Derivatives of an Esterel Program

3.3.3 Extending to Derivatives of an Instruction-Processor Representation

For the NDAM representation the situation is analogous to that of Esterel. Each NDAM process or subprocess communicates with its environment by means of signals. These signals can be determined to be either input signals, output signals or both by virtue of their appearance in either *emit* instructions or in *wait* instructions or some combination of both. The various domains needed to define the NDAM event-derivative semantics are defined by equation in Figure 19.

The basic idea behind the development there is that we must have a characterization of the state of the network and that must be defined from in the constituent components recursively. The reader should note that in the following development a great deal of mathematical notation is used; this notation is intended to provide some sort of intuition and rigor to the process of defining the derivative semantics of NDAM networks. The mathematical notation provides a communication device by which the ideas described here can be conveyed from place to place; too they hint at possible implementation strategies and data structures. This is a point that will be explored more fully in the sections ahead.

Ultimately the goal of the following development is to define the derivative of a network

$$\frac{\partial Network}{\partial Env}$$

in such a way that an explicit state graph[1] can be extracted from the implicit-state representation of the processes in the NDAM network. The best way to understand the set of domain equations of Figure 19 is to read them from the bottom up

$Network$ — The state of the network is defined in terms of the states of all of the processes in the network, $Proc$, and the states of the internal signals of the network, $SIGINT$.

$SIGINT$ — The internal signals are simply the signals mentioned in the processes or subprocesses but which are not also mentioned at the level of the network. These signals are thus internal to the system and a portion of the system's state is maintained in them.

$Proc$ — The state of each process is defined in terms of the value of its process program counter, PC_p, the values in its internal registers, $VARS$, and the states of the subprocesses, $SubProc$.

The value of PC_p is drawn from a set of possible locations specific to each process plus a distinguished $start$ label which is the value of PC_p before the process commences execution. The fact that no process can be idle is indicated by the fact that there is no label $idle$ that can be assigned to PC_p to indicate in the transition relation that the process is inactive. Thus the derivative operator is phrased in terms of all processes being active.

$$\begin{aligned}
SIGS &= \{ s \mid \text{signal} \} \\
SIGINT &= \{ s \mid \text{internal-signal} \} \subseteq SIGS \\
ISIGS &= \{ s \mid \text{input-signal} \} \subseteq SIGS \\
Env &= 2^{ISIGS} \\
SubProc &= (PC_s, VARS, SubProc) \\
Proc &= (PC_p, VARS, SubProc) \\
Network &= (Proc, SIGINT)
\end{aligned}$$

$$PC_s \in \{ l \mid \text{labels} \} \cup \{ idle \}$$
$$PC_p \in \{ l \mid \text{labels} \} \cup \{ start \}$$

Figure 19 Basic Domains for NDAM Process Semantics

1. The terminology here is unfortunate and the reader's patience is requested. The *explicit-state* graph referred to here is one containing states and edges marked by transition predicates; it is derived from the *implicit-state* representation of the NDAM processes' control flows. This graph may be represented *implicitly* in a BDD data structure in the style of [CM90]. The intended meaning of these terms will be clear from the context.

The $VARS$ is a predicate over the registers and register-values indicating the values which may be present there.

$SubProc$ The state of each subprocess is defined in terms of its subprocess program counter, PC_s, its register bindings, $VARS$, and the state of its subprocesses, $SubProc$.

The value of PC_s is drawn from the set of labels specific to the subprocess as well as the distinguished symbol $idle$ which is used to indicate that the subprocess is not active. The derivative operator is defined to take this meaning of the symbol $idle$ into account.

The definition of $VARS$ for the subprocess domain is the same as that for the process: it is a predicate over the registers and register values of the subprocess.

$ISIGS$ The set of input signals is the set of signals which are mentioned in the network declaration and are also used in a *wait* instruction in some process or subprocess of the network. Though other signals may be mentioned in *selection*, *select* or *present* instructions, it is only the signals mentioned in wait statements that can cause computation. Thus only these signals appear in the $ISIGS$ and in Env as events.

Env The environment is simply the power-set of the input signals. Events can occur from the environment in any combination at any instant. Thus an event E_i can be any subset of $ISIGS$ so the set of all possible events is the set of subsets of $ISIGS$.

With these domains and their relationship to the NDAM language syntax in mind, the derivative of a network, $\partial Network/\partial Env$ be defined. Actually however, the interest here is not so much in definition of a single-step derivative, but rather the transitive closure of the derivative operator on the network with respect to the environment. Further though, the interest is not just in the specific values of the derivatives, but also the connectivity of the graph induced by the transitive closure of the derivative operator. The example of Figure 18 showed the graph induced by the Esterel program derivative for that simple example. Here, we are interested in behavior of the transitive closure of the network derivative $\partial Network/\partial Env$. Fortunately however, the transitive closure of the derivative is simply the derivative transitive closure which in turn is the ∞-th derivative:[1]

$$\left(\frac{\partial}{\partial Env}Network\right)^{*} = \frac{\partial^{*}}{\partial Env}Network = \frac{\partial^{\infty}}{\partial Env}Network$$

Additionally we know from [Brz62] that there are guaranteed to be a finite number of derivatives of a network. Thus the computation of the right-hand side of the above equation must be computable in a finite number of steps. The definition of the derivative transitive closure of a network is shown in Figure 20. In that figure, the derivative transitive closure is defined in terms of the k-th network derivative. In turn the k-th network derivative, which is just the derivative operator, $\partial/\partial Env$, applied k times, is shown to be the set of network-states emanating from the derivatives of the members N_i of the set of $(k-1)$-th derivatives. The k-th derivative set is computed relative to every possible event E_i that can occur in Env and also relative to every possible N_i from the $(k-1)$-th derivative set.

All the terms in Figure 20 have already been defined save for N_i which is the symbol used to describe a network state as a member of the set produced by the $(k-1)$-th network derivative. The notation for the network state in terms of the cross-

1. There is enough the notation here that it does not seem worthwhile to add more by in turn formally definining the ∞-th derivative as the limit of the k-th derivative as k approaches ∞ in the traditional manner. It is not clear that "limit theory" illuminates this problem to any extent. The point of this notational development is to present the intuition of the method and to indicate what structures need to be developed to support the computation in the NDAM assembler.

$$\frac{\partial^k}{\partial Env} Network = \{E_i \in Env, N_i \in \frac{\partial^{(k-1)}}{\partial Env} Network \Big| \quad \frac{\partial N_i}{\partial E_i}\}$$

$$\frac{\partial^*}{\partial Env} Network = \frac{\partial^\infty}{\partial Env} Network$$

Figure 20 The Network Derivative and its Transitive Closure

product of process states is shown in Figure 20. This notation gives some clue as to how the development must progress beyond this point. The state of each process must be characterized in terms of its program counter, PC_p, and the values of its registers $VARS$. It is on this product that $\partial/\partial E_i$ is defined.

Unfortunately though at this point the mathematical notation ceases to to provide intuition and a prose description must be given of how the derivative operator, $\partial/\partial E_i$ is defined on a single network state N_i. The idea is that somehow the instructions of the processes, P_i that are "ready" to go are executed. These instructions may perform some data manipulations inside the processes P_i but they may also emit some events on the internal signals of the network. These internal signals will in turn "awaken" other processes which were waiting on those signals. The condition of *causality* [BC84] ensures that the order in which the internal events occur is well-defined; situations of *non-causality* can easily occur where a signal's occurrence forces its own absence or where a signal's occurrence triggers its occurrence. It is the job of the analysis phase of the assembler to ensure that such NDAM descriptions are flagged as illegal; in such cases $\partial N_i/\partial E_i$ is undefined.[1]

The network derivative $\partial Network/\partial Env$ induces a graph mapping some source range of network states to some image of network states. A second application of $\partial Network/\partial Env$ uses the image of the first application as the range and computes a second image, and so on. Some intuition to this process is given in Figure 22. There, the initial network state is shown as a point and the successive applications of $\partial Network/\partial Env$ map sets of network states to sets of network states. Each of the edges in the figure correspond to transitions from an N_i caused by the reception of some event E_i.

The interesting thing about Figure 21 is that it is reminiscent of the presentation of the implicit state enumeration techniques of [CM90].[2] The trick in the definition of $\partial Network/\partial Env$ therefore is to ensure that it is defined implicitly in

$$N_i = P_1 \times P_2 \times \ldots \times P_n$$

$$N_i = \prod_{i=1}^{n} P_i$$

Figure 21 The Network State as a Cross-Product of Process States

1. Conditions for causality in terms of a partial orders on instruction streams of processes P_i and for $\partial N_i/\partial E_i$ in terms of the partial derivatives P_i with respect to instruction execution exist. They are not presented here because I do not yet have an elegant notation for presenting them. That presentation will have to await the development of the appropriate notation.

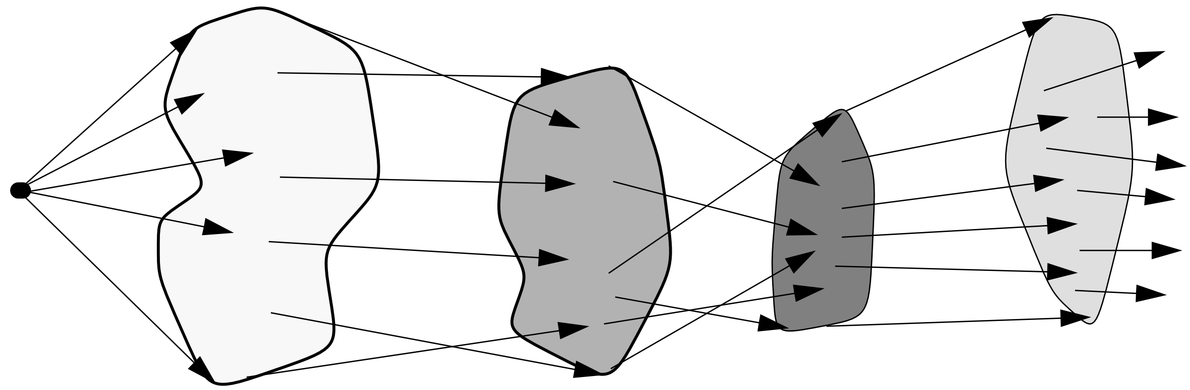


Figure 22 The Network Derivative Mapping Sets of States to Sets of States

terms of operations on a data structure, and that at no point does a depth-first search of the network's state-space have to be computed.

The transition relation for a subset of the NDAM instructions is shown in this form in Table 1. The remaining work in this section involves formally defining the transition relation table for all NDAM instructions. Having done this, some work must be done to formulate this transition relation table in terms of BDD (MDD) operations and then make some sort of assessment of the complexity of the operations involved in defining the NDAM architecture in terms of MDD operations. The hope would be that some sort of claim about the minimality of these operations relative to the number of MDD operations.

3.3.4 Extending to Derivatives of Multiple Processes

The definition of the network derivative $\partial Network/\partial Env$ in the previous section only took into account processes, and further, it only took into account the situation where all processes are active at all stages. Thus the definition $\partial Network/\partial Env$ given previously was rather simplistic. This was done on purpose to motivate the notion of transitive closure and the formulation of the NDAM architecture as an implicit-state transition table in the style of [CM90].

What is necessary for a completely correct definition of $\partial Network/\partial Env$ is a notion of causality and of subprocess invocation. The definition of causality is allows for the condition where the set of processes which are awakened by the initial event E_i compute to their *wait* instructions and turn cause another set of processes to commence execution in the current instant. This second set may then cause a third set of processes to compute and so on until there is nothing left to do in the current instant. The very definition of causality ensures that this process can never be cyclic for that would be a non-causal situation: either the cycle never occurred or it oscillated forever which is a paradox. Unfortunately there does not seem to be a nice notational vehicle to elaborate this notion of causality though some sort of notational mechanism to express causality might be appropriately developed.

So too with the notion of subprocess invocation; there does not seem to be any convenient notational vehicles to add to the previous development. The essential idea with subprocess invocation is that some of the NDAM instruction transition

2. Note that in this case the term *implicit-state* is in fact being used to imply that there is a data-structure which represents the next-state relation as a function in a BDD.

relation entries will define the program counter of some subprocesses, thereby folding those subprocesses into the current active set of subprocesses.

3.4 Abstraction though Control Trees

Unfortunately it was not possible to investigate control abstractions very much for this report. The basic idea of the abstraction method however is to use *control trees* [ARZ91] to provide automatic abstraction of the control space of NDAM processes. Control trees were originally developed as an abstraction aid for optimizing compilers to be used in case the control flow of the procedure bodies became too complex. Control trees provide a way of breaking up the control flow graph of the procedure into large macro blocks within which computationally-expensive algorithms are more feasible. An example of a control tree for the Traffic Light Controller process of Figure 9 is shown in Figure 23 and Figure 24. In a more general setting, some general control-tree templates are shown in Figure 25

Control trees were used in [Bak93] as an aid in the rederivation of a valid Esterel program from an NDAM-1 process. The importance of control trees relative to this report is that control trees provide an automated way of abstracting large chunks of an instruction-level program into units each of which have their own transition relation. Thus the derivative semantics of a particular network could be rephrased in terms of derivatives, not at the level of instructions in a process, but at the level of regions at some level of granularity in the control flow tree. Thus, for a particular process in the network the transition relation table, as shown in Figure 19, would be modified to include not only instructions but also *regions* of the control tree. Regions would be expected to be heavily nondeterministic with respect to the next-pc.

Table 1 The Transition Relation for Selected NDAM Instructions

Label	Instruction	Current PC	Current VARS	Next PC	Next VARS
L(i)	goto L(a), L(b)	pc = L(i)	vars	pc = L(1)	vars
L(i)	goto L(a), L(b)	pc = L(i)	vars	pc - L(2)	vars
L(i)	if R(1) goto L(a)	pc = L(i)	R(1) = 1	pc = L(a)	vars
L(i)	if R(1) goto L(a)	pc = L(i)	R(1) = 0	pc = L(i)+1	vars
L(i)	case R(1) when R(2) goto L(a); when R(3) goto L(b)	pc = L(i)	R(1) = R(2)	pc = L(a)	vars
L(i)	case R(1) when R(2) goto L(a); when R(3) goto L(b)	pc = L(i)	R(1) = R(3)	pc = L(b)	vars
L(i)	case R(1) when R(2) goto L(a); when R(3) goto L(b)	pc = L(i)	R(1) /= R(2) R(1) /= R(3)	pc = L(i)+1	vars
L(i)	R(1) := R(2)	pc =L(i)	vars	pc = L(i)+1	R(1) = R(2) and other vars
L(i)	R(1) := R(2) + R(3)	pc = L(i)	vars	pc = L(i)+1	R(1) = R(2)+R(3) and other vars
L(i)	R(1) := R(2) * R(3)	pc = L(i)	vars	pc = L(i)+1	R(1) = R(2)*R(3) and other vars
L(i)	R(1) := selection S(1)	pc = L(i)	vars	pc = L(i)+1	R(1) := selection and other vars
L(i)	emit S(1) with R(1), R(2)	pc = L(i)	vars	pc = L(i)+1	vars

4 V-VHDL - VHDL-like Verification Handling Description Language

This section provides a speculative look at what an extension to the VHDL language that takes into account the new features of the NDAM-2 language might look like. For this report, only the addition of non-determinism to VHDL was examined though future versions of V-VHDL might fruitfully involve additions such as subprocess invocation (allowing a behavior to contain another behavior), watchdogs and exceptions.

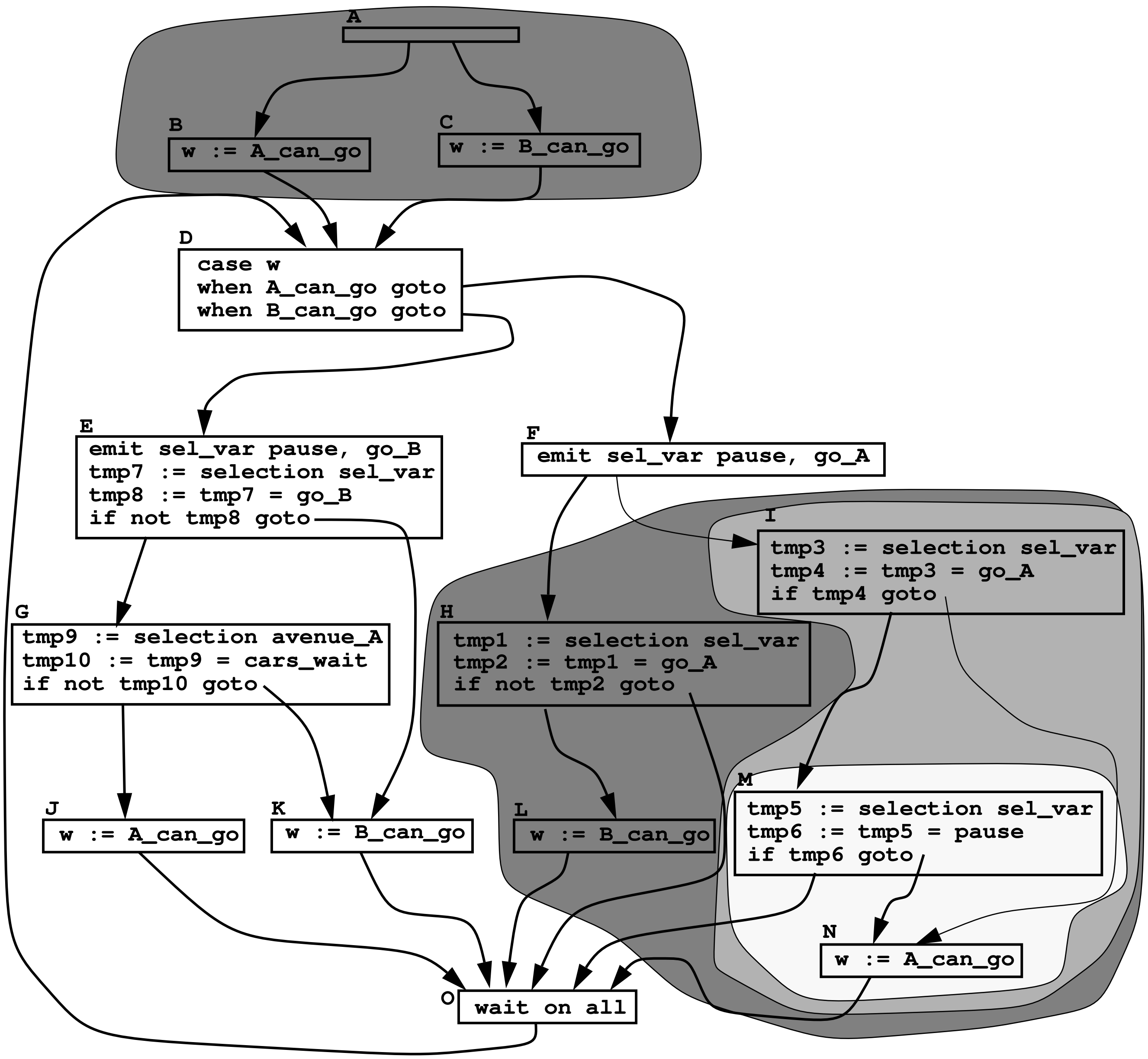


Figure 23 The Control-Flow Graph of Figure 9 with some Control Tree Regions

The exercise conducted for this report was rather restricted in the sense that only extensions needed to use a VHDL-like language in the context of the verification-based debugger shown in Figure 1 were examined. Further, within that guideline, some very strict guidelines were followed to ensure that the notion of "VHDL-like" was preserved in the extensions as much as possible.

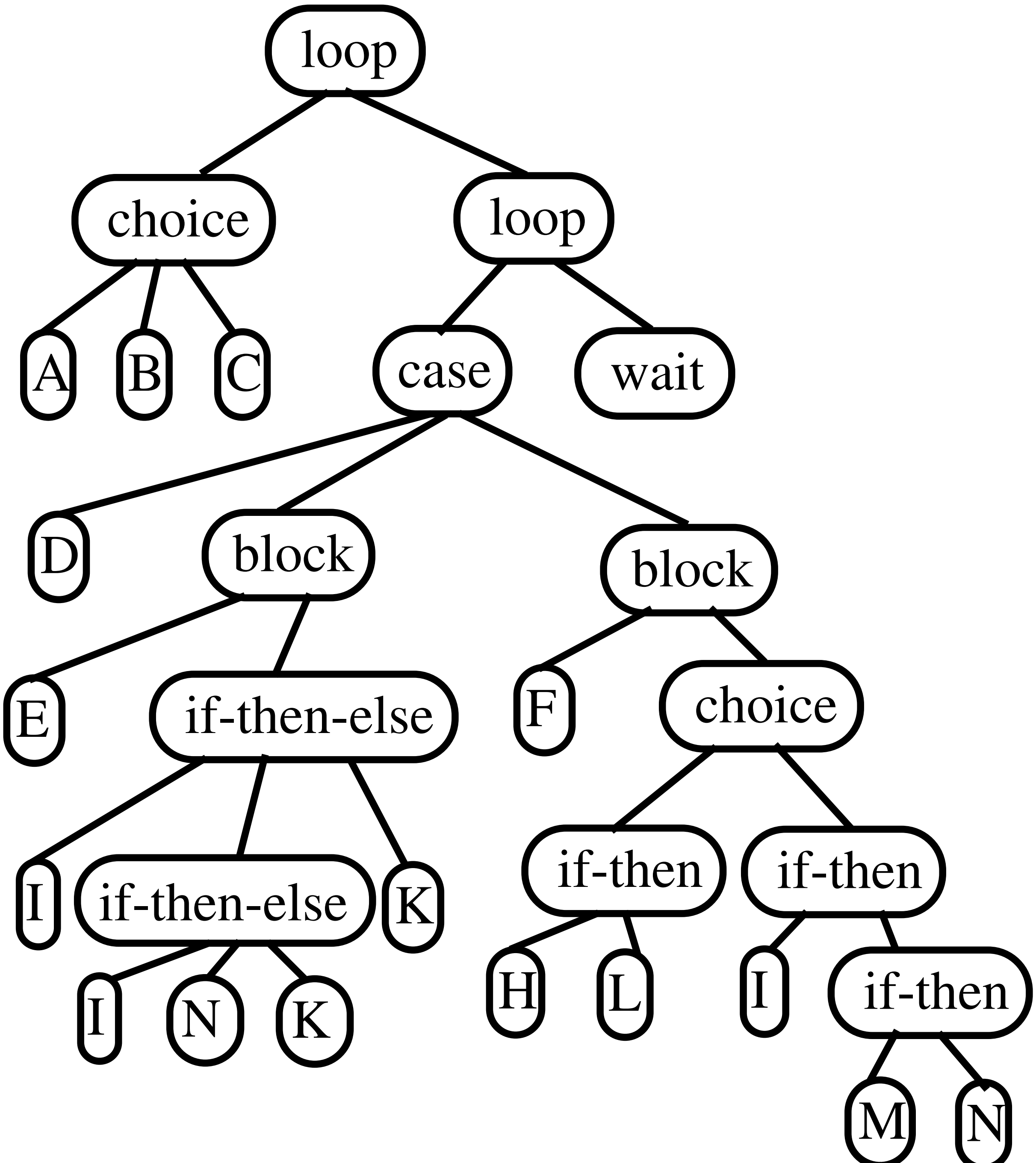


Figure 24 The Complete Control Tree of Figure 9

The following points outline the design goals for the V-VHDL language.

- No new execution semantics. The Synchronous VHDL subset upon which the V-VHDL language is based is complete with respect to the S/R execution model. There is no need to add extra constructs to V-VHDL to account for further execution semantics.

  Any additional constructs in V-VHDL over and above those already defined in the Synchronous VHDL subset are required for the design property declarations and the design verification hooks.
- Existing VHDL constructs are used a basis for V-VHDL constructs whenever possible. If a VHDL construct is converted for use as a V-VHDL construct then both the syntax and semantics of that construct is preserved in V-VHDL.

  Thus declarative constructs which are modified in V-VHDL to contain new expressions behave as they do in Synchronous VHDL (*c.f.* the VHDL and V-VHDL **configuration**).

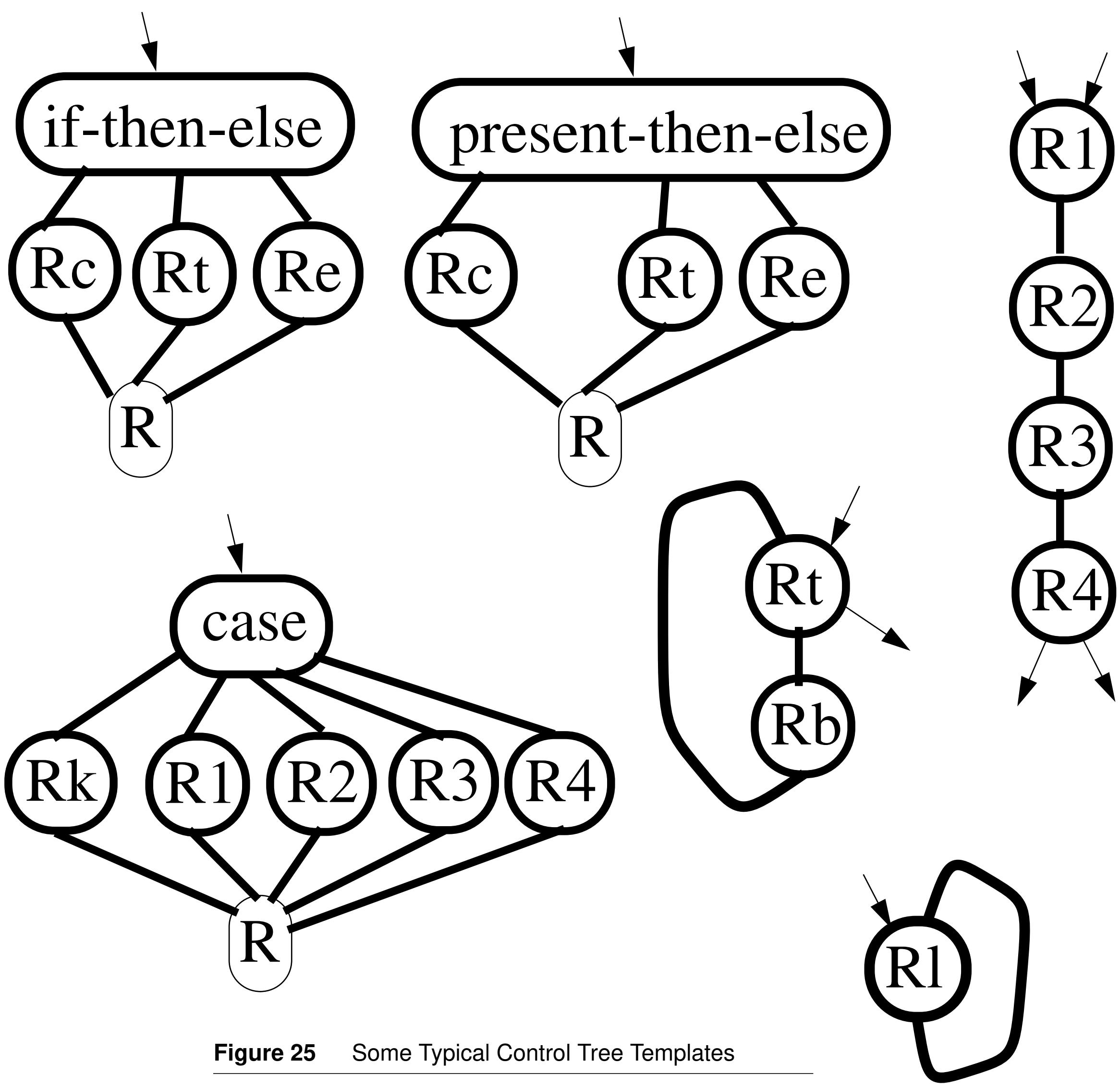


Figure 25 Some Typical Control Tree Templates

- When defining new constructs the stylistic philosophy of VHDL is used appropriately. This means that simple and explicit keywords should be used in the terminating-repetitive style of VHDL (*c.f.* the **loop**, **block** or **process** constructs). This will make the resulting language "feel" the same as the original VHDL.
- When it is necessary to change VHDL, the modifications to the underlying VHDL are minimal and affect only the declarative aspects of the processes.

  An example of this is the addition of labels before *any* sequential VHDL statement; Synchronous VHDL, and 1076 VHDL only allow labels before loop statements.

4.1 Design Verification Hooks

This section offers some possibilities for declarations within V-VHDL that might be used to help "hook up" the design to be verified with the property declarations themselves.

4.1.1 Labels Prefixing Any Statement

```
label : sequential_statement
```

Example

Figure 26 is an example of a simple process that is sensitive to transactions on the bit signal **`value`**. The process manages a single small integer which must be kept between the values of -4 and 4, inclusive. On receiving a bit value '1' the process increments the small integer, while on receiving a bit value of '0' the process decrements the small integer. So in

```
architecture control_style of small_memory is
   signal value: bit;
begin

   P1:
   process
      subtype small_integer is integer range -4 to 4;
      variable small: small_integer := 4;
   begin
      if value = '1' then
         -- increment the small integer
         L2: small := small + 1;
      else
         -- decrement the small integer
         L3: small := small - 1;
      end if ;
      wait on all;
   end process P1;

end some_arch;
```

Figure 26 A Small Memory

this example the important consideration is whether or not the design will cause the small integer variable to overflow or underflow. The conditions which will cause overflow and underflow respectively are given by the following two state declarations:

```
state increment_will_overflow is
      in P1
      at L2
      with small = small_integer'high;

state decrement_will_underflow is
      in P1
      at L3
      with small = small_integer'low;
```

If either of these two states can be reached in process **P1** then a constraint violation on the variable **small** will occur as addition or subtraction statement is executed. The constraint violation will occur because the value to be stored in **small** after the operation will either be too great or too small. The purpose of the design verification then in this example is to insure that the process **P1** is never driven, by its environment, to a point where either state **increment_will_overflow** or **increment_will_underflow** apply.

4.1.2 The Choice Statement

The *choice-statement* allows the declaration of control nondeterminism. The construct offers a set of sub-blocks each of which may be executed nondeterministically. In the context of V-VHDL, this construct allows for the declaration of possible paths of execution without forcing the programmer to declare *how* that path will be excited.

The grammatical formation of the choice-statement is designed to mimic the grammatical structure of the *case-statement* with respect to the overall structure and the keyword usage.

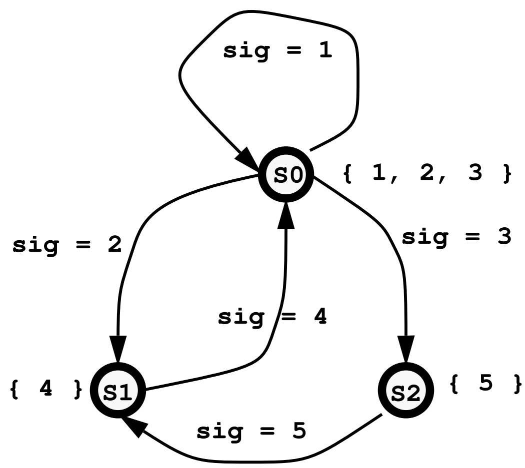
```
choice-statement ::=
   choice in
      sequential_statements
      { else sequential_statements }
   end choice ;
```

The *choice-statement* might be used in any situation where an *if-statement* or a *case-statement* might be used. The determinized form of a *choice-statement* is the *case-statement.*

The choice statement expresses what is called here *control nondeterminism*. This nondeterminism is distinct from state or *selection nondeterminism* which is expressed in the nondeterministic signal assignment described in Section 4.2.3. The distinction between these two types of nondeterminism arises because of the implicit-state nature of the V-VHDL language. Where there is only one type of nondeterminism in the explicit-state S/R language, there needs to be two types of nondeterminism in V-VHDL. The need for this arises from the definition of

Example

The example of Figure 27 shows the use of the *choice-statement* to describe a set of successor states for a process. The process has three possible states, **S0**, **S1**, and **S2** with various transitions between those states. An integer value is selected and output on **sig** for each transition. There are three alternatives to the *choice-statement* with each path declaring a different binding for the successor state.The process, taken as a whole, then declares the following state graph:[1]



The example shows how the c*hoice-statement* is used to express that a set of possible successor states is possible from given state by declaring a several paths through the code. The nondeterminism of the choice statement indicates that any path can be taken and thus allows for nondeterministic control actions. The nondeterministic choice of the selection is made available through the nondeterministic signal assignment; that is the subject of the next section.

4.1.3 Nondeterministic Signal Assignment

The nondeterministic signal assignment declares the selection set for the current instant.

target **<= nondetermine** *expression* **{ ,** *expression* **} ;**

The **selection** attribute is a function which returns the nondeterministic selection for the current instant.

S'**selection**

A signal attribute is used in this way to reinforce the notion that a signal's value is not a singleton as is the case in discrete-event simulation, but rather is the set of possible values declared in a non-deterministic signal assignment. The **selection** signal attribute "resolves" that selection and produces a singleton value which can be used in any computation.

It is useful to think of the **selection** signal attribute as being symmetric with the **delay** signal attribute. In the same sense that delay produces the value of the signal at a past point in time, the selection signal attribute produces the value of the signal in a possible future where the nondeterminism has been resolved; that future is determined by the predicates in which the selection value is used.

4.2 Other Extensions

The semantics of NDAM-2 allow for a wider range of computations to be expressed than can be expressed in VHDL. The additions presented in the previous section for V-VHDL only address the use of non-determinism in the context of a VHDL-like language. One might wish to make further modifications to V-VHDL proposal to make it easier to specify

1. The notation used here is the same as that found in [Ku87], namely that states are labeled with their names and annotated with the selection set for that state while edges are labeled with the transition predicate. In this example sig is the process' selection.

```
architecture state_based of rippling_protocol is
   signal sig: integer := 4;
begin

   P1:
   process
      type state is (S0, S1, S2);
      variable S: state := S1;
   begin
      case S in
         when S0 =>
            choice in
               sig <= 1;
               S := S0;
            else
               sig <= 2;
               S := S1;
            else
               sig <= 3;
               S := S2;
            end choice;
         when S1 =>
            sig <= 4;
            S := S0;
         when S2 =>
            sig <= 5;
            S := S1;
      end case;
      wait on all;
   end process P1;

end state_based;
```

Figure 27 A Use of the Choice Statement

behaviors within that language. The ramifications of those extensions were not investigated for this report but might include features such as:

- nested processes; that is allowing a process to appear as a *sequential-statement* within another process.
- exceptions; that is allowing a subprocess to transfer control to one of its ancestors.
- watchdogs; that is allowing a parent process to preempt the execution of one or more of its children.

As NDAM-2 and its event-derivative semantics allow for the representation of such structures directly, the question then becomes one of identifying the appropriate source-level language structures

References

ARZ91 Fran Allen, Barry K. Rosen and Kenneth Zadek, *Optimization in Compilers*, ACM Press, 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 *Proceedings of the International 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.

[Bak92] Wendell Baker, *An Application of a Synchronous Reactive Semantics to the VHDL Language,* Masters Report, University of California, December 1992.

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.

BS86 Gérard Berry, R. Sethi, “From Regular Expressions to Deterministic Automata,” in *Theoretical Computer Science*, Vol. 48, pages 117-126, Elsevier Science Publishers, 1986.

Bry86 R. E. Bryant, “Graph-based algorithms for Boolean Function Manipulation,” IEEE Transactions on Computers, Vol. C-35, No. 8, 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.

CES86 E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications” in *Transactions on Programming Languages and Systems*, Vol 8, No. 2, April 1986, pages 244-263.

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.

CM90 O. Coudert and J.C. Madre “A Unified Framework for the Formal Verification of Sequential Circuits,” in *Proceedings of the International Conference on Computer-Aided Design*, November 1990.

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

Hal93 Nicholas Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993. This recent book contains an excellent bibliography collecting references to the synchronous programming languages Esterel, Lustre, Argos, and Signal.

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

HK87 Z. Har'El and R.P. Kurshan, *COSPAN User's Guide*, AT&T Bell Laboratories, Murray Hill, NJ 07974, October 9 1987. Available from the authors.

HTKB92 Ramin Hojati, Hervé Touati, Robert P. Kurshan, Robert K. Brayton, "Efficient ω-Regular Language Containment" in *Proceedings of the 4th Computer-Aided Verification Workshop*, June 29-July 1 1992, pages 371-382.

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.

Kur92 R.P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes*, Mathematical Sciences Research Center, AT&T Bell Laboratories Murray Hill, NJ 07974, March 30 1992. Available from the author.

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.

McM92 Kenneth L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, PhD Thesis, Carnegie-Mellon University, March 16, 1992.

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

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

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.

A NDAM - Summary of Language Definition

A.1 NDAM-1

A.1.1 Declarations

There are three basic types of declarations: type declarations, register declarations and signal declarations. All names must be defined by the appropriate declaration before they are used. The only exception to this rule are labels which may be referenced before they are defined, thus allowing goto statements to transfer forward.

A.1.1.1 Type Declarations

The type declaration introduces the name of a type and defines how many values that type may take on. The effect of type declarations is to declare a class of multi-valued variable. Later constant register declarations may associate names with the values of the type. Instances of such declarations can be seen in the section entitled EXAMPLE.

type T(NAME) N

A type with the indicated number of possible values is declared. It is assumed that the range of values is from 0 at the low end to N-1 at the high end. This statement is but a more convenient version of the basic type declaration statement.

Example:

```
type t(bool) 2
type t(mv8) 8
```

In this example, two multi-valued types are declared: one type has two possible values, 0 and 1; the other has eight possible values ranging from 0 to 7.

type T(NAME) range V1 V2

A type with values ranging from V1 on the low end to V2 on the high end is declared. The value V1 must be no greater than V2 so that a non-vacuous range is defined.

Examples:

```
type t(int5) -16 15
type t(int32) -2147483648 2147483647
```

Here, two multi-valued types are declared: one type has thirty-two possible values, ranging from -16 to 15; the other has four billion possible values.

A.1.1.2 Register Declarations

There are three kinds of registers, constant registers, temporary registers and permanent registers. It is most useful however to consider registers as belonging to one of two kinds: non-constant and constant registers of which the non-constant registers are further divided into temporary registers and permanent registers. Non-constant registers are suitable for use in representing variables, temporary or otherwise, which are to be operated upon using the various assignment operators. Constant registers on the other hand are suitable for representing constants such as the members of a scalar type. Temporary registers are intended to be used to store (compiler-generated) values that do not need to be saved across wait instructions; thus temporary registers may be added and destroyed with impunity by the compilation process.

register R(NAME) T(NAME) [:= V]

A singleton register is declared; it contains one field. The register can optionally be initialized with a value at the point of declaration. If no initializer value is declared the default initialization value of zero is assumed.

Examples:

```
register r(a_bit) t(bool) := 1
register r(r32) t(int32) := -9000233
```

Here, two singleton registers are defined; the first is a single bit which contains the positive phase and the second is presumed to be multi-valued representation of a 32-bit signed integer.

register R(NAME) T(NAME), T(NAME), ... T(NAME) [:= V, V, ... V]

A record register is declared; it contains a number of fields. The fields can be optionally initialized at the point of declaration. If initializer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

Examples:

```
register r(a_rec) t(bool), t(int32), t(bool) := 1, -9000233, 1
register r(three_r32) t(int32), t(int32), t(int32) := 1, 2, 3
```

register R(NAME) T(NAME) [:= V], T(NAME) [:= V], ... T(NAME) [:= V], T(NAME) [:= V]

The alternate syntax for the record register allows for the fields and optional initializers to be directly associated. Initializers may be provided or left in default for each field independently. As usual, fields with no explicit initializer value are understood to receive the default initialization value of zero.

Examples:

```
register r(a_rec) t(bool) := 1, t(int32) := -9000233, t(bool) := 1
register r(three_r32) t(int32) := 1, t(int32) := 2, t(int32) := 3
```

register R(NAME) T(NAME) D, D, ... D [:= V, V, ... V]

An array register is declared; it contains as many fields as dictated by the “volume” of the dimensions. These fields can be optionally initialized at the point of declaration. If initializer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

Although arrays are declared to be multi-dimensional entities, in reality they are only linear objects. All indexing that takes place in the instruction stream is in terms of linear offsets. The use of the multi-dimensional declaration format is a convenience only.

Examples:

```
register r(an_array32) t(bool) 32 := 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 1
register r(r32_3by3) 3, 3 := 1, 2, 3, 4, 5, 6, 7, 8, 9
```

A.1.1.3 Signal Declarations

The signal declarations have the same essential form as the register declarations. They declare typed signals in the same way that typed registers are defined.

signal R(NAME) T(NAME) [:= V]

A singleton signal is declared; it contains one field. The signal can optionally be initialized with a value at the point of declaration. If no initializer value is declared the default initialization value of zero is assumed.

signal R(NAME) T(NAME), T(NAME), ... T(NAME) [:= V, V, ... V]

A record signal is declared; it contains a number of fields. The fields can be optionally initialized at the point of declaration. If initializer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

signal R(NAME) T(NAME) [:= V], T(NAME) [:= V], ... T(NAME) [:= V]

The alternate syntax for the record signal allows for the fields and optional initializers to be directly associated. Initializers may be provided or left in default for each field independently. As usual, fields with no explicit initializer value are understood to receive the default initialization value of zero.

signal R(NAME) T(NAME) D, D, ... D [:= V, V, ... V]

An array signal is declared; it contains as many fields as dictated by the ‘‘volume’’ of the dimensions. These fields can be optionally initialized at the point of declaration. If initial izer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

Although arrays are declared to be multi-dimensional entities, in reality they are only linear objects. All indexing that takes place in the instruction stream is in terms of linear offsets. The use of the multi-dimensional declaration format is a convenience only.

A.1.2 Assignment Operations

Assignment statements are a special case of a more general class of instructions which perform operations as well as data movement. The other class of instructions, the Operations are described in the following section. The assignment statements are broken out into a special category because they provide access into the aggregate registers: the record register and the array register.

To be noted in this section as well as the following section is that there are two sides to an assignment operation: the left-hand side (abbreviated LHS) and the right-hand (abbreviated RHS). Only non-constant registers may appear on the left-hand side of an assignment while either constant or non-constant register references may appear on the right-hand side. This rule is driven by the common sense that one must not destroy copies of constants; this is a statically checkable condition.

R(LHS) := R(RHS)

An assignment is made from the register R(RHS) to the register R(LHS). The assignment is independent of the size of the registers, thought the sizes of the arrays must match. That is the assignment must be between two singletons registers, two record registers with the same number of fields or two array registers with the same volume. This is a statically-checkable condition.

R(LHS) := R(RHS).offset

An assignment is made from the field offset of the register R(RHS) to the register R(LHS). The register R(LHS) must be a singleton as the assignment only references a single field of R(RHS). The offset must be an unsigned integer which is within the bounds of the number of fields in the register R(LHS). This is a statically-checkable condition.

R(LHS).offset := R(RHS)

An assignment is made to the field offset of the register R(LHS) from the register R(RHS). The register R(RHS) must be a singleton as the assignment only references a single field of R(LHS). The offset must be an unsigned integer which is within the bounds of the number of fields in the record register R(LHS). This is a statically-checkable condition.

R(LHS) := R(RHS) [R(E)]

An assignment is made from the member indicated by the value of R(E) of the array register R(RHS) to the register R(LHS). The register R(LHS) must be a singleton as the assignment only references a single member of R(RHS). This is not a statically-checkable condition and so such assignments should be protected by a range statement which ensures that the bounds of R(RHS) are not dynamically violated.

R(LHS) [R(E)] := R(RHS)

An assignment is made to the member indicated by the value of R(E) of the array register R(LHS) from the register R(RHS). The register R(RHS) must be a singleton as the assignment only references a single field of R(LHS). The R(E) must be an unsigned integer which is within the bounds of the number of members in the array register R(LHS). This is not a statically-checkable condition and so such assignments should be protected by a range statement which ensures that the bounds of R(LHS) of are not dynamically violated.

R(LHS) := R(RHS) [R(LOW) , R(HIGH)]

An assignment is made from the member indicated by the values of R(LOW) to R(HIGH) of the array register R(RHS) to the register R(LHS). The register R(LHS) must be an array as the assignment references the multiple members of R(RHS). This is not a statically-checkable condition and so such assignments should be protected by a range statement which ensures that the bounds of R(RHS) are not dynamically violated.

R(LHS) [R(LOW) , R(HIGH)] := R(HIGH)

An assignment is made to the array indicated by the values in the rangeof R(LOW) to R(HIGH) of the array register R(LHS) from the register R(RHS). The register R(RHS) must be an array as the assignment references the multiple fields of R(LHS). The R(LOW) and R(HIGH) must be an unsigned integer which is within the bounds of the number of members in the array register R(LHS). This is not a statically-checkable condition and so such assignments should be protected by a range statement which ensures that the bounds of R(LHS) of are not dynamically violated.

A.1.3 Operations

The register-to-register operations are broken up into two classes: unary and binary operators.

A.1.3.1 Unary Operations

The unary operations are defined by a destination register called R(LHS) and a single source register called R(RHS). There is also an *operation* which is performed on the value of R(RHS) as it is transferred. The following sections describe all the possible unary operation statements.

For the unary operations, the R(RHS) and R(LHS) registers may be any a singleton, record or array register, but both registers must match with respect to type, storage class and in size, for record registers, or volume for array registers. This condition is statically checkable.

In the case of aggregate registers, the unary operation is performed element-wise on all the members of the aggregate. Thus the sense of the unary operator acting upon a register or array register is that of a vector operation.

R(LHS) := abs R(RHS)
The absolute value of R(RHS) is taken and placed in R(LHS).

R(LHS) := dec R(RHS)
The value of R(RHS) is taken, decremented and placed in R(LHS).

R(LHS) := inc R(RHS)
The value of R(RHS) is taken, incremented and placed in R(LHS).

R(LHS) := not R(RHS)
The value of R(RHS) is taken, complemented and placed in R(LHS). The complementation performed is a logical one, so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := - R(RHS)
The value of R(RHS) is taken, complemented and placed in R(LHS).

A.1.3.2 Binary Operations

The binary operations are defined by a destination register called R(LHS) and a two source registers called R(RHS-1) and R(RHS-2). There is also an *operation* which is performed on the value of R(RHS) as it is transferred. The following sections describe all the possible unary operation statements.

For the unary operations, the R(RHS) and R(LHS) registers may be any a singleton, record or array register, but both registers must match with respect to type, storage class and in size, for record registers, or volume for array registers. This condition is statically checkable.

In the case of aggregate registers, the unary operation is performed element-wise on all the members of the aggregate. Thus the sense of the binary operator acting upon a register or array register is that of a vector operation.

R(LHS) := R(RHS-1) and R(RHS-2)
The value of R(RHS-1) and R(RHS-2) are taken, combined via logical-and and placed in R(LHS). The operation performed is a logical one so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := R(RHS-1) or R(RHS-2)
The value of R(RHS-1) and R(RHS-2) are taken, combined via logical-or and placed in R(LHS). The operation performed is a logical one so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := R(RHS-1) nand R(RHS-2)
The value of R(RHS-1) and R(RHS-2) are taken, combined via logical-nand and placed in R(LHS). The operation performed is a logical one so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := R(RHS-1) nor R(RHS-2)
The value of R(RHS-1) and R(RHS-2) are taken, combined via logical-nor and placed in R(LHS). The operation performed is a logical one so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := R(RHS-1) xor R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via logical-xor and placed in R(LHS). The operation performed is a logical one so nonzero values are transformed to zero while the 0 value is transformed to 1.

R(LHS) := R(RHS-1) = R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via an integer comparison for equality, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) /= R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via an integer comparison for inequality, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) < R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via an integer less-than comparison, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) > R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via an integer greater-than comparison, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) <= R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via an integer less-than or equal comparison, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) >= R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via a greater-than or equal comparison, and placed in R(LHS). A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

R(LHS) := R(RHS-1) + R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer addition, and placed in R(LHS).

R(LHS) := R(RHS-1) - R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer subtraction, and placed in R(LHS).

R(LHS) := R(RHS-1) * R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer multiplication, and placed in R(LHS).

R(LHS) := R(RHS-1) / R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer division, and placed in R(LHS).

It is an error for the value of R(RHS-2) to be the value 0 and so instances of operations where there is a possibility of such a situation should be protected by the appropriate range instructions. This protection declares to the language containment verification system that the nonzeroness property of R(RHS-2) must be checked.

R(LHS) := R(RHS-1) mod R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer modulus, and placed in R(LHS).

It is an error for the value of R(RHS-2) to be the value 0 and so instances of operations where there is a possibility of such a situation should be protected by the appropriate range instructions. This protection declares to the language containment verification system that the nonzeroness property of R(RHS-2) must be checked.

R(LHS) := R(RHS-1) rem R(RHS-2)

The value of R(RHS-1) and R(RHS-2) are taken, combined via integer remainder, and placed in R(LHS).

It is an error for the value of R(RHS-2) to be the value 0 and so instances of operations where there is a possibility of such a situation should be protected by the appropriate range instructions. This protection declares to the language containment verification system that the nonzeroness property of R(RHS-2) must be checked.

A.1.4 Signal Operations

The signal operations provide the means by which the values in the registers of the process are assigned to signals and by which the presence and value components of signals are recovered into the registers of a process.

emit S(LHS) R(RHS-1), R(RHS-2), ... R(RHS-N)

A set of possible values is emitted onto the signal S(LHS). The value of S(LHS) is then, nondeterministically, one of the R(RHS- I). The selection statement can be used to reference this non-deterministic value. A deterministic emission is simply an emission where there is but one R(RHS).

Examples:

```
emit s(result) r(1)
emit s(ok) r(bad), r(good)
```

The first example is a deterministic emission which places the value currently found in the register r(1) onto the signal s(result). The second example is a nondeterministic emission which places the value of both r(bad) and r(good) onto s(ok).

R(LHS) := selection S(RHS)

The selection of S(RHS) is assigned to R(LHS). In the deterministic case, this merely represents extracting the current value of the signal S(RHS) and placing that value in R(LHS).

Example:

```
r(1) := selection s(result)
```

The value on the signal s(result) is transferred to the register r(1).

R(LHS) := presence S(RHS)

The presence of S(RHS) is assigned to R(LHS).

Example:

```
r(1) := presence s(result)
```

The presence of the signal s(result) in the current instant is transferred to the register r(1).

A.1.5 Control Transfers

Control-transfer instructions modify the program counter. They should be thought of as instructions which conditionally (or not) assign to the program counter, thereby dictating the next control-state of the abstract machine.

goto L(TARGET-1), L(TARGET-2), ... L(TARGET-N)

An unconditional branch is performed to (one of) the target(s). If there is only one target then the goto is a deterministic branch; if there is more than one branch then the goto is considered to be a nondeterministic branch.

Examples:

```
            goto L(left), l(middle), L(right)
L(left):    r(1) := inc r(1)
            goto L(end)
L(middle):  r(1) := dec r(1)
            goto L(end)
L(right):   r(1) := abs r(1)
            goto L(end)
L(end):     null
```

The example shows the declaration of three paths which are non-deterministically executed. The first goto is nondeterministic and declares three possible targets L(left), L(middle) and L(right). Subsequently for each target an instruction performs some work, either incrementing, decrementing or taking the absolute value of the previously-declared register r(1). A deterministic goto is used to continue at common point labeled L(end).

if [not] R(TEST) goto L(TARGET)

A conditional branch is performed depending on the binary value of R(TEST). If the value is positive then the goto is affected; if the value is zero then control passes to the next statement. The keyword not inverts the sense of this test. Of note, the type of R(TEST) must be binary in the sense that the type must have the range 0 to 1.

Example:

```
type t(bool) 2
temporary r(tmp) t(bool)
r(tmp) := r(1) = r(2)
if r(tmp) goto L(good)
if not r(tmp) goto L(bad)
exit
```

The example shows a type t(bool) which is declared to have an appropriate range, “from 0 to 1” for use in an if statement. This type is then used to declare a temporary register r(tmp) which is in turn used to store the result of a comparison between two previously-declared registers r(1) and r(2). The core of the example is the use of the if statement in both phases to test the value of the comparison. As there is no possibility of executing beyond the if statements an exit is used to declare that the fallthrough path is unexecutable.

case R(KEY)

when R(V-1-1), R(V-1-2), ... R(V-1-N) goto L(TARGET-1) ;

when R(V-2-1), R(V-2-2), ... R(V-2-N) goto L(TARGET-2) ;

...

when R(V-M-1), R(V-M-1), ... R(V-M-N) goto L(TARGET-N)

A multi-way branch is performed based on the value in R(KEY). Conceptually the R(KEY) is matched against V-i-j and whichever value is equal, then the goto of that when clause is taken; if none match then execution continues with the following statement. All of the V-i-j are required to be in the range of the type of the R(KEY) and each V-i-j must be unique to ensure that the statement is deterministic.

Example:

```
temporary r(key) t(type)
r(key) := selection s(value)
case r(key)
when 0, 1, 2 goto L(low);
when 3 goto L(high);
when 4 goto L(redo)
exit
```

The example shows a small fragment of code which acquires the current selection on the signal s(value) and uses the value as the key for a case statement. The case statement targets one of three possible outcomes L(low), L(middle) and L(redo) which are not described. Finally an exit is used to indicate that falling through the case statement to the subsequent statement is impossible.

present [not] S(NAME) goto L(TARGET)

A conditional branch is performed depending on whether the signal S(NAME) is present in the current instant. If so then the transfer is affected; if not then control passes to the next statement. The keyword not inverts the sense of this test. Of note, the type of S(NAME) can be arbitrary as its value is not referenced; only its presence or absence is detected.

Example:

```
type t(any) 18
signal s(some) t(any)
emit s(some) := r(1)
present s(some) goto L(here)
if not s(some) goto L(there)
exit
```

The example shows a type t(any) which is declared to have an some appropriate range. This type is then used to declare a signal s(some) onto which is emitted a constant r(1) which was presumably declared previously. The core of the example is the use of the presence statement in both phases to test for the presence or absence of the signal. As there is no possibility of executing beyond the present statements an exit is used to declare that the fallthrough path is unexecutable.

A.1.6 Waiting for Events

The wait statement serves to break up the flow of computation and to separate one reactive instant from the next. The basic wait statement waits on a set of signals; there are two other variants which may be more convenient to use from time to time, but are merely defined in terms of the basic wait statement. Its definition follows.

wait on S(NAME-1), S(NAME-2), ... S(NAME-N)

The basic wait statement waits on a set of signals; should an event occur on any or all of the set, then execution continues at the next statement. The signals may be of any type whatsoever for all that matters is the presence or absence of events on the signals in the current reactive instant.

Example:

```
L(start):   emit s(1) r(1)
            emit s(2) r(2)
            emit s(3) r(3)
            wait on s(1), s(2)
            goto L(start)
```

This fragment of code emits during each instant the value of the previously-declared registers r(1), r(2) and r(3) onto the previously-declared signals s(1), r(2) and r(3). The wait statement serves to break up the control flow into instants so that there is a finite amount of work in each instant.

wait on S(NAME)

This is variant of the wait statement merely a more convenient representation of the multiple-signal wait statement. Awaiting a single event can be performed just as easily as awaiting for multiple events. The distinction is merely one of representation in the internal data structures.

wait on all

The process waits on all the signals in the sensitivity list of the containing process. This instruction is but a convenient shorthand for the primary wait statement which references the complete signal list.

Example:

```
L(start):   emit s(1) r(1)
            emit s(2) r(1)
            emit s(3) r(1)
            wait on all
            goto L(start)
```

This fragment of code emits during each instant the value of the previously-declared register r(1), r(2) and r(3) onto the previously-declared signals s(1), s(2) and s(3). The use of wait on all is exactly equivalent to explicitly naming the signals in the sensitivity list of the containing process as follows:

```
L(start):   emit s(1) r(1)
            emit s(2) r(1)
            emit s(3) r(1)
            wait on S(1), S(2), S(3)
            goto L(start)
```

A.1.7 Miscellaneous

In addition to the various declarations, the arithmetic operations, the signal operations and the control flow statements there are certain other miscellaneous statements. These statements are in general oriented not at describing what is to be executed in the NDAM process, but rather at describing what is not to be executed. The information provided by these instructions allows compilers to make claims about the values of registers under various conditions or the certainty of non-execution of a path.

There are two ways to declare paths which must never be executed, both of which involve the execution, or potential execution of an instruction. The two instructions are the report and exit instructions.Through their use, paths in the assembly code can be declared to be “bad.” It is then up to the language containment verification system to demonstrate that those bad paths can never be executed.

Though different in their overt functionality, the report and exit instructions should be considered as one in the same for their intent is the declaration of invalid computations. The report instruction declares an invalid execution sequence but also contains informational message about what went wrong. The exit instruction on the other hand contains no informational message yet equivalently declares an invalid computation.

A.1.7.1 Path Constraint Declaration

report R(MESSAGE) R(SEVERITY)

An error is declared. In addition to declaring an invalid computation sequence, there is an informational message and severity level which is presumably useful for debugging purposes. The contents of R(MESSAGE) and R(SEVERITY) are uninterpreted by the instruction; the values contained in those registers is presumably useful to the verification / debugging system.

The message of the report is held in R(MESSAGE) which must be an array register. Each array element is interpreted as the integer code for an ASCII character, thereby interpreting the whole array as a string message. The severity code is held in R(SEVERITY) which must be a singleton register.

The purpose of this instruction is to indicate an error condition. With respect to language-containment verification the utility of this statement is to declare a path which must never be executed: in the real system the report instruction must never be executed and it is the job of verification to demonstrate that it won’t be.

exit

This instruction permanently halts the process in which it resides.The purpose of this instruction is to indicate an error condition for a process which runs infinitely long can never execute the exit instruction. With respect to language-containment verification the utility of this statement is to declare a path which must never be executed: in the real system the report instruction must never be executed and it is the job of verification to demonstrate that it won’t be.

A.1.7.2 Range Constraint Checking

The range-check instructions assert that the value of R(VALUE) lies between the values of R(LOW) and R(HIGH); they implement the check R(LOW) <= VALUE <= R(HIGH). Such an instruction is useful for declaring simple design properties that ensure that an array is never indexed out of bounds or the like. It is the job of a high-level language compiler

range R(VALUE) R(LOW) R(HIGH)

The value in the register R(VALUE) is checked against the range constraint R(LOW) and R(HIGH).

range R(VALUE).offset R(LOW) R(HIGH)

The value in the field R(VALUE).offset is checked against the range constraint R(LOW) and R(HIGH).

range R(VALUE)[R(E)] R(LOW) R(HIGH)

The value in the array member indicated by R(E) is checked against the range constraint R(LOW) and R(HIGH). The value of R(E) is not range-checked against the bounds of the array, so it is up to the compiler to insert a range-check on R(E) against the bounds of the array R(VALUE) as appropriate.

A.2 NDAM-2

A.2.1 Linking it All Together

In the definition of Synchronous VHDL [Bak93] the definition of NDAM-1 was given with the implicit understanding that some other tool "linker-like" would be given which linked together a set of NDAM processes which at that point were assumed to be synonymous with files. With the introduction of NDAM-2, the definition of that linker-like tool has been subsumed into the definition of a compiler.

The NDAM-2 language definition defines an explicit *process* construct. A unit of compilation, the file, now consists of multiple processes which are understood to be executing in synchronous parallelism. To formalize the notion of the interface to this network of processes, and to provide a place to declare the subset of the network's signals that should be considered the input, the notion of the network has been formalized with an explicit *network* construct. Additionally the concept of the *process* has been generalized in the form of the *subprocess* in order to allow one process to control the invocation and termination; the need for this was noted in [Bak93].

The following sections detail each of these additions to original NDAM-1 language.

A.2.1.1 The Network

The network is a description of all of the top-level types and signals used in the system. The signals declared at the top-level in the network defines the input alphabet of the network. The input alphabet is made up of those signals which are input-only signals or input-output signals. Whether a signal is input-only, output-only or an input-output signal is determinable by the NDAM compiler by examining whether the signal is written or read in any of the processes of the network.

The network is defined as follows:

```
network N(NAME) is

declarations
processes-list

end network N(NAME)
```

The *declarations* consist of type and signal declarations as would exist in a process. The declaration of the types at the top-level is needed to allow for the declaration of the types of the signals. The declaration of the signals at this level names the signals as part of the input alphabet (if they are read in any of the processes mentioned in the *process-list).*

Example:

```
network N(1) is

type t(unit) 1
type t(bool) 2
type t(int5) 5

signal S(tick) t(unit)
signal S(go) t(bool)
signal S(val) t(int5)

P(1), P(2), P(3), P(4)
```

```
end network N(1)
```

This defines a network with three signals and four processes. The declaration of the processes would follow the declaration of the network

A.2.1.2 Processes

The notion of the process was implicit in NDAM-1 as each process description was a file. In NDAM-2 the process has been formalized in a construct so that multiple named processes may exist in the same unit of compilation. The process structure is as follows:

```
process P(NAME) is

declarations-and-instructions

end process P(NAME)
```

The following is a simple process showing the wrapper declarations;

```
process P(1) is

type T(unit) 1
signal S(tick) T(unit)
L(1):    wait on S(tick)
L(2):    wait on S(tick)
L(3):    wait on S(tick)
         goto L(1)

end process P(1)
```

A.2.1.3 Sub-Processes

One of the conclusions that was drawn from the definition and implementation of Synchronous VHDL [Bak93] was that there is a need in synchronous systems for one logical process to have the ability to control the invocation and termination of another process. Conversely, it is extremely inconvenient to have all processes executing at all instants; there is utility in being able to only invoke some processes at certain points in time. Further, not-always-executing processes are those processes to which control is transferred and from which control is recovered at certain points in the execution flow.

Thus the concept of *subprocesses* has been added to NDAM-2. Subprocesses are like processes only they are not always executing. Control must be transferred explicitly from a calling process to one or more subprocesses; upon termination of all of the invoked subprocesses control is returned to the caller. Subprocesses are thus associated explicitly with a single parent (sub)process which can invoke it and to which control is returned on termination. Subprocesses themselves are processes in the sense that they can contain subprocesses.

The structure of a subprocess declaration is as follows:

```
subprocess P(NAME-1) of P(NAME-2) of P(NAME-3) ... is

declarations-and-instructions

end subprocess P(NAME-1)
```

Of note in this structure is the *of-list* chain of parent process names in the subprocess declaration header. This *of-list* has the utility of declaring where the subprocess fits in the overall tree of processes in the network. The information in this list of parent processes is redundant in the sense that only the first one is truly needed, it is useful from a language design point of view as it serves to mention all of the parent process names and thus preserves the explicit define-before-use property of NDAM descriptions. Process names are referenced in the *raise* instruction.

The declarations and instructions which can occur in a subprocess are exactly the same as those which can occur in a process. The only difference between a process and a subprocess and a processes is that a process is assumed to be constantly executing at every instant while a subprocess is only executing when directed to by an explicit call instruction; in that case the parent process is suspended.

The *call* instruction is used to invoke subprocesses as is described in the next section.

A.2.2 The Call Instruction

The call instruction allows for control transfers between a parent process and one more child processes and also provides support for exception handling and watchdogs. The simplest form of the call instruction merely calls a single subprocess as follows:

```
call P(NAME) at L(NAME)
```

This form of the call instruction is not very interesting because control transfers involving only a single target (one subprocess) can be best performed via the goto instruction. The call instruction is more interesting when it involves multiple subprocess targets.

The more general form of the call instruction provides for all of parallel sub-execution, exception handling and watchdogs. Each of these three features are independent and so may be left out of the instruction structure. The fully-general form of the call instruction is as follows:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-2);
   ...
   call P(NAME-L) at L(NAME-L)
catch
   handle E(NAME-1) goto L(NAME-1);
   handle E(NAME-2) goto L(NAME-2);
   ...
   handle E(NAME-M) goto L(NAME-M)
watching S(NAME-1), S(NAME-2), ... , S(NAME-N)
```

A.2.2.1 Parallel Sub-execution

Simple parallel sub-execution is performed by omitting the catch and watching clauses as follows:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-2);
   ...
   call P(NAME-L) at L(NAME-L)
```

The semantics of this instruction is that named subprocesses commence executing in the current instant at the indicated instruction labels. The instruction labels name entry-points relative to the indicated subprocesses. When all of the subprocesses have terminated then the whole instruction is terminated and control is passed to the next instruction.

With respect to exceptions and watchdogs, the call instruction, without either a catch-clause or a watching-clause is transparent: control transfers due to exceptions or watchdogs occur through it

A.2.2.2 Preemption

The watching-clause of the call instruction declares that the invoked subprocesses are interrupted in the current instant if an event on any of the indicated signals occurs. The form of this variant of the call instruction is as follows:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-2);
   ...
   call P(NAME-N) at L(NAME-L)
watching S(NAME-1), S(NAME-2), ... , S(NAME-N)
```

Analogously with the simple parallel sub-invocation form of the call instruction, this instruction is transparent with respect to exception raising. Exceptions raised in the subprocesses transfer directly through this instruction.

One can determine if the call instruction was interrupted by an event on one of the watchdog signals by testing for the presence of that signal in the current instant in the instructions following the call instruction. Testing can occur with either a present instruction or with a select instruction as shown in the following example:

```
try
   call P(1) at L(1);            -- L(1) in P(1)
   call P(2) at L(1);            -- L(1) in P(2)
   call P(3) at L(1)             -- L(1) in P(3)
watching S(1), S(2), S(3)
select
   when S(1), S(2) goto L(1);       -- L(1) in this process
   when S(3) goto L(2)
goto L(terminated_normally)
L(1):
   ... common stuff ...
   present S(1) goto L(1a)          -- resolve S(1) versus S(2)
   present S(2) goto L(1b)
```

A.2.2.3 Exception Handling

The call instruction also allows for handlers for exceptions to be declared. After an exception is raised by any one of the subprocesses, control is recovered and transferred according to the handle clauses of the instruction. The form of this variant of the instruction is as follows:

```
try
   call P(NAME-1) at L(NAME-1);
   call P(NAME-2) at L(NAME-2);
   ...
   call P(NAME-L) at L(NAME-L)
catch
```

```
handle E(NAME-1) goto L(NAME-1);
handle E(NAME-2) goto L(NAME-2);
...
handle E(NAME-M) goto L(NAME-M)
```

Analogously with the simple parallel sub-invocation form of the call instruction and the watchdog form of this instruction, this instruction is transparent with respect to guard signals of enclosing processes. Guard signals declared in parent process' call instructions transfer directly through this instruction.

A.2.3 Exceptions

Exceptions are provided explicitly through the notion of subprocesses. The granularity of exceptions is at the level of the subprocess. Exceptions are handled across the subprocess call boundary through the handle-clause of a call instruction. Exceptions are raised in subprocesses by naming the exception name and parent process to which control is to be transferred.

A value which may be recovered in the handling parent is transferred also. In this sense an exception name is a register through which the exception-raising subprocess and the exception-handling parent process may communicate a value. The exception value is not a register however in the sense of contributing to the state space of the parent process. The exception value only persists from the point of the exception-raise to the next wait statement. In that sense exception registers are more akin to temporaries. The exception value must be explicitly read from the exception register space into the true register space the raised-value is to be preserved.

The basic declarations for exceptions are as follows:

exception E(*NAME*) T(*NAME*)

An exception named **E(*NAME*)** is declared; its type is **T(*NAME*)**.

raise E(NAME) of P(NAME) with R(NAME)

The exception **E(*NAME*)** declared in **P(*NAME*)** is raised and the value contained in **R(*NAME*)** is assigned to the exception register of the parent **P(*NAME*)**.

R(NAME) := caught E(NAME)

The value in the exception register E(NAME) is transferred to the register R(NAME) which may be either a temporary register or a true register (but of course may not be a constant register). This instruction is only valid on a path from the handle subclause mentioning E(NAME) to a wait statement; it is illegal for this instruction to appear anywhere else in the parent process as the value of the exception register is undefined in those contexts.

try
- **call P(*NAME-1*) at L(*NAME-1*);**
- **call P(*NAME-2*) at L(NAME-2);**
- **...**
- **call P(*NAME-L*) at L(*NAME-L*)**

catch
- **handle E(*NAME-1*) goto L(*NAME-1*);**
- **handle E(*NAME-2*) goto L(*NAME-2*);**
- **...**
- **handle E(*NAME-M*) goto L(*NAME-M*)**

The call instruction is used to instantiate handlers for the various exceptions defined at the current level in the process hierarchy. The call instruction was described in the previous section.

A.2.4 Multi-Way Branching on Signal Presence

One of the lessons learned in the design of V-VHDL was that activities which can be performed on the data-space need to be replicated in the control space. To that end, experience with NDAM-1 has shown that it is useful to have a multi-way branch instruction that tests against the presence or absence of signals. The *select* instruction is analogous to the case instruction as a multi-way extension of the present instruction.

select

when S(*NAME-1-1*), S(*NAME-1-2*) ... S(*NAME-1-M*) goto L(*NAME-1*);
when S(*NAME-2-1*), S(*NAME-2-2*) ... S(NAME-2-M) goto L(*NAME-2*);
...
when S(*NAME-N-1*), S(*NAME-N-2*) ... S(*NAME-N-M*) goto L(*NAME-N*)

A multi-way branch is performed based on the presence or absence of the named signals.

As more than one signal may be present in the current instant, the derivative semantics of this instruction is defined relative to the order of the when clauses, but is independent of the order of the signals within a when clause. Thus the first when clause is tested and if none of those signals are present then the second clause is tested and so on.

A.2.5 Sensors

As was discovered in the development of Esterel [BC84], it is important to be able to interact with the environment in ways that have nothing to do with the event-derivative semantics. In Esterel [CIS88] this facility was built into the signal mechanism with a special sort of signal, the *sensor*. For NDAM-2, that same name has been adopted for the ad-hoc method by which an NDAM process can interact with its environment.

Sensors are defined in their own name space, **X(*NAME*)** to make it clear that the sensor values have nothing to do with either the data space of the process or with the event-derivatives of the process. The letter **X** was chosen to indicate that the value of a sensor is unknown and unpredictable. The value of a sensor can thus not be predicted by dataflow analysis and thus imposes constraints on the high-level optimizations which can be performed around it.[1]

Sensors are supported by three instructions:

sensor X(*NAME*) T(*NAME*)

The sensor **X(*NAME*)** is declared to have type **T(*NAME*)**.

R(*NAME*) := read X(*NAME*)

The value of the sensor **X(*NAME*)** is read into the register **R(*NAME*)**. **R(*NAME*)** must be a true register or a temporary and must have the same type as **X(*NAME*)**.

write X(*NAME*) with R(*NAME*)

The value in **R(*NAME*)** is written to the sensor **X(*NAME*)**; the type of **X(*NAME*)** and **R(*NAME*)** must match.

B V-VHDL - Summary of Language Extensions

This appendix outlines the grammatical structure of the V-VHDL language extensions.

1. In this sense a sensor is the same as a *volatile* variable in ANSI C or C++.

B.1 Design Property Declarations

B.1.1 The Monitor Declaration

```
monitor_declaration ::=
   monitor identifier is
      monitor_header
      monitor_declarative_part
   begin
      monitor_statement_part
   end [ simple_name ] ;
```

```
monitor_header ::=
   [ generic_clause ]
   [ port_clause ]
   [ state_clause ]
```

```
monitor_statement ::=
   concurrent_assertion_statement
|  concurrent_procedure_call
|  process_statement
```

B.1.2 Identification of States of a Process

```
state_declaration ::=
   state identifier is
      in process_name
      [ at statement_name ]
      [ with expression ] ;
```

B.1.3 Repeated Behavior Specifications

B.1.3.1 Repeated States

```
state_name ;
```

B.1.3.2 Repeated State Transitions

```
from state_name to state_name ;
```

B.1.4 Behaviors

B.1.4.1 Behaviors to be Required

```
requirement_declaration ::=
   requirement identifier
      property_specification
      { property_specification }
   end requirement ;
```

B.1.4.2 Behaviors to be Suppressed

```
supppression_declaration ::=
   suppression identifier
      property_specification
      { property_specification }
   end suppression ;
```

B.2 Design Verification Hooks

B.2.1 Labels on Any Sequential Statement

```
label : sequential_statement
```

B.2.2 The Nondeterministic Control Flow

```
choice-statement ::=
   choice in
      sequential_statements
      { else sequential_statements }
   end choice ;
```

B.2.3 Nondeterministic Selection

B.2.3.1 Nondeterministic Signal Assignment

```
target <= nondetermine expression { , expression } ;
```

B.2.3.2 The Selection Attribute

```
S'selection
```

B.2.4 Sensitivity to All Referenced Signals

```
wait on all;
```

B.2.4.1 Library Support

Component Declarations

```
component_declaration ::=
   component identifier
      [ generic_clause ]
      [ port_clause ]
      [ state_clause ]
   end component ;
```

Component Instantiation

```
component_instantiation_statement ::=
   label :
```

```
component_name
   [ generic map ( association_list ) ]
   [ port map ( association_list ) ]
   [ state map ( association_list ) ] ;
```

C Example - The Crossroads

C.1 The S/R Description[1]

C.1.1 Generic Useful Definitions

Within S/R, one can define a template process that can be used later. The following is the definition of a template road process that manages cars passing, waiting or stopped.

```
#include "1.h"

proctype ROAD( controller: signal )
   import controller
   selvar # : traffic
   stvar $ : (STOPPED, GO)
   init STOPPED@
   trans

   STOPPED{ no_cars: car_waiting }
    -> GO: (ROAD: car_waiting)*(controller = go)
    -> $ : else;

   GO    { cars_passing: no_cars }
    -> STOPPED: (ROAD: no_cars) + (controller = stop)
    -> $ : else;

   end ROAD()
```

C.1.2 The Crossroads Design

The whole design consists of the crossroads itself, modeled as a hierarchical process **X** and a controller process **C**. The two subprocesses within the crossroads process **X** are symmetric instantiations of a prototype process **ROAD** which is configured with the appropriate conditions.

```
proc X   /*The Crossroads*/
   import C
   selvar A.#, B.#

   proc A : /*Avenue A*/
      ROAD( go ?(C: go_A) | slow ?(C: slow) | stop )
```

1. This example is taken from the file crd.sr which can be found in the examples directory of the COSPAN distribution. One modification to that example has been done in that the Policeman of that example has been replaced with the Controller of the Crossroads example described in the appendices of [Kur92].

```
      proc B : /*Boulevard B*/
         ROAD( go ?(C: go_B) | slow ?(C: slow) | stop )

      end X

   proc C   /*Controller*/
      import X.A, X.B
      selvar # : (pause, go_A, go_B )
      stvar $ : (go_A, go_B)
      init go_A, go_B
      trans

      go_A  { pause: go_A }
         ->go_B: (C: go_A)*(X.B: car_waiting)
         ->go_B: (C: go_B)*(X.B: car_waiting) + (C: pause)
      go_B  { pause: go_B }
         ->go_A: (C: go_B)*(X.A: car_waiting)
         ->go_B: else

      end C
```

C.1.3 The Monitor

The monitor for the design ensures that progress is always made. The monitor is named **ST** and is an instance of a general monitor prototype called **STARVATION**. The general monitor prototype defines the abstract concept which is then configured for use in design by passing the specific conditions as arguments

```
   proctype STARVATION( _hungry, _fed : boolean )
      import _hungry, _fed
      stvar $ : (1, 0)
      recur *->0
      init 0
      trans
      0     ->1: _hungry
            ->$: else;
      1     ->0: _fed
            ->$: else;
      end STARVATION()

   monitor ST : STARVATION((X.A: car_waiting)*~(P: go_A),
                              (P: go_A))
```

Since both avenues **A** and **B** are symmetrical in this example, it is enough to show only that every car on avenue **A** will eventually get across the street.

C.2 V-VHDL

The V-VHDL description of the crossroads consists of a number of V-VHDL *units*, each of which are declared separately, but are assembled together in Section B.2.5 to form two separate but equivalent designs. There are two design descriptions of the crossroads, a state-based description and a control-based description. Both descriptions are presented here to reinforce the claim that the style of the description is independent of the fundamental structure of the design or of the properties to be proved about it. The independence of the design declaration from the description style is provided by the compilation algorithm which extracts the same state graph out of both types of descriptions. Programmer may thus describe the design or property to be proved in whatever way is most convenient for expression.

The V-VHDL design described in the following sections consists of a number of units which, when compiled, are installed into the library **Work** in the structure shown in Figure 28. It is useful to consider the VHDL defines an abstract filesystem not unlike the Unix filesystem. The abstract filesystem of VHDL, and thus of V-VHDL consists of *libraries* which may contain *packages*, *entities* and *monitors* and *configurations*. Associated with packages and entities are *package bodies* and *architecture bodies* respectively; the association is in much the same spirit as subdirectory containment within the Unix filesystem. Figure 28 illustrates the relationships among the design units in this way.

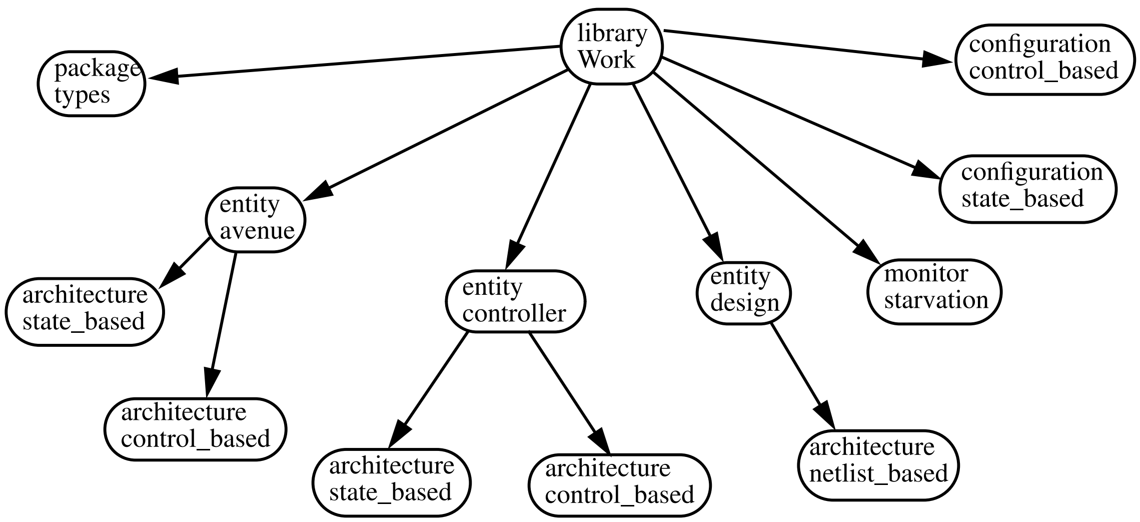


Figure 28 The Crossroads Design Units

The following sections describe each one of the design units to be found in library **Work**, giving the code and describing the main features of each.

C.2.1 Generic Useful Definitions

There is a single package which declares all of the useful types that are available throughout the design. This package is imported to every design entity.

```
package types is
   type avenue_state is (no_cars, cars_waiting, cars_going);
   type controller_state is (pause, go_A, go_B);

   component monitor_template
      port(avenue_A, avenue_B: in avenue_state;
            controller_C: controller_state);
   end component;

   component avenue_template
```

```
        port(the: inout avenue_state;
              controller: in controller_state);
     end component;

     component controller_template
        port(the: inout controller_state;
              avenue_A: in avenue_state;
              avenue_B: in avenue_state);
     end component

  end types;
```

The main feature of this package is that it defines two enumerated types that are used as the underlying type of signals which represent the selection of the relevant processes.

Also defined in the package are the component declarations for the entities of the design. These component declarations are placed here for convenience. Their being available in a package allows them to be used in a wide variety of contexts. It would be equally plausible to declare these components directly in the architecture where they are used: architecture **`netlist_based`** of entity **`crossroads`**.

There is no package body for this package because there are no declarations in the package which need further definition in a package body. For example, there are no function declarations which need definition; all of the declarations in the package types are complete in their present form.

C.2.2 The Entity Declarations

There are three entities to this hierarchical design. The top-level entity is of course the **`crossroads`**. As shown in the following section which describes the architecture bodies, the entity is decomposed into a netlist of three other instances, two instances of **`avenue`** and one instance of **`controller`**. These entity declarations are thus the interface to those other modules of the design.

```
entity crossroads is
   -- No ports or anything
end crossroads;

use Work.types.all;
entity avenue is
   port(the: inout avenue_state;
         controller: in controller_state);
end avenue;

use Work.types.all;
entity controller is
   port(the: inout controller_state;
         avenue_A: in avenue_state;
         avenue_B: in avenue_state);
end controller;
```

A useful point to note here is that the package **`types`** is incorporated into the entity declarations for **`avenue`** and **`controller`** so that the ports can be of the types declared in that package.

C.2.3 The Architecture Bodies

The *entity* defines the external; interface to a design unit; the *architecture* thus defines the internal contents of the design unit. There may be several possible architectures for the same external interface, each of which represents a point in the so-called design space. By convention, architectures are equivalent under the restrictions defined by the external interface.

This section provides two architectures for each of the low-level basic entities **`avenue`** and **`controller`**. There is a state-based architecture which uses the value of a process' variable to denote the state of the processes. There is also a control-based architecture in which the process contains no variable, but rather denotes state by the current execution position in the code body. Both architectures are equivalent for they have the same set of states and the same state-transition behavior.

The Avenue Architectures

```
architecture state_based of avenue is
begin

   process
      type action is (STOP, GO);
      variable a: action := STOP;
   begin
      case a in
         when STOP =>
            the <= nondetermine no_cars, cars_waiting;
            if the'selection = cars_waiting and
                  controller'selection = go_A then
               a := GO;
            end if;
         when GO =>
            the <= nondetermine cars_going, no_cars;
            if the'selection = no_cars or
                  (the'selection = cars_going and
                   controller'selection = go_A) then
               a := STOP;
            end if;
      end case;
      wait on all;
    end process;

end state_based;

architecture control_based of avenue is
begin

   process
   begin
      stopped:
      loop
         the <= nondetermine no_cars, cars_waiting;
         if the'selection = cars_waiting and
               controller'selection = go_A then
```

```
                exit;
            end if;
            L1: wait on all:
        end loop;
        L2: wait on all:

        going:
        loop:
            the <= nondetermine cars_going, no_cars;
            if the'selection = no_cars or
                    (the'selection = cars_going and
                        controller'selection = go_A) then
                exit;
            end if;
            L3: wait on all:
        end loop;
        L4: wait on all;
    end process;

end control_based;

architecture reduced of avenue is
begin

    process
    begin
        the <= nondetermine no_cars, cars_waiting, cars_going;
        L1: wait on all:
    end process;

end reduced;
```

The Controller Architectures

```
architecture state_based of controller is
begin

    process
        type who_can_go is (A_can_go, B_can_go);
        variable w: who_can_go;
    begin
        choice in
            w := A_can_go;
        else
            w := B_can_go;
        end choice;
        loop
            case w in
                when A_Can_go =>
                    the <= nondetermine pause, go_A;
```

```
                choice in
                   if the'selection = go_A then
                      w := B_can_go;
                   endif;
                else
                   if the'selection = go_A or
                         the'selection = pause then
                      w := A_can_go;
                   end if;
                end choice;
             when B_can_go =>
                the <= nondetermine pause, go_B;
                if the'selection = go_B and
                      avenue_A'selection = cars_waiting then
                   w := A_can_go;
                else
                   w := B_can_go;
                end if;
          end case;
          wait on all;
       end loop;
    end process;

end state_based;

architecture control_based of controller is
begin

   process
   begin
      A_may_go:
      loop
         choice in
            the <= pause;
            wait on all;
         else
            exit;
         end choice;
      end loop;
      the <= go_A;
      wait on all;

      B_may_go:
      loop
         choice in
            the <= pause;
            wait on all;
         else
            exit;
         end choice;
```

```
        end loop;
        the <= go_B;
        wait on all;
    end process;

end control_based;
```

The Crossroads Architecture

The architecture of the top-level of the design is relatively simple as all it does is declare the interconnection of the lower-level design entities. For this reason the architecture is named **`netlist_based`** to denote that there is no actual functionality defined here, only the interconnection of previously-declared entities.

```
use Work.types;
architecture netlist_based of crossroads is
    signal avenue_A_selection: types.avenue_state;
    signal avenue_B_selection: types.avenue_state;
    signal controller_C_selection: types.controller_state;
begin

    avenue_A: types.avenue_template
          port map(the => avenue_A_selection,
                   controller => controller_C_selection);

    avenue_B: types.avenue_template
          port map(the => avenue_B_selection,
                   controller => controller_C_selection);

    controller: types.controller_template
          port map(the => controller_C_selection,
                      avenue_A => avenue_A_selection;
                      avenue_b => avenue_B_selection);

end netlist_based;
```

An interesting point in this description is the use of the qualified names throughout the architecture body. For example the types of the signals are all qualified by the name of the package **`types`**. This qualified name syntax is required because the **`use`** declaration at the top of the architecture imported only the name **`types`**, it did not import all of the names *inside* of the package as was the case previously when the statement used was use **`Work.types.all`**.

While this ability to reference through a qualified name is not of fundamental importance for this example *per se,* the use of qualified names is important for other aspects of the V-VHDL language; one of the most significant of these uses is the reference of a variable within a process from a monitor as was shown in Section 2.2.1.

C.2.4 The Monitor

The monitor is the construct that specifies one or more properties that the design is to obey. A monitor is declared in much the same spirit as an entity for it is allowed to have *generic parameters* and *port parameters*. In addition, since a monitor is allowed to reference the internal state variables of the processes that it observes, there is a provision for *state parameters* which bind references in the monitor to the actual variables in the observed process.

The monitor process for the crossroads design does not use all of the possible generality available; it only requires a few port parameters which allow it to import the selections of the processes that it is to observe. Beyond that, the monitor is made up of a single process which observes the values selected and emitted onto the signals, and maintains a small bit of state.

```
use Work.types.all;
monitor cross_street is
   port(avenue_A, avenue_B: in avenue_state;
         controller_C: in controller_state);
begin

   process
      type two_state is ('0', '1');
      variable s: two_state := '0';

      state s0 is at L1 with s = '0';
      state s1 is at L1 with s = '1';

      always_make_progress:
      require
         from s0 to s1;
         from s0 to s0;
      end require
   begin
      case s in
         when '0' =>
            if avenue_A = cars_waiting then
               s := '1';
            else
               s := '0';
            end if;
         when '1' =>
            if avenue_A = no_cars or avenue_A = cars_going then
               s := '0';
            else
               s := '1';
            end if;
      end case;
      L1: wait on all;
   end process;

end cross_street;
```

The process in the body of the monitor maintains two states which are labeled **`'0'`** and **`'1'`** for lack of any other imaginative names. Based on the conditions observed on the selection signals, the process evolves over time between the two states. In the declarative portion of the monitor, two states, **`s0`** and **`s1`** are named and a requirement is declared that there must always be transitions out of state **`s0`**. This requirement is interpreted to mean that progress is always made. Thus the monitor as a whole ensures that the design makes progress, or more plainly, that cars can cross the street.

An important feature to note in the above example is the similarity of the monitor construct to an entity. In fact, this specific monitor could be "converted" to an entity directly simply by changing the leading keyword. Using the VHDL-1076 entity construct, the process in the entity body would then be considered a *passive process* for it performs no signal assignments. Thus this example, the monitor **cross_street**, is a good example of a case where the roots of the monitor construct in the passive process of VHDL-1076 are plain to see.

C.2.5 The Configurations - Assembling The Design

At this point, all of the parts of the design have been declared, and, having been compiled, are ready to be used in the library **Work**. The task now is to assemble the parts into a design hierarchy. This is done through a **configuration** construct which defines the instance hierarchy of the design, and provides an opportunity to insert monitors into the design hierarchy.

The following examples define two different design hierarchies. The first is a design hierarchy that uses the state-based representations of the leaf entities, while the second is a design hierarchy is one that uses the control-based representations of the leaves. Both designs are equivalent by virtue of the fact that the state-based and control-based descriptions of the entities **avenue** and **controller** are equivalent.

Assembling the State-Based Design

The configuration is a unit which defines the binding of each instantiation site in the design. These bindings are accomplished with *for-statements* within the body of the configuration. Each for statement opens up a new scope, and possibly binds the instantiation sites at that level, thereby creating the opportunity for deeper scopes to be opened up.

In the case of this configuration, named **state_based**, the first scope to be opened up within the entity **design** is the scope defined by the architecture **netlist_based**. Within the scope of that architecture, there are three sites for instantiation. The three nested for statements populate the sites with a specific architecture of a given entity. For example, in the case of the **avenue_template**, the first for statement declares that the architecture **state_based** of the entity **avenue** in the library **Work** will be used at the indicated named sites **avenue_A**, **avenue_B**.

```
configuration state_based of design is
   for netlist_based
      for avenue_A, avenue_B: avenue_template
         use entity Work.avenue(state_based);
      end for;

      for controller: controller_template
         use entity Work.controller(state_based);
      end for;

      A_will_eventually_cross_the_street: monitor_template
            port map(avenue_A => avenue_A_selection,
                     avenue_B => avenue_B_selection,
                     controller => controller_C_selection);
      for A_will_eventually_cross_the_street: monitor_template
         use monitor Work.cross_street;
      end for;

   end for;
end state_based;
```

At the bottom of the scope opened by the **for netlist_based** statement is a usage which is unique to V-VHDL: the instantiation of a monitor. This instantiation is interpreted to be the same as if the monitor were instantiated directly in the body of architecture **netlist_based**. The syntax used is exactly the same as that which is used for the instantiation of entities in the design. In fact, directly after the definition of instantiation site, there is a configuration item which defines the binding of that instantiation site to a specific monitor, **cross_street** in the library **Work**.

Assembling the Control-Based Design

The same situation occurs for the control-based design; the configuration simply populates the instantiation sites in the hierarchy with the **control_based** architecture.

```
configuration control_based of design is
   for netlist_based
      for avenue_A, avenue_B: avenue_template
         use entity Work.avenue(control_based);
      end for;

      for controller: controller_template
         use entity Work.controller(control_based);
      end for;

      A_will_eventually_cross_the_street: monitor_template
            port map(avenue_A => avenue_A_selection,
                        avenue_B => avenue_B_selection,
                        controller => controller_C_selection);
      for A_will_eventually_cross_the_street: monitor_template
         use monitor Work.cross_street;
      end for;


   end for;
end control_based;
```

It should be noted that this second configuration is a special case of a more general phenomenon. In this example, the instance hierarchy between the state-based and the control-based configurations are exactly the same. In general however, the instance hierarchies of the two configurations will not be the same and so there will be different opportunities for verification between the two designs.