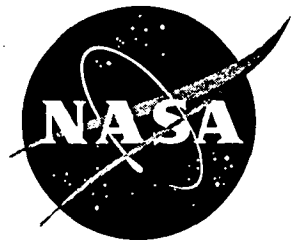
NASA Contractor Report 191577



NASA-CR-191577
19940028228

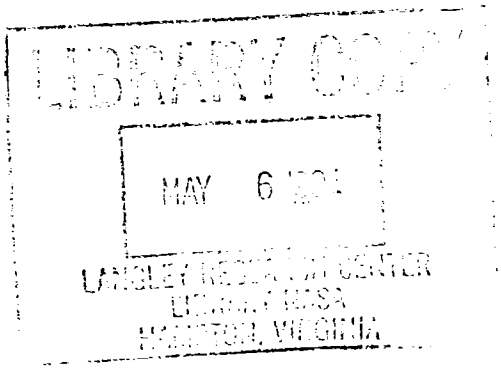
FORMAL SEMANTICS FOR A SUBSET OF VHDL AND ITS USE IN ANALYSIS OF THE FTPP SCOREBOARD CIRCUIT

Mark Bickford
Odyssey Research Associates, Ithaca, NY

Contract NAS1-18972

April, 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, VA 23681-0001



Formal Semantics For a Subset of VHDL and its use in analysis of the FTPP Scoreboard circuit [1]

Mark Bickford

ORA Inc.,
301A Harris B. Dates Drive
Ithaca, NY 14850.

April 5, 1994

[1] Prepared for Langley Research Center under Contract NAS1-18972.

Abstract

In the first part of the report, we give a detailed description of an operational semantics for a large subset of VHDL, the VHSIC Hardware Description Language. The semantics is written in the functional language Caliban, similar to Haskell, used by the theorem prover Clio. We also describe a translator from VHDL into Caliban semantics and give some examples of its use. In the second part of the report, we describe our experience in using the VHDL semantics to try to verify a large VHDL design. We were not able to complete the verification due to certain complexities of VHDL which we discuss. We propose a VHDL verification method that addresses the problems we encountered but which builds on the operational semantics described in the first part of the report.

Contents

List of Figures

1 Introduction

Formal verification is a method of validating computer designs and programs by applying symbolic logic techniques. Hardware verification is formal verification applied to the validation of hardware designs. To formally verify a design, one specifies the design and the requirements that the design is expected to satisfy in a formal logical notation. Then, one constructs a formal proof that the design meets the stated requirements. To construct a proof, one needs to use a system for symbolic manipulation, such as a theorem prover, that manipulates expressions belonging to the logic used in the specifications.

The advantage of formal verification over traditional validation methods, such as simulation and testing, is that formal verification is equivalent to total test coverage for the verified property. When one constructs a formal proof of a property, the property is shown to hold for all permissible inputs and initial conditions of a design. Of course, there is no guarantee that the requirements specification with respect to which the design is verified is what one wants. But, formal verification brings potential errors in a design into sharper focus by subjecting the design to more intense scrutiny than is possible in simulation. The process of formal verification is, in general, labor intensive. But, there are certain application domains, such as digital hardware, where recent experience [9, 6, 5] suggests that the technology is applicable to industry-scale designs, and certain application areas, such as safety-critical and mission-critical systems, where the advantages of formal verification outweigh its cost.

NASA Langley Research Center has recently initiated a concerted effort [4] involving several organizations, including ORA Corporation, to study the use of formal verification as a possible validation technology for fault-tolerant digital flight-control systems in an application area that requires ultra-high reliability and availability. The first step in this effort was concerned with demonstrating the application of formal verification to key design problems in this area. Some of the significant case studies that were completed as part of the first step are formal verification of a clock synchronization algorithm framework [8], an interactive consistency algorithm [1], and a Byzantine-resilient microprocessor system [11].

One of the goals of the second step of the NASA effort is to explore the integration of formal methods into the design and verification of key components of current fault-tolerant architectures being built in the aerospace industry. Toward this goal, ORA is currently teamed with Charles Stark Draper Laboratory (CSDL), which is one of the organizations in the forefront of building fault-tolerant systems for safety-critical applications. As part of a NASA/Army-sponsored project, called Army Fault-Tolerant Architecture (AFTA), CSDL is developing the Fault-Tolerant Parallel Processor (FTPP). One of ORA's current tasks is to formally specify and verify a key component, called Scoreboard, of FTPP. The Scoreboard is part of the FTPP virtual bus that guarantees reliable communication between processors in the presence of physical faults in the sys-

tem. The virtual bus design is based on a conservative fault model, called the *Byzantine* fault model, in which a faulty component can exhibit arbitrary and malicious behavior. The Scoreboard implements a piece of control logic that approves and validates a message before it can be transmitted. The verification of the Scoreboard is being performed in phases.

Since the detailed design of the Scoreboard was still evolving at the time we started Phase 1, our main objective then was to lay the foundation for the verification effort. We did this by formally specifying and verifying a simplified Scoreboard design as reported in [12]. The Scoreboard was subsequently fully designed and built (at Draper Labs) using VHDL to specify the design. VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, which is an IEEE standard and is mandated for application-specific hardware developed for the USAF. We were to continue the formal verification of the Scoreboard, this time verifying the actual design built at Draper Labs. Because the design was written in VHDL we focused our effort in Phase 2 on the formal verification of VHDL designs.

1.1 Formal verification of VHDL

In order to integrate formal methods into the design process, a common language for designs that can be the interface to formal verification tools as well as to CAD tools such as synthesis tools, is of the utmost importance. While, from the formal methods perspective, VHDL may not be the ideal language in which to describe designs, because of its complexity, it is nevertheless becoming a standard and several other formal methods practitioners are working with VHDL [13, 14]. Therefore, we decided to develop a general method for verifying designs expressed in VHDL and apply the method to the Scoreboard design. Since a version of the Scoreboard design had already been verified, the general methods we developed for VHDL and our experience in applying them to the Scoreboard, are of more importance than the fact that this particular VHDL design is correct.

Any formal, theorem-prover based, verification method for a language such as VHDL involves the definition of a semantics for the modeled language in the language of the theorem prover (Clio [2] in our case). Requirements are then written directly in the language of the prover, or a suitable requirements language can be interpreted into the language of the prover. We call the statement that a particular design entity meets a particular requirement a *judgement*. Then any judgement becomes an assertion, in the language of the prover, that the semantics of the entity satisfies the given requirement. The developer of a verification method must therefore choose the kind of formal semantics with which to model the design language, the kind of requirements that will supported, and then build up enough of the theory of the semantic domain to support the proofs of the judgements.

In choosing the kind of formal semantics with which to model the entities,

one of main choices we had to make was between an *operational* semantics and *logic-based* semantics. In the operational approach, the meaning of a entity is given by its behavior, and its behavior is given by defining appropriate notions of the *state* and the *state transition function* (or *state transition relation* if the semantics is not deterministic). A *run* is then defined to be a sequence of states, where each state is related to the previous state by the transition function (or relation), and requirements can be stated as assertions about runs. A judgement is then the assertion that any run of the given entity satisfies the given requirement. In the logic-based semantics, a set of axioms and inference rules for judgements is given directly. Then the meaning of a particular entity is just the set of judgments about it which can be proved using the given rules.

We chose to write an operational semantics for VHDL for several reasons. First, since VHDL is a deterministic language, and the language of our prover, Clio, is based on functional programming, it was very natural to write a semantics using a transition function. Second, in order to use the logic-based approach, the set of judgements for which proof rules must be given must be known. This means that the requirements language must be explicitly given. The VHDL designs consist of sets of concurrent processes, and creating a language that can express all the requirements on such sets that we might want to verify is an active research area. In the operational approach, the runs of a design can be defined within the system, and any predicate on runs that can be stated in the language of the prover can be used as a requirement. To be sure, stating requirements in this way requires more expertise, and therefore we will want to create a higher level requirements language that can be interpreted into these predicates on runs, but this task can be done after we have experience in using the semantics, so that we know what kinds of requirements we want to write. Third, once an operational semantics is written, we can still use the logic-based approach by proving the desired inference rules as "meta-theorems" about the semantics.

Once the semantic domain has been defined, we may assign meanings to each language construct. There then arises the question of whether these meaning assignment functions should themselves be defined within the logic of the theorem-prover or be encoded, external to the theorem-prover, into a tool which interprets the design language (VHDL) into its formal semantics. The terms *deep embedding* and *shallow embedding* are introduced in the paper by Gordon et.al. [3] to refer to these two alternatives. In order to define the meaning functions in the logic of the theorem-prover, the abstract syntax of the design language must be embedded as an abstract data type in the logic of the prover. Then the meaning functions can be defined as functions from this data type into the semantic domain. This approach is known as making a *deep embedding*, and has the advantage that "meta-theorems" about the meaning functions can be proved within the system, since the meaning functions themselves are defined in the system. The second approach, writing an interpreter from the design language into its formal semantics, is known as making a *shallow embedding*.

It has the advantage that it can be implemented efficiently using tools such as yacc, and it avoids the overhead of defining the syntax of the design language within the formal logic. Using the shallow embedding, a judgement about particular design is translated into an assertion in the logic of the prover. We may not directly state meta-theorems about all judgements, however rewrite-rules about the semantic functions can be proved which can have the same effect. Since our effort in this project was to be modest (less than one staff-year), and since we planned to verify a particular design (the Scoreboard), we chose to make a shallow embedding, which we implemented using yacc. However, as the work progressed, we found it desirable to add some of the features of a deep embedding to our model. Thus the final semantics may be seen as something intermediate between a shallow and a deep embedding.

In the first part of this report (sections 2– 4), we describe the semantic domains for the operational semantics of VHDL as defined in the language (Caliban) of our prover, Clio. We also describe the translation process from VHDL texts into this semantics and initial results using the semantics on simple examples.

In the second part of the report (sections 5– 8) we describe our experience in applying this method to the Scoreboard design. Here we cannot claim to have been successful. Our original plan was to use our VHDL semantics to map the behavior of the Scoreboard design onto the more abstract behavior that was used to verify the simplified Scoreboard design in phase I of our work. We hoped that this step could be made semi-automatic, so that the behavior of any design in the synchronous, clocked style of the Scoreboard design could be mapped automatically to a more abstract behavior at the level of our previous, successful verification of the Scoreboard. This abstraction would amount to replacing VHDL's complex notion of explicit time, with a more abstract notion of consecutive clock cycles. Although the initial results seemed promising, we were unable to get the desired semi-automatic mapping to work on the Scoreboard design. We therefore give an analysis of why our approach was not successful. In particular we pinpoint those parts of the VHDL operational model that cause the level of complexity that we were unable to overcome. In the light of this analysis, we can formulate a method of reasoning about VHDL designs that we feel would have succeeded in verifying the Scoreboard, but which we could not implement in this phase of the project.

The proposed solution to the problems we encountered involves moving from the operational approach to a more logic-based approach, which, in hindsight, seems to invalidate the choices we made in creating our VHDL semantics. However, the current semantics will provide the basis for the proposed future semantics, and thus much of the current effort can be reused. Also, without the experience gained in this effort, the essential problems that must be addressed would not have been uncovered. Our approach to VHDL modeling was a natural one that would likely be followed by other formal methods practitioners. Thus, the pitfalls we encountered and our proposed solutions should be important to

other efforts of this kind as well as to our own future work.

2 The Semantics

In this section we describe in detail the semantic domains and associated definitions for our operational semantics of VHDL. We give this amount of detail (and full annotations in English) because we hope that other formal methods practitioners who would like to try this method or a similar one will be able to interpret our definitions into the logic of their system easily. We assume that the reader has some familiarity with VHDL, but we motivate our semantics by giving examples from VHDL and informal descriptions of VHDL concepts so that a reader with limited familiarity with VHDL should be able to follow the discussion. We also assume some familiarity with functional programming languages since our definitions are written in Caliban which is a language similar to Miranda or Haskell. Here again, we give some hints to help the reader decipher Caliban expressions which may be unfamiliar.

2.1 Simulation

We start with the top level module of definitions, in which **STATE** and **Simulate** are defined, in order to give the basic structure of our semantics. In the following sections, we work in a bottom-up way through the supporting definitions. After reading those sections, the reader may want to reread this section.

To give an operational semantics we must define appropriate notions of *STATE* and *state transition function*. The semantics of VHDL is based on the idea of *simulation*. The simulation advances through a sequence of states called *simulation cycles*. Thus if we take the state transition function to be the function, *Simulate*, that gives the state of the next simulation cycle as a function of the current state, then a simulation is the same as our notion of a *run*.

The STATE consists of three things:

1. An assignment of values to the set of signals and variables.
2. A set of waiting processes.
3. A list of times at which events are scheduled.

In Caliban:

```
type STATE = <<SIGSTATE,[PROCESS],[TIME]>>
```

The values assigned to signals are complicated because in VHDL we may assign waveforms to signals, so the value of a signal must contain information about future times, and we may ask how long a signal has been stable, so the value must also contain information about past times. We will give the details of the SIGSTATE later, but the basic form is:

```
type SIGSTATE  = NAME->NAMEVAL
```

Here NAME is a type that includes the atomic signals and variables as well as aggregates. The type NAMEVAL includes simple values as well as the complex signal values. We record the changes to a SIGSTATE as a list of *updates* or *transactions*. Each update changes one NAME,NAMEVAL pair in the SIGSTATE. We call a single such update a SIGSTATE-ITEM.

Now, a VHDL simulation cycle proceeds as follows. The *global clock* is advanced to the next time at which an event is scheduled (this can be an increment of zero), then all waiting processes are checked to see whether they will be active in this cycle. Each process that becomes active will execute until it suspends. The result of executing a process is a triple containing a list of updates (transactions) to the SIGSTATE, a new waiting process (its continuation), and a list of times at which the new transactions are scheduled. If the process does not become active then its result will contain no transactions and its continuation will be itself. The results of all the processes are combined into a triple containing the list of all new transactions, all the new waiting processes, and all the new transaction times. Then the new state is obtained by updating the old signal state with the new transactions, replacing the old set of waiting process by the new set, and merging the new transaction times with the old.

To read the following Caliban definitions, note that double angle brackets construct tuples, [] is the empty list, ++ is the list append operator, the definitions of FOLDR, MAP, and MERGE are standard (see section 2.14), and that `__update` is a built-in operation that updates a function with a list of changes.

```
Simulate :: STATE->STATE
Simulate S = ONE_CYCLE (START_CYCLE S)

ONE_CYCLE <<S,Q,T>> = FINISH S T (DO_PROCS S Q)

FINISH S T <<SI,PS,TS>> = <<__update S SI, PS, MERGE TS T>>

type RESULT = <<[SIGSTATE_ITEM],[PROCESS],[TIME]>>
DO_PROCS :: SIGSTATE->[PROCESS]->RESULT
DO_PROCS S Q = ZIP3 (MAP (DO_PROC S) Q)

ZIP3 = FOLDR APPEND3 <<[],[],[]>>
APPEND3 <<A1,A2,A3>> <<B1,B2,B3>> = <<A1++B1,A2++B2,MERGE A3 B3>>
```

We start the cycle by advancing the clock to the next time on the global list and decrementing the remainder of the list.

```
START_CYCLE <<S,Q,[]>> = <<S, [] , []>>
START_CYCLE <<S,Q,(T:REST)>> = <<ADVANCE T S, Q ,DECR T REST>>

DECR T = MAP (\X-> X #- T)
```

2.2 Names and Expressions

The SIGSTATE is a function from NAMEs to NAMEVALs. Abstractly, the only names that must be assigned values by the SIGSTATE are the atomic signals and variables. However, since the grammar for VHDL includes such things as function and array applications, record selections, ports and generics as names, the translation is smoother if we include these in semantic domain of NAMEs. This is an example of where our embedding, by including some of the abstract syntax of VHDL, is intermediate between a shallow and a deep embedding.

Now in the VHDL for the Scoreboard we encounter:

```
type vlbit_1d is array(Natural range <>) of vlbit;
architecture voter of voter is
  ...
signal   cmem0:        vlbit_1d(7 downto 0);
  ...
end voter;
```

This declares a SIGNALNAME "cmem0" of an array type indexed by 7 ...0. Since there may be more than one instance of the entity "voter" in a given design, every signal name has an instance name as well as a local name. The instance name is a list of strings such as

```
["board1","voter1"]
```

This identifies the instance "voter1" inside the instance "board1". Every signal is uniquely identified by its instance name, local name, and type. The first two productions in the following Caliban data type create names for all the signals and variables in the state. Note that, in Caliban, strings are lists of characters, hence `[CHAR]`. The other names are used for record and array selections and for other kinds of names which do not need to be declared, and also a special name to hold the value of the global clock.

```
NAME ::= SIGNALNAME [[CHAR]] [CHAR] TYPE
         | VARIABLENAME [[CHAR]] [CHAR] TYPE
         | FIELDNAME [CHAR] TYPE
         | PORTNAME [CHAR]
         | GENNAME [CHAR]
         | SELECT NAME NAME
         | INDEX NAME NAT
         | FUNCAPPLY FUNCTION EXPRESSION
         | ARRAYAPPLY NAME EXPRESSION
         | CONSTANT VALUE
         | ATTRIBUTEOF [CHAR] NAME
         | GLOBALCLOCK
         | FUNCRETURN TYPE
```

```
        | LOOPINDEX [CHAR]
        | ILLEGAL [CHAR]

FUNCTION ::= FUNCNAME [CHAR]
```

The TYPE of a name merely distinguishes atomic names from names of aggregates, and in the case of aggregates, allows us to find the set of atomic names which make up the aggregate.

```
TYPE ::= RECORD [NAME]
        | ARRAY [NAT] TYPE
        | ARRAY_UNCONSTRAINED TYPE
        | ATOMIC

ADD_CONSTRAINT L (ARRAY_UNCONSTRAINED T) = ARRAY L T

NAME_KIND ::=  SIG_KIND | VAR_KIND
```

We also introduce here the type of expressions. They are built up from names and constants by applying the predefined unary and binary operators and by the formation of aggregates.

```
EXPRESSION ::= BINARY EXPRESSION OPERATOR EXPRESSION
             | UNARY UNARYOP EXPRESSION
             | AGGREGATEEXP [EXPRESSION]
             | EXPNAME NAME
             | EXPCONSTANT VALUE

OPERATOR ::= ANDOP | OROP | XOROP | NOROP | NANDOP
           | EQUALOP | NOTEQUALOP
           | LESSOP | LEQOP | GREATEROP | GEQOP
           | PLUSOP | MINUSOP | AMPEROP
           | TIMESOP | DIVIDEOP | MODOP | REMOP

UNARYOP ::= NOTOP | ABSOP | UPLUSOP | UMINUSOP
```

In VHDL, we may assign a waveform to a signal. A waveform is a list of time-value pairs, so an expression for a waveform is, abstractly, a list of time-expression,value-expressions pairs. Also waveform expressions may be conditional as in:

```
x <= y after 1ns  when z=1 else u after 5ns;
```

Therefore we define:

```
WAVEFORM_EXP ::= WFEXP [<<EXPRESSION,EXPRESSION>>]
               | CONDWF EXPRESSION WAVEFORM_EXP WAVEFORM_EXP
```

2.3 Simple Values

The signals have a complex value that includes information about past and future times, but variables have simple values. The union type VALUE includes all the data types defined by the user but also starts with booleans, bits, and natural numbers. There is a special UNINITIALIZED value and lists of values may be grouped into aggregate values.

```
VALUE ::= UNINITIALIZED
        | boolVAL BOOL
        | bitVAL bit
        | natVAL NAT
        | AGGREGATE [VALUE]
        |+
```

In Caliban, the notation:

```
VALUE ::= |+
```

means that more productions can be added in later declarations. We use this feature to allow our translator to add more kinds of values to the type VALUE when type declarations are seen.

The type, `bit`, is modeled as a seven value lattice because that is the bit type used by the Synopsis synthesis tools that were used by Draper labs. Our current translator does not allow us to redeclare data types in VHDL. If it did, then the bit type could start with just 0 and 1, and then the Synopsis bit package could redefine bit as the seven value type.

```
bit ::= bit0 | bit1 | bitX | bitZ | bitW | bitL | bitH
```

We have not defined the operations on bits. The following axioms imply that however bitAND,bitOR,etc are defined on bits, they must be definiteness preserving. (In Clio, the double exclamation mark means "is completely defined" which technically means that no constructor is applied to "bottom".)

```
bitAND,bitOR,bitNAND,bitNOR,bitXOR :: bit->bit->bit
AXIOM (X)(Y) '!!(bitAND X Y)'='True', '!!X & !!Y' = 'True'
AXIOM (X)(Y) '!!(bitOR X Y)'='True', '!!X & !!Y' = 'True'
AXIOM (X)(Y) '!!(bitXOR X Y)'='True', '!!X & !!Y' = 'True'
AXIOM (X)(Y) '!!(bitNOR X Y)'='True', '!!X & !!Y' = 'True'
AXIOM (X)(Y) '!!(bitNAND X Y)'='True', '!!X & !!Y' = 'True'
```

In VHDL, the bits are named by characters.

```
CHARTOBIT '0' = bitVAL bit0
CHARTOBIT '1' = bitVAL bit1
CHARTOBIT 'X' = bitVAL bitX
CHARTOBIT 'Z' = bitVAL bitZ
CHARTOBIT 'W' = bitVAL bitW
CHARTOBIT 'L' = bitVAL bitL
CHARTOBIT 'H' = bitVAL bitH
```

The operations AND, OR, XOR, NOR, NAND, and NOT are overloaded on bits, bools, and aggregates. Any other use would be a type error so we give arbitrary well defined default values to those cases.

```
AND (boolVAL X)(boolVAL Y) = boolVAL(X & Y)
AND (bitVAL X)(bitVAL Y) = bitVAL(bitAND X Y)
AND (AGGREGATE X)(AGGREGATE Y) = AGGREGATE(MAPOP AND X Y)
AND X Y = boolVAL(False)

OR (boolVAL X)(boolVAL Y) = boolVAL(X | Y)
OR (bitVAL X)(bitVAL Y) = bitVAL(bitOR X Y)
OR (AGGREGATE X)(AGGREGATE Y) = AGGREGATE(MAPOP OR X Y)
OR X Y = boolVAL(False)

XOR (boolVAL X)(boolVAL Y) = boolVAL(X xor Y)
XOR (bitVAL X)(bitVAL Y) = bitVAL(bitXOR X Y)
XOR (AGGREGATE X)(AGGREGATE Y) = AGGREGATE(MAPOP XOR X Y)
XOR X Y = boolVAL(False)

NOR (boolVAL X)(boolVAL Y) = boolVAL(~(X | Y))
NOR (bitVAL X)(bitVAL Y) = bitVAL(bitNOR X Y)
NOR (AGGREGATE X)(AGGREGATE Y) = AGGREGATE(MAPOP NOR X Y)
NOR X Y = boolVAL(False)

NAND (boolVAL X)(boolVAL Y) = boolVAL(~(X & Y))
NAND (bitVAL X)(bitVAL Y) = bitVAL(bitNAND X Y)
NAND (AGGREGATE X)(AGGREGATE Y) = AGGREGATE(MAPOP NAND X Y)
NAND X Y = boolVAL(False)


NOT (boolVAL X) = boolVAL(~X)
NOT (bitVAL bit0) = bitVAL bit1
NOT (bitVAL bit1) = bitVAL bit0
NOT (AGGREGATE X) = AGGREGATE (MAP NOT X)
NOT X = boolVAL(False)
```

Since Clio has a built-in type NAT of natural numbers, we have used them to model the integers in VHDL. These definitions should be changed to use

INTEGER instead of NAT, but we haven't needed to make the change since the Scoreboard design does not use negative integers.

```
PLUS (natVAL X)(natVAL Y) = natVAL(X #+ Y)
PLUS X Y = natVAL(#0)

MINUS (natVAL X)(natVAL Y) = natVAL(X #- Y)
MINUS X Y = natVAL(#0)

TIMES (natVAL X)(natVAL Y) = natVAL(X #* Y)
TIMES X Y = natVAL(#0)

DIVIDE (natVAL X)(natVAL Zero) = natVAL(Zero)
DIVIDE (natVAL X)(natVAL Y) = natVAL(X #/ Y)
DIVIDE X Y = natVAL(#0)

MOD (natVAL X)(natVAL Y) = natVAL(X #% Y)
MOD X Y = natVAL(#0)

REM (natVAL X)(natVAL Zero) = natVAL(Zero)
REM (natVAL X)(natVAL Y) = natVAL(X #/ Y)
REM X Y = natVAL(#0)
```

To define UMINUS and ABS correctly, we need INTEGER rather than NAT.

```
UMINUS X = natVAL(#0)
ABS X = natVAL(#0)

EQUAL X Y = boolVAL (X = Y)

LESS (natVAL X) (natVAL Y) = boolVAL (X < Y)
LESS X Y = boolVAL(False)

LEQ (natVAL X) (natVAL Y) = boolVAL (X <= Y)
LEQ X Y = boolVAL(False)

GEQ (natVAL X) (natVAL Y) = boolVAL (X >= Y)
GEQ X Y = boolVAL(False)

GREATER (natVAL X) (natVAL Y) = boolVAL (X > Y)
GREATER X Y = boolVAL(False)

CONCAT (AGGREGATE X) (AGGREGATE Y) = AGGREGATE (X ++ Y)
CONCAT  X (AGGREGATE Y) = AGGREGATE ([X] ++ Y)
CONCAT  (AGGREGATE X) Y = AGGREGATE (X ++ [Y])
```

```
CONCAT  X Y = AGGREGATE [X ,Y]

KTRUE X = True
```

2.4 Waveforms and Drivers

As mentioned earlier, a waveform is a list of time-value pairs. The time components of a waveform will always be in increasing order. A single time-value pair is a WAVEFORM-ELEMENT and we may increment or decrement its time component. What we are calling a WAVEFORM-ELEMENT is sometimes called a *transaction* in VHDL terminology, but we are using the word *transaction* for different notion.

```
type WAVEFORM = [WAVEFORM_ELEMENT]
type TIME = NAT
type WAVEFORM_ELEMENT = <<TIME,VALUE>>

TIMEOF <<t,v>> = t
WE_PLUS T <<TM,VAL>> = <<TM #+ T, VAL>>
WE_MINUS T <<TM,VAL>> = <<TM #- T, VAL>>
```

The complex signal value includes information about future times. This is because a signal assignment statement assigns a waveform to a signal. Also, in a VHDL design several entities may assign waveforms to the same signal, for example, several entities may write to a bus. The entity that is assigning to the signal will be identified by its INSTANCE-NAME. Therefore we define a DRIVER to be essentially a record containing an INSTANCE-NAME, a WAVEFORM, and for convenience, a current VALUE.

```
DRIVER ::= DR INSTANCE_NAME VALUE WAVEFORM
WAVEFORMOF (DR i v w) = w
DRIVERVAL (DR i v w) = v
```

Now, when we advance the global clock at the beginning of a simulation cycle, the time components of all waveforms in the state are decremented. When the time component of a WAVEFORM-ELEMENT becomes Zero, then that driver is active, and the value component of the WAVEFORM-ELEMENT becomes the value of the driver. A list of drivers is active if one or more of the drivers on the list is active.

```
DVR_MINUS T (DR ID V WF) = DR ID V (MAP (WE_MINUS T) WF)

ADVANCE_DRIVER (DR i v (<<Zero,w>>:wf)) = DR i w wf
ADVANCE_DRIVER dvr = dvr

DRIVER_ACTIVE (DR i v (<<Zero,w>>:wf)) = True
```

```
DRIVER_ACTIVE dvr = False

DRIVERS_ACTIVE [] = False
DRIVERS_ACTIVE (dvr:more) =
          (DRIVER_ACTIVE dvr) | (DRIVERS_ACTIVE more)
```

When several entity instances are driving a signal, then the signal value will contain several drivers, one for each instance. In that case the current value of the signal will be computed using a *resolution function* which we will discuss later. When, on the other hand, the same entity instance makes more than one assignment to a signal, the later assignments may "cancel" some of the WAVEFORM-ELEMENTs of the earlier assignments. Which of them get cancelled depends on whether the signal assignment has *inertial* or *transport* delay.

The function MERGE-DRIVER is used to provide a meaning to a signal assignment. It takes the DELAY-KIND, the current list of drivers, and the additional driver being assigned, and it returns the resulting list of drivers. The instance name of the additional driver is compared with that of each driver on the current list. If it differs from all of them, then the additional driver is merely appended to the list. If it is the same as one of the current drivers, then the additional waveform is combined with the current waveform by ADD-WAVEFORM which implements the rules for "cancelling" WAVEFORM-ELEMENTs (transactions) according to the kind of delay. Note that the current value, v2, of the driver being merged is always ignored.

```
MERGE_DRIVER :: DELAY_KIND->[DRIVER]->DRIVER->[DRIVER]
DELAY_KIND ::= INERTIAL | TRANSPORT

MERGE_DRIVER knd [] dvr = [dvr]
MERGE_DRIVER knd ((DR i1 v1 wf1):more) (DR i1 v2 wf2) =
     (DR i1 v1 (ADD_WAVEFORM knd wf1 wf2)):more
MERGE_DRIVER knd ((DR i1 v1 wf1):more) (DR i2 v2 wf2) =
     (DR i1 v1 wf1):(MERGE_DRIVER knd more (DR i2 v2 wf2))
```

To add the new waveform, wf2, to the existing waveform, wf, we consider the time, t, and value, v, of the first element in wf2, and the delay kind. We first create waveform wf1 by "cancelling" all elements of wf that have times greater than or equal to t. Since the elements of the waveform are kept in increasing order of time, this amounts to removing the tail of the list, wf, starting at the first element with time component greater than or equal to t. If the delay is transport then we merely append the new waveform to wf1. If the delay is inertial then we must also cancel any element of wf1 with a value that differs from v, or which has a time earlier than some element that is cancelled because of this.

```
ADD_WAVEFORM :: DELAY_KIND->WAVEFORM->WAVEFORM->WAVEFORM
```

```
ADD_WAVEFORM knd wf [] = wf
ADD_WAVEFORM TRANSPORT wf (<<t,v>>:wf2) =
    (CANCEL t wf) ++ (<<t,v>>:wf2)
ADD_WAVEFORM INERTIAL wf (<<t,v>>:wf2) =
    (INERTIAL_CANCEL v (CANCEL t wf)) ++ (<<t,v>>:wf2)

CANCEL t [] = []
CANCEL t (<<t2,v2>>:wf) = (t2 >= t)->[]; <<t2,v2>> : CANCEL t wf

INERTIAL_CANCEL v [] = []
INERTIAL_CANCEL v (<<t,v>>:rest) =
    ((INERTIAL_CANCEL v rest) = rest)-> <<t,v>>:rest;
            INERTIAL_CANCEL v rest
INERTIAL_CANCEL v (<<t,w>>:rest) = INERTIAL_CANCEL v rest
```

2.5 Functions on Names

The NAMEs can be partioned into two classes, those NAMEs that are signals will have values that have drivers. The names that are variables have plain values.

```
KINDOF (SIGNALNAME X Y Z) = SIG_KIND
KINDOF (VARIABLENAME  X Y Z) = VAR_KIND
KINDOF (SELECT N1 N2) = KINDOF N1
KINDOF (INDEX N1 N2) = KINDOF N1
KINDOF (ARRAYAPPLY N1 E) = KINDOF N1
KINDOF GLOBALCLOCK = SIG_KIND
KINDOF N = VAR_KIND
```

Given a NAME we can find its TYPE.

```
TYPEOF (CONSTANT V) = ATOMIC
TYPEOF (ATTRIBUTEOF A N) = ATOMIC
TYPEOF (SIGNALNAME INST NM T) = T
TYPEOF (VARIABLENAME INST NM T) = T
TYPEOF (FIELDNAME id T) = T
TYPEOF (SELECT N1 N2) = TYPEOF N2
TYPEOF (INDEX N1 N2) = ARRAYTYPEOF (TYPEOF N1)
TYPEOF GLOBALCLOCK = ATOMIC
TYPEOF (FUNCRETURN T) = T
TYPEOF (LOOPINDEX N) = ATOMIC
ARRAYTYPEOF (ARRAY L T) = T
```

Given a NAME and its TYPE, we can form the list of SUBNAMES whose values form the aggregate value.

```
SUBNAMES N ATOMIC = [N]
SUBNAMES N (RECORD L) = MAP (SELECT N) L
SUBNAMES N (ARRAY L T) = MAP (INDEX N) L
subnames N (ARRAY_UNCONSTRAINED T) = bottom
```

Given a NAME we can find its INSTANCE-NAME

```
INSTANCE_NAMEOF (SIGNALNAME INST NM T) = INST
INSTANCE_NAMEOF (VARIABLENAME INST NM T) = INST
INSTANCE_NAMEOF (SELECT N1 N2) = INSTANCE_NAMEOF N1
INSTANCE_NAMEOF (INDEX N1 N2) = INSTANCE_NAMEOF N1
INSTANCE_NAMEOF (ARRAYAPPLY N1 E) = INSTANCE_NAMEOF N1
INSTANCE_NAMEOF (ATTRIBUTEOF A N1) = INSTANCE_NAMEOF N1
INSTANCE_NAMEOF N = []
```

2.6 Instances

The following statement creates an instance of the entity "voter" with the generic "active" mapped to bit '1' and the ports "p1" and "p2" mapped to names "x" and "y". The label "A" will be used to give signals and variable in this instance unique identifiers.

```
A: voter generic map ( active => '1')
         port map (p1 => x, p2 => y )
         ;
```

In our semantics, the INSTANCE records the information contained in a component instantiation statement such as the above. It contains enough information so that the meaning of an arbitrary name can be resolved. For example, in the above example, suppose that the name "x" had been declared a port of entity "scoreboard" and that the component instantiation statement labeled "A" is part of the architecture of "scoreboard". Suppose the top level "testbed" entity contained the following signal declaration and component instantiation statement:

```
signal clock : bit;
B: scoreboard port map (x => clock) ;
```

Then when evaluating an expression in the body of "voter" that contains the name "p1", we must resolve the name "p1" to the signal to which it is mapped. We give the Caliban definition of the type INSTANCE, and then continue the example.

```
INSTANCE ::= TOP | Inst [CHAR] PORTMAP GENMAP INSTANCE
type PORTMAP = [<<[CHAR],NAME>>]
type GENMAP = [<<[CHAR],VALUE>>]
```

Now, in our example the "testbed" instance is TOP. The instance for the component instantiation statement labeled "B" is:

```
I1 = (Inst "B" [<<"x",clock>>] [] TOP)
      where clock = (SIGNALNAME [""] "clock" ATOMIC)
            x = PORTNAME "x"
```

and the instance for the component instantiation statement labeled "A" is:

```
I2 = (Inst "A" [<<"p1",x>>,<<"p2",y>>] [<<"active",'1'>>] I1),
```

Each instance contains its label, its port and generic maps and it parent instance. Now, given instance `I2` and the portname "p1", we resolve the name using the portmap, and find the portname, `x`. This is resolved, recursively, using the parent instance, `I1` to the signal, `clock`.

Given an instance we can pull out the list of names which uniquely identify the instance.

```
type INSTANCE_NAME = [[CHAR]]
INST_NAME :: INSTANCE -> INSTANCE_NAME
INST_NAME TOP = []
INST_NAME (Inst n p g i) = (INST_NAME i)++[n]
```

For example `INST_NAME I2 = ["B","A"]`.

When a generic is declared it can be assigned a default value. The translator creates from these declarations the definition of the function `DEFAULT-VALUE`. Then by updating the default function with the generic map from an instance, we may look up the current value of a generic.

```
DEFAULT_VALUE :: [CHAR]->VALUE
GEN_LOOKUP :: GENMAP->[CHAR]->VALUE
GEN_LOOKUP L X = _update DEFAULT_VALUE L X
```

2.7 Signal Values

As mentioned earlier, there are two kinds of values that a name can have. The names that refer to variables have simple values. The names that refer to signals have complex values containing information about the past and future times. The complex `SigVal` contains:

1. the time since last active
2. the time since last event
3. the current value
4. a list of drivers

The "time since last active" is updated whenever there is any assignment to the name, even when the current value does not change. The "time since last event" is updated only when the value of the name changes. Using this information about the past, we can evaluate the attributes **x'stable, x'event, x'active** of a name **x**.

```
NAMEVAL ::= SigVal TIME TIME VALUE [DRIVER]
          | VarVal VALUE

VALOF (SigVal t1 t2 v wf) = v
VALOF (VarVal v ) = v
EVENTOF (SigVal t1 t2 v wf) = boolVAL (t2 = Zero)
EVENTOF (VarVal v) = boolVAL False
STABLEOF (SigVal t1 t2 v wf) = natVAL t2
STABLEOF (VarVal v) = natVAL Zero
```

2.7.1 The signal state

A SIGSTATE assigns values to the names (signals and variables). The initial SIGSTATE, STARTSIG, is a function that assigns the UNINITIALIZED value to all name. and an empty list of drivers to all signals.

```
type SIGSTATE  = NAME->NAMEVAL
type SIGSTATE_ITEM  = <<BOOL,NAME,NAMEVAL>>

STARTSIG :: SIGSTATE
STARTSIG n {KINDOF n = SIG_KIND} = SigVal #0 #0 UNINITIALIZED []
STARTSIG n {KINDOF n = VAR_KIND} = VarVal UNINITIALIZED
```

A SIGSTATE is meant to map names that are signals to SigVals and names that are variables to VarVals. We define a predicate that is true of such SIGSTATEs.

```
PROPER 's' := (name)
              'IS_SigVal (s name)'='KINDOF name = SIG_KIND'
               & 'IS_VarVal (s name)'='KINDOF name = VAR_KIND'
```

2.7.2 Advancing the global clock

Here we define functions which advance the global clock and update the current values of the signals. When the global clock advances by a non-zero amount, we decrease the time components of all waveforms in drivers and increase the times in the "time since last event" and "time since last active" components. Then we do a δ-transition. A δ-transition does not advance the clock, but if the time component of any waveform in a driver is zero, then its value becomes the current value of the driver. Then the current value of the signal is computed

from the current values of its drivers. If there is more than one driver for a signal this involves resolution. Assigned to each signal is a commutative-associative resolution function. (The default is always undefined.)

```
Resolve :: NAME->VALUE->VALUE->VALUE
AXIOM (S)(X)(Y) 'Resolve S X Y' = 'Resolve S Y X'
AXIOM (S)(X)(Y)(Z)
      'Resolve S X (Resolve S Y Z)'='Resolve S (Resolve S X Y) Z'
```

If there is more than one driver for a signal then its resolution function is applied to the list of current values of its drivers.

We start with the function ADVANCE, which advances the signal-state, S, by T units of time. This is done by advancing the signal-value for each signal, SIG. The function ADVANCE-NAMEVAL does this, and it is provided with the resolution function, (Resolve SIG), assigned to SIG, which is used when there are multiple drivers of SIG.

```
ADVANCE :: TIME->SIGSTATE->SIGSTATE
ADVANCE T S SIG = ADVANCE_NAMEVAL T (Resolve SIG) (S SIG)
```

To advance a signal-value, we adjust all the times by T units, and then perform a DELTA, which computes new values for signals with active drivers.

```
ADVANCE_NAMEVAL T F (VarVal V ) = VarVal V
ADVANCE_NAMEVAL Zero F SV = DELTA F SV
ADVANCE_NAMEVAL T F (SigVal TM1 TM2 V DVRS) =
    DELTA F
     (SigVal (TM1 #+ T) (TM2 #+ T) V (MAP (DVR_MINUS T) DVRS))
```

To do a DELTA transition on a signal-value, we check whether any of its drivers are active, and if so advance all the active drivers, and (if there's more than one driver) resolve their values to get the new value, newv. Then we update the last-active to Zero, and adjust the last-event if the value has changed.

```
DELTA f (SigVal t1 t2 v  dvrs) {DRIVERS_ACTIVE dvrs} =
   SigVal Zero lastevt newv newdvrs
        where newdvrs = MAP ADVANCE_DRIVER dvrs
              newv = doResolve f (MAP DRIVERVAL newdvrs)
              lastevt = (v=newv)->t2;Zero
DELTA f sv = sv

doResolve f [a] = a
doResolve f (a:b:more) = doResolve f ((f a b):more)
```

2.8 Evaluating Expressions

Expressions are built up from names and constants using the predefined operators and aggregates. Given an instance we can resolve all the names to atomic

signals and variables. Given a signal-state we can find the current value of the atomic signals and variables. Thus, given an instance and a signal-state, we may evaluate an expression. The definition of OPERATORMEANING is given in section 2.9.

```
EVAL INST SS (BINARY e1 op e2) =
    OPERATORMEANING op (EVAL INST SS e1 )(EVAL INST SS e2 )
EVAL INST SS (UNARY op e1) =
    UOPERATORMEANING op (EVAL INST SS e1)
EVAL INST SS (AGGREGATEEXP L) =
    AGGREGATE(MAP (EVAL INST SS ) L)
EVAL INST SS (EXPNAME n) = CURRENT INST SS n
EVAL INST SS (EXPCONSTANT v) = v

ISTRUE e INST S = ((EVAL INST S e) = (boolVAL True))
```

To find the current value of a name given the instance and state, we first use the instance (and state) to find the real name, which is a name in which references to port connections and generic values have been resolved using the instance. Also "local" names are made absolute by adding the instance name. The value of the real name then depends only on the current state.

```
CURRENT INST SS N = CURRENTVAL SS (REALNAME INST SS N)
```

The current value of an ATOMIC name is read directly from the signal state. Otherwise the aggregate value is formed from the current values of the subnames.

```
CURRENTVAL SS (CONSTANT V) = V
CURRENTVAL SS (ATTRIBUTEOF A N) = ATTRIBUTEVAL A SS N
CURRENTVAL SS N {TYPEOF N = ATOMIC} = VALOF (SS N)
CURRENTVAL SS N =
       AGGREGATE (MAP (CURRENTVAL SS) (SUBNAMES N (TYPEOF N)))

SUBNAMES N ATOMIC = [N]
SUBNAMES N (RECORD L) = MAP (SELECT N) L
SUBNAMES N (ARRAY L T) = MAP (INDEX N) L
SUBNAMES N (ARRAY_UNCONSTRAINED T) = bottom
```

The meanings of signal attributes are defined below. We haven't defined all the attributes that are predefined in VHDL. These were all that were needed in Scoreboard.

```
ATTRIBUTEVAL "event" SS N = EVENTOF (SS N)
ATTRIBUTEVAL "stable" SS N = STABLEOF (SS N)
ATTRIBUTEVAL "left" SS N = LEFTOF N
```

```
ATTRIBUTEVAL "length" SS N = LENGTHOF N

LEFTOF (SIGNALNAME X N (ARRAY L T)) = natVAL (hd L)
LEFTOF (VARIABLENAME X N (ARRAY L T)) = natVAL (hd L)
LENGTHOF (SIGNALNAME X N (ARRAY L T)) = natVAL (## L)
LENGTHOF (VARIABLENAME X N (ARRAY L T)) = natVAL (## L)
```

As described earlier, the REALNAME replaces relative names by absolute names, and resolves references to ports and generics. It also evaluates expressions that occur in names (in function applications and array applications).

```
REALNAME INST SS (SIGNALNAME X N T) =
          SIGNALNAME ((INST_NAME INST)++X) N T
REALNAME INST SS (VARIABLENAME X N T) =
          VARIABLENAME ((INST_NAME INST)++X) N T
REALNAME INST SS (SELECT N1 N2) = SELECT (REALNAME INST SS N1) N2
REALNAME INST SS (INDEX N1 N2) = INDEX (REALNAME INST SS N1) N2
REALNAME INST SS (PORTNAME id) = PORTCONNECT INST (PORTNAME id)
REALNAME INST SS (GENNAME id) = CONSTANT (GENVAL INST id)
REALNAME INST SS (ARRAYAPPLY N E) =
          INDEX (REALNAME INST SS N) (natVALOF(EVAL INST SS E))
REALNAME INST SS (ATTRIBUTEOF A N ) =
          ATTRIBUTEOF A (REALNAME INST SS N)
REALNAME INST SS (FUNCAPPLY f E) =
          CONSTANT(FUNCMEANING f (EVAL INST SS E))
REALNAME INST SS N = N
```

To find out what a port is connected to, we need only the INSTANCE. The connection cannot depend on the state, e.g. a port cannot be connected to A(x) where x is an expression containing variables. When we resolve such a connection we check that the expression x is state-independent. The definition of state-independent is given in section 2.9. A port can only be connected to certain names, namely: signals of its parent, ports of its parent, and to components of array or record aggregates of its parent. Any other connection will resolve to ILLEGAL.

```
PORTCONNECT (Inst Id Pt Gn Parent) (PORTNAME X) =
    PORTCONNECT Parent (_update PORTNAME Pt X)
PORTCONNECT TOP (PORTNAME X) = ILLEGAL X
PORTCONNECT INST (SIGNALNAME [] N T) =
    SIGNALNAME (INST_NAME INST) N T
PORTCONNECT INST (VARIABLENAME [] N T) =
    VARIABLENAME (INST_NAME INST) N T
PORTCONNECT INST (ARRAYAPPLY N E) {STATE_INDEPENDENT E} =
    INDEX (PORTCONNECT INST N) (natVALOF(EVAL INST bottom E))
```

```
PORTCONNECT INST (SELECT N1 N2) = SELECT (PORTCONNECT INST N1) N2
PORTCONNECT INST N = ILLEGAL "PORT"


GENVAL TOP N = UNINITIALIZED
GENVAL (Inst Id Pt Gn Parent) N = GEN_LOOKUP Gn N
```

Waveform expressions were built up from lists of time-expression, value-expression pairs using conditionals. They are evaluated to waveforms using the instance and state as follows:

```
WAVEFORM_EVAL :: WAVEFORM_EXP -> INSTANCE -> SIGSTATE -> WAVEFORM
WAVEFORM_EVAL (WFEXP wf) INST s = MAP (WF_ELEM_EVAL INST s) wf
WAVEFORM_EVAL (CONDWF e wf1 wf2 ) INST s =
    (ISTRUE e INST s)->(WAVEFORM_EVAL wf1 INST s);
                       (WAVEFORM_EVAL wf2 INST s)
WF_ELEM_EVAL INST s <<e1,e2>> =
    <<natVALOF(EVAL INST s e1), (EVAL INST s e2)>>
```

2.8.1 Function meanings

In the definition of REALNAME, a name which is a function application is evaluated by evaluating its arguments and then applying whatever function is bound to the function name. This binding should come from an environment, so that EVAL would need an instance, a state, and an environment. But our current translator does not allow functions to be declared locally. All functions are declared globally. Therefore the translator defines a global function, FUNCMEANING, which assigns a meaning to each function name. The meaning is a function from values to values. A function of several arguments is thought of as taking a single, aggregate, value as its argument. Thus, FUNCMEANING, acts as a global environment. We do not support procedures, but they should not be difficult to add to the semantics. The Scoreboard design does not use procedures.

```
FUNCMEANING :: FUNCTION -> VALUE -> VALUE
```

2.9 Functions on Expressions

The predefined operators are mapped to their meanings which were previously defined on VALUEs.

```
OPERATORMEANING ANDOP = AND
OPERATORMEANING OROP = OR
OPERATORMEANING XOROP = XOR
OPERATORMEANING NOROP = NOR
OPERATORMEANING NANDOP = NAND
```

```
OPERATORMEANING PLUSOP = PLUS
OPERATORMEANING MINUSOP = MINUS
OPERATORMEANING TIMESOP = TIMES
OPERATORMEANING DIVIDEOP = DIVIDE
OPERATORMEANING MODOP = MOD
OPERATORMEANING REMOP = REM
OPERATORMEANING EQUALOP = EQUAL
OPERATORMEANING NOTEQUALOP X Y = NOT (EQUAL X Y)
OPERATORMEANING GEQOP = GEQ
OPERATORMEANING LEQOP = LEQ
OPERATORMEANING LESSOP = LESS
OPERATORMEANING GREATEROP = GREATER
OPERATORMEANING AMPEROP = CONCAT

UOPERATORMEANING NOTOP = NOT
UOPERATORMEANING UMINUSOP = UMINUS
UOPERATORMEANING UPLUSOP = I
UOPERATORMEANING ABSOP = ABS
```

Some handy abbreviations.

```
EXPTRUE = EXPCONSTANT (boolVAL True)
EXPFALSE = EXPCONSTANT (boolVAL False)
EXPZERO = EXPCONSTANT (natVAL Zero)
EXPPLUS X Y = BINARY X PLUSOP Y
EXPGEQ X Y = BINARY X GEQOP Y
EXPEQUAL X Y = BINARY X EQUALOP Y
ATTRIBUTEEXP att n = EXPNAME (ATTRIBUTEOF att n)
```

There is a special name, `GLOBALCLOCK`, which cannot be assigned to. Therefore it will remain stable from the initial simulation cycle on. Its attribute, "stable", will therefore measure the elapsed time on the global clock. We abbreviate this as `CLOCK`.

```
CLOCK  = EXPNAME (ATTRIBUTEOF "stable" GLOBALCLOCK)
```

When we combine expressions using OR or AND, we can check for some simple reductions.

```
EXPOR (EXPCONSTANT (boolVAL False)) Y = Y
EXPOR X (EXPCONSTANT (boolVAL False)) = X
EXPOR (EXPCONSTANT (boolVAL True)) Y = EXPTRUE
EXPOR X (EXPCONSTANT (boolVAL True)) = EXPTRUE
EXPOR X Y = BINARY X OROP Y

EXPAND (EXPCONSTANT (boolVAL False)) Y = EXPFALSE
```

```
EXPAND X (EXPCONSTANT (boolVAL False)) = EXPFALSE
EXPAND (EXPCONSTANT (boolVAL True)) Y = Y
EXPAND X (EXPCONSTANT (boolVAL True)) = X
EXPAND X Y = BINARY X ANDOP Y
```

When we know that a name represents a constant, then we can get its value.

```
CONSTNAMEVAL (CONSTANT v) = v
```

The value of an expression that represents a constant, possibly depending on the values of the generics, does not depend on the current state. These are the only expressions that can occur in the names of connections to ports.

```
STATE_INDEPENDENT (BINARY e1 op e2)=
    (STATE_INDEPENDENT e1 ) & (STATE_INDEPENDENT e2 )
STATE_INDEPENDENT (UNARY op e1) = (STATE_INDEPENDENT e1 )
STATE_INDEPENDENT (AGGREGATEEXP L) = ALLTRUE STATE_INDEPENDENT L
STATE_INDEPENDENT (EXPNAME (GENNAME id)) = True
STATE_INDEPENDENT (EXPNAME (FUNCAPPLY f E)) = STATE_INDEPENDENT E
STATE_INDEPENDENT (EXPCONSTANT v) = True
STATE_INDEPENDENT E = False
```

Sometimes we need a list of all names mentioned in an expression. In particular, a signal assignment statement like:

```
x <= y1 & y2 & y3;
```

will be translated into a process that is waiting on an event on any of the names `y1,y2,y3` which are mentioned.

```
NAMES (BINARY e1 op e2) = (NAMES e1) ++ (NAMES e2)
NAMES (UNARY op e1) = NAMES e1
NAMES (AGGREGATEEXP L) = FOLDR (++) [] (MAP NAMES L)
NAMES (EXPNAME n) = [n]
NAMES (EXPCONSTANT v) = []
```

`(EXPEVENT e)` is an expression that evaluates to true in any state in which one of the names mentioned in expression `e` has an event. `(EXPTRUEEVENT e)` is an expression that evaluates to true in any state in which one of the names mentioned in expression `e` has an event and `e` evaluates to true. This is used to translate a wait statement like:

```
wait until clock = '1';
```

```
NAMESEVENT N =
    FOLDR EXPOR EXPFALSE (MAP (ATTRIBUTEEXP "event") N)
EXPEVENT e = NAMESEVENT (NAMES e)
EXPTRUEEVENT e = EXPAND (EXPEVENT e) e
```

To get an expression which is true when E equals one of the things on a list of CHOICEs, we use:

```
CHOICE ::= OTHERS | CHOOSE EXPRESSION
CHOICEEXP E [] = EXPFALSE
CHOICEEXP E (OTHERS:REST) = EXPTRUE
CHOICEEXP E ((CHOOSE A):REST) =
          EXPOR (EXPEQUAL E A) (CHOICEEXP E REST)
```

We can convert a waveform expression into an expression that is true when any signal mentioned in the waveform expression has an event.

```
WFEVENT (WFEXP L) = FOLDR EXPOR EXPFALSE (MAP WF_ELEM_EVENT L)
WFEVENT (CONDWF e wf1 wf2) =
    EXPOR (EXPEVENT e) (EXPOR (WFEVENT wf1) (WFEVENT wf2))
WF_ELEM_EVENT <<t,v>> = EXPEVENT v
```

A GENMAP is a list of name-value pairs. If we have a list of name-expression pairs, where each expressions is a constant, we can convert.

```
MAKE_GENMAP L = MAP F L
                where F <<N1,N2>> = <<N1,CONSTNAMEVAL N2>>
```

2.10 Processes

A VHDL design is a collection of instances of entities. Each entity is a collection of processes. Thus the process is the basic actor in our semantics. The process is what changes the state by updating the variables and adding waveforms to the signal drivers.

In any simulation cycle, a set of processes are *waiting*. If the condition a process is waiting on is true in that simulation cycle, it becomes active. The body of a process is a list of sequential statements. When a process becomes active, its body is executed. This generates state changes of two kinds. Changes to the variables are immediate. Changes to the signal drivers will not affect the values of signals at least until the next simulation cycle. When the process suspends, the result is that there is a new process waiting (which may be the original process, or some other process). The new process is the *continuation* of the process. Thus the result of executing a process is three things:

1. a list of changes to the variables
2. a list of scheduled changes to the signals
3. a new process

A single change to a variable can be represented as a single update to the SIGSTATE. We have already defined a type for such single updates, called

SIGSTATE-ITEM. A single scheduled change to a signal we call a TRANSACTION (see section 2.12). Thus, if we call the result of executing a process an ACTION-RESULT, we can make the following definition:

```
type ACTION_RESULT = <<[SIGSTATE_ITEM],[TRANSACTION],PROCESS>>
```

Now, the result of executing a process depends, of course, on the values of expressions in the body. Hence, it depends on the instance and signal-state. These are the only things the result depends on, so we can say that the meaning of a process body is a function from instance and signal-state to result.

```
type PROCESS_ACTION = INSTANCE->SIGSTATE->ACTION_RESULT
```

Now a single process is completely characterized by:

1. the condition it is waiting on
2. the meaning of its body
3. the entity instance it belongs to

So, we define the following data type in Caliban:

```
PROCESS ::= PROC EXPRESSION PROCESS_ACTION INSTANCE
```

Note that PROCESS is constructed from PROCESS-ACTION which is a function type that returns ACTION-RESULT which includes PROCESS. Thus, PROCESS is a recursively defined type. The type system of Caliban and the lazy evaluation semantics of Caliban make this sort of definition convenient.

Now we can define how a process is run in a given signal-state, `SS`. If the process, `P`, is `(PROC W A INST)`, then we use the given instance `INST`, and state `SS` to evaluate the wait condition `W`. If it evaluates to true, then the body `A` is applied to the instance and state to give the result. Otherwise, the result is `<<[],[],P>>`, which means that no changes will be made to the state, and the process `P` will still be waiting in the next simulation cycle.

```
RUN_PROC :: SIGSTATE->PROCESS->ACTION_RESULT

RUN_PROC SS (PROC W A INST) = (A INST SS), (ISTRUE W INST SS)
                              <<[],[],(PROC W A INST)>>
```

In section 2.1 we defined the type:

```
type RESULT = <<[SIGSTATE_ITEM],[PROCESS],[TIME]>>
```

This is the information about the result of running a process that the top level function, `Simulate`, uses. Here the first component is a list of changes to the SIGSTATE which includes both the immediate changes to the variables and the scheduled changes to the signals. The second component is the process

continuation made into a singleton list so that it can be appended to the continuations of the other processes. The third component is the list of times at which transactions are scheduled.

Since the function RUN-PROC returns and ACTION-RESULT we must convert this into a RESULT as follows: The ACTION-RESULT, <<VAR-UPDTS,TRS,CONT>>, contains the updates to the variables, a list of transactions, and a continuation. In section 2.12 we define TRANSACTIONS-TIMES which maps a list of transactions to the times of the scheduled events. We use this to define the third component of the result. The second component of the result is just a singleton list containing the process continuation CONT. So, we must explain how the first component of the following definition is computed.

```
PROC_RESULT :: SIGSTATE->ACTION_RESULT->RESULT
PROC_RESULT S <<VAR_UPDTS,TRS,CONT>> =
   << (TRANSACTION_UPDATES S TRS) ++ VAR_UPDTS,
     [CONT],
     TRANSACTIONS_TIMES TRS
   >>
```

A transaction represents a scheduled change to a signal. This change is accomplished by changing the signal's list of drivers. Thus, the transaction corresponds to a change of the signal state. This change is computed by taking the current state S, extracting the current set of drivers for the given signal, and then merging the additional driver from the transaction. This computation is done by the function TRANSACTION-UPDATES which is defined in section 2.12. This list of signal-updates is then appended to the variable updates.

By composing this conversion, PROC-RESULT, with RUN-PROC we obtain the function DO-PROC which is used in the definition of Simulate.

```
DO_PROC S P = PROC_RESULT S (RUN_PROC S P)
```

2.11 Conditional function updates

The reader may have noticed that some definitions involving a single update to a function (say from NAME to VALUE) represent the update not as a pair <<NAME, VALUE>> but as a triple <<BOOL,NAME, VALUE>> . We call such a triple a *conditional function update*. To motivate their introduction, consider the following example in an unspecified programming language L.

```
if a then x := 1; else x := 2; endif
if b then y := 3; else y := 4; endif
```

If we give L an operational semantics similar to the semantics we have given for VHDL, then a state, S, would be a function from names to values. Then the effect of executing the first statement of the example in an arbitrary state S would be:

```
if a1 then (_update S [<<x,1>>]) else (_update S [<<x, 2>>])
    where a1 = (eval S a)
```

Now, if we compose the meanings of the two statements in the example in the natural way to get the meaning of the sequence, then the effect of the sequence would be:

```
if a1 then if b1 then (_update S [<<x,1>>,<<y,3>>])
                 else (_update S [<<x,1>>,<<y,4>>])
      else if b1 then (_update S [<<x, 2>>,<<y,3>>])
                 else (_update S [<<x, 2>>,<<y,4>>])
```

We can see the beginnings of an exponential "explosion" in the sizes of conditional expressions that result from symbolic execution of the semantics.

Since the changes to "x" and "y" in the example are independent, we would rather have the resulting expression for the state be:

```
_update S [<<x,if a1 then 1 else 2>>,<<y,if b1 then 3 else 4>>]
```

In this form, many such updates could be composed (provided they were independent) without an exponential explosion.

We have found no easy way to define the semantics so that they directly give results as in the above expression. However, by introducing the conditional function update we get similar results in a natural way. Using the conditional updates, the effect of the example would be:

```
__update S [<<a1,x,1>>,<<~a1,x,2>>,<<b1,y,3>>,<<~b1,y,4>>]
```

The operators `_update` and `__update` are both predefined in Caliban. The conditional update, `__update`, acts as follows:

```
__update S (<<a,x,v>>:more) arg {a && x=arg} = v
__update S (<<a,x,v>>:more) arg  = __update S more arg
__update S [] arg  = S arg
```

(Note that the "most recent" updates are at the head of the list. Therefore, when interpreting a list of updates as a history of changes, the list must be "read backwards").

From this definition we may prove a rewrite rule:

```
'__update S (<<a,x,v>>:<<b,x,w>>:more) '
  = '__update S (<<a | b, x, (a->v;w)>>:more)'
```

This rule says that two conditional updates to the same "address" can be combined by "or-ing" their conditions and making the "value" conditional. Using this rewrite rule the effect of our example rewites to:

```
__update S [<<(a1|~a1),x,(a1->1;2)>>,<<(b1|~b1),y,(b1->3;4)>>]
```

This simplifies to:

```
__update S [<<True,x,(a1->1;2)>>,<<True,y,(b1->3;4)>>]
```

This form is essentially the form originally desired.

These rewrite rules (and others, such as deletion of updates with condition False) are built-in to Clio. In several other efforts we have found that the introduction of conditional updates was the key technical "trick" needed to make an operational semantics usable.

In the next section we define the type **TRANSACTION**, and we introduce conditional transactions that behave with respect to transactions the way conditional updates behave with respect to function updates.

2.12 Transactions

One result of executing a process may be that a driver is added to the list of drivers of a signal or is merged with an existing driver. We define a data type, **TRANSACTION**, that holds the information needed to record one such result. The production **TR** constructs a simple transaction assigning a driver to a name; it also records the delay kind (which is either transport or inertial).

```
TRANSACTION ::= TR DELAY_KIND NAME DRIVER
              | CONDTR BOOL TRANSACTION
              | DUMMY NAT
```

As explained in the previous section, it is convenient to be able to tag a transaction with a boolean to make a conditional transaction. That is what the production **CONDTR** is for. A transaction contains a driver which contains a waveform. The time components of the waveform are the times at which the driver is scheduled to change values, hence they are the times at which a simulation cycle must happen. There is one case in which we schedule a simulation cycle even though there may be no driver scheduled to change value. That is the case of a "timeout wait" as in:

```
wait for 10 ns;
```

In this case we want to schedule a simulation cycle for **10 ns** in the future. In our semantics, the next simulation cycles is scheduled for the next transaction time. Therefore, to handle the timeout wait, we also have a **DUMMY** transaction that has a time but no driver or signal name.

Here is how we extract the transaction times from a list of transactions.

```
TRANSACTIONS_TIMES :: [TRANSACTION]-> [NAT]
TRANSACTIONS_TIMES TL = MERGE_LIST (MAP TRANSACTION_TIMES TL)

TRANSACTION_TIMES :: TRANSACTION-> [NAT]
```

```
TRANSACTION_TIMES (DUMMY T) = [T]
TRANSACTION_TIMES (TR KND SIG DVR) = MAP TIMEOF (WAVEFORMOF DVR)
TRANSACTION_TIMES (CONDTR BEXP TR1) = TRANSACTION_TIMES TR1
```

Every non-DUMMY transaction, corresponds to a (conditional) change to the signal state, S. The change to S corresponding to the transaction, T, is a single conditional update and is computed by extracting the current set of drivers for the given signal, and then merging the additional driver from the transaction.

```
SIG_UPDATE S T =
   <<CONDOF T, SIGOF T, UPDATE_DRIVERS (S (SIGOF T)) (BODYOF T)>>

UPDATE_DRIVERS (SigVal TM WE V DVRS) (TR KND SIG DVR) =
                        SigVal TM WE V (MERGE_DRIVER KND DVRS DVR)

SIGOF (TR KND SIG DVR) = SIG
SIGOF (CONDTR BEXP T) = SIGOF T

CONDOF (TR KND SIG DVR) = True
CONDOF (CONDTR BEXP T) = BEXP && (CONDOF T)

BODYOF (CONDTR BEXP T) = BODYOF T
BODYOF T = T
```

When we define the result of a process in section 2.10, we convert a list of transactions into a list of (conditional) updates to the signal state. Since the dummy transactions do not affect the state they are removed before computing the list of updates.

```
TRANSACTION_UPDATES :: SIGSTATE->[TRANSACTION] -> [SIGSTATE_ITEM]
TRANSACTION_UPDATES S TL = MAP (SIG_UPDATE S) (REMOVE_DUMMY TL)

REMOVE_DUMMY [] = []
REMOVE_DUMMY ((DUMMY X):MORE) = REMOVE_DUMMY MORE
REMOVE_DUMMY (T:MORE) = T: (REMOVE_DUMMY MORE)
```

As explained in section 2.11, conditional function updates and conditional transactions have been introduced in order to define the effect of if-then-else statements in an efficient way. To give a meaning to the statement sequence:

```
if BEXP then A else B endif; C;
```

Suppose that the statements in A generate a list of transactions, TR1, the statements in B generate the transactions TR2, and the statements in C generate transactions TR0. Then the combined list of conditional transactions must contain the list TR1, conditional on BEXP, and the list TR2, conditional on ¬BEXP, and the list TR0 unconditionally.

```
COND_TRANS BEXP TRO TR1 TR2 =
    TRO
    ++ (MAP (ADDCOND BEXP) TR1)
    ++ (MAP (ADDCOND (~BEXP)) TR2)

ADDCOND BEXP (CONDTR BEXP2 X) = CONDTR (BEXP && BEXP2) X
ADDCOND BEXP X = CONDTR BEXP X
```

For variable updates generated by statements, we use conditional function updates directly (rather than transactions). For use in defining the if-then-else statement we define a function similar to COND_TRANS that works on conditional function updates.

```
COND_VAR_UPDATE BEXP S0 S1 S2 =
    S0
    ++ (MAP (ADDCOND1 BEXP) S1)
    ++ (MAP (ADDCOND1 (~BEXP)) S2)

ADDCOND1 BEXP <<BEXP2,NM,VAL>> = <<BEXP && BEXP2,NM,VAL>>
```

2.13 Statements

The body of a process is composed of sequential statements. Recall that the meaning of a process body was assigned the type:

```
type PROCESS_ACTION = INSTANCE->SIGSTATE->ACTION_RESULT
```

where

```
type ACTION_RESULT = <<[SIGSTATE_ITEM],[TRANSACTION],PROCESS>>
```

We use the standard continuation semantics for sequential statements by letting the meaning, `M S`, of a statement `S`, be the transformation from `PROCESS-ACTION` to `PROCESS-ACTION` which is describe informally as follows. If `A` is the meaning of a process body containing sequential statements `SS`, then `M S A` is the meaning of the process body containing the sequence `S;SS` . Hence, the meaning of any sequential statement has the following type:

```
type PROCESS_TRANSFORMATION = PROCESS_ACTION -> PROCESS_ACTION
```

We next give a Caliban data type that encodes the abstract syntax of the sequential statements that we currently support. Thus, at the sequential statement level, we are making a deep embedding of VHDL into Caliban. We will discuss each kind of statement when we define its meaning. The main omissions from this set are procedure call statements, assertion statements, next statements, and while loops.

```
STATEMENT ::= SIGASSIGN DELAY_KIND NAME WAVEFORM_EXP
            | WAIT EXPRESSION
            | TIMEOUTWAIT EXPRESSION
            | IF_THEN_ELSE EXPRESSION [STATEMENT] [STATEMENT]
            | CASE [<<EXPRESSION,[STATEMENT]>>]
            | RETURN EXPRESSION
            | EXIT
            | FORLOOP NAME [NAT] [STATEMENT]
            | VARASSIGN NAME EXPRESSION
            | NULL
```

As stated above, the meaning of a STATEMENT is a PROCESS-TRANSFORMATION. Therefore we define a meaning function:

```
M_S :: STATEMENT -> PROCESS_TRANSFORMATION
```

In order to define `M_S ST`, where `ST` is a statement, we must define `(M_S ST A INST S)` where `A` is a `PROCESS-ACTION` representing the effect of the statements that come after `ST`, and `INST` and `S` are the current instance and signal-state. Then `(M_S ST A INST S)` must be defined to correspond to the effect of executing `ST` in instance `INST` and signal-state `S`, and then executing the action `A` in the resulting state. We will call the action `A` the *continuation* in this section, but this is a continuation of a statement sequence and is different from the process continuation mentioned in section 2.10, which was the process that results when a process suspends.

2.13.1 Signal assignment statement

The signal assignment `SIGASSIGN knd name wf` is the abstract syntax for the VHDL statement `name <= wf`, (with optional key word "transport" to indicate transport delay). To execute this statement with instance `INST` and state `S`, we evaluate the waveform expression `wf` to get a waveform, make a driver labeled with the instance name, resolve the name being assigned to its real name, and then make a transaction out of the delay-kind, the resolved name and the driver. This transaction has no immediate effect on the state, so `(A INST S)` is the effect of the continuation. The effect of the signal assignment followed by its continuation is obtained by appending the transaction to transaction list returned by the continuation.

```
M_S (SIGASSIGN knd name wf) A INST S =
    APPEND_TRANSACTION
        (TR knd (REALNAME INST S name) (DRIVER_EVAL INST wf S))
        (A INST S)

DRIVER_EVAL INST wf s =
    DR (INST_NAME INST) UNINITIALIZED (WAVEFORM_EVAL wf INST s)
```

```
APPEND_TRANSACTION t <<s,ts,c>> = <<s,t:ts,c>>
```

2.13.2 Variable assignment statement

A variable assignment `N := E;` is similar to the signal assignment in that we evaluate `E` and resolve the name `N` to get a single state update `X`. Since variable assignments are immediate, we must update the state `S` with the single change `X` before executing the continuation. The change `X` is appended to the changes made by the continuation.

```
M_S (VARASSIGN N E) A INST S =
    APPEND_VAR_UPDATE X (A INST (__update S [X]))
    where X = VARUP N E INST S

VARUP N E INST SS =
    <<True,(REALNAME INST SS N), VarVal (EVAL INST SS E)>>
APPEND_VAR_UPDATE X <<L1,L2,CONT>> = <<L1++[X],L2,CONT>>
```

2.13.3 Wait statement

The wait statement `WAIT W` is the abstract syntax of the statement **`wait until`** `W`. When `A` is its continuation, then the effect of the wait statement is to make no state changes, and to suspend the process. The *process continuation* is then the process that waits on `W` and has `A` as its body, and `INST` as its instance.

```
M_S (WAIT W) A INST S = <<[], [], (PROC W A INST) >>
```

A wait statement of the form **`wait for t`** is a timeout wait. It is equivalent to `temp := CLOCK; wait until CLOCK >= temp+t;` And it schedules a dummy transaction after t, to make sure that there will be a simulation then even if there were no other transactions then.

```
M_S (TIMEOUTWAIT t) A INST S =
    <<[], [DUMMY (natVALOF(EVAL INST S t))], TIMEOUT t A INST S>>

TIMEOUT t A INST S =
    PROC (TIMEOUTTEST (EVAL INST S (EXPPLUS CLOCK t))) A INST
TIMEOUTTEST v = EXPGEQ CLOCK (EXPCONSTANT v)
```

2.13.4 Null statement

A null statement does nothing, so as a PROCESS-TRANSFORMATION it is the identity.

```
M_S NULL = I
```

2.13.5 Conditional statement

The statement (IF_THEN_ELSE e ss1 ss2) is the abstract syntax of the statement **if e then ss1 else ss2**. Its meaning is derived from the expression **e** and the meanings of the two statement sequences, **ss1** and **ss2**. The function M_SS gives the meaning of a statement sequence and is defined in section 2.13.11.

```
M_S (IF_THEN_ELSE e ss1 ss2) = IFMEANING e (M_SS ss1) (M_SS ss2)

IFMEANING :: EXPRESSION
               -> PROCESS_TRANSFORMATION
                  -> PROCESS_TRANSFORMATION
                     -> PROCESS_TRANSFORMATION
```

Now to understand the definition of IFMEANING, remember that (A INST S) is the result of executing the continuation (the statements that come after the if-then-else) and that (IFMEANING e ss1 ss2 A INST S) is the result of executing the if-then-else and then the continuation. Basically, we apply the meanings of the two branches of the conditional to the continuation and then make the resulting transactions conditional on the value of the expression **e**. However, this would make all the transactions from the continuation appear twice, as conditional on **e** being true and on **e** being false. This would lead to an "explosion" of conditional expressions in a simulation. Therefore we make our definition as follows. We let **A1** be an action which is the same as **A** except that it generates no transaction (so it makes the same variable updates and has the same process continuation). We let **A1** be the continuation of the two branches, **ss1** and **ss2**, of the conditional. Then we combine the results of the two branches and the original continuation.

```
IFMEANING (EXPCONSTANT (boolVAL True)) ss1 ss2 = ss1
IFMEANING (EXPCONSTANT (boolVAL False)) ss1 ss2 = ss2
IFMEANING e ss1 ss2 A INST S =
    CONDACTION (ISTRUE e INST S)
               (A INST S)
               (ss1 A1 INST S)
               (ss2 A1 INST S)
    where A1 = DELETE_TRANS A

DELETE_TRANS A INST S = REMOVE_TRANSACTIONS (A INST S)
REMOVE_TRANSACTIONS <<S,TRS,CONT>> = <<S,[],CONT>>
```

The functions COND_VAR_UPDATE and COND_TRANS have already been defined. They take three lists of transactions and return a list of conditional transactions where the first of the three argument lists is left unconditional, and the second and third argument lists are made conditional on BEXP and ¬BEXP.

```
CONDACTION BEXP <<S0,TR0,C0>> <<S1,TR1,C1>> <<S2,TR2,C2>> =
     <<COND_VAR_UPDATE BEXP S0 S1 S2,
       COND_TRANS BEXP TR0 TR1 TR2,
       (BEXP->C1;C2)>>
```

2.13.6 Case statement

The statement `case e is when x1 => ss1 ...when xn => ssn end case` is, abstractly, `CASE [<<e=x1,ss1>>,...,<<e=xn,ssn>>]`, so it is given by a list of `<<EXPRESSION,[STATEMENT]>>` pairs. We reduce its meaning to the IFMEANING as follows:

```
M_S (CASE []) = I
M_S (CASE (<<e,ss>>:rest)) =
        IFMEANING e (M_SS ss) (M_S (CASE rest))
```

2.13.7 Return statement

The return statement "return e" is the same as "FUNCRETURN := e; exit;" for a special name "FUNCRETURN".

```
M_S (RETURN e) A INST S = (SAVERETURN e (M_S EXIT A) INST S)
SAVERETURN e A INST S =
    A INST (__update S [VARUP (FUNCRETURN ATOMIC) e INST S])
```

2.13.8 Exit statement

The exit statement, does nothing and continues to the dead process.

```
M_S EXIT A INST S = <<[],[],DEADPROC INST >>

DEADPROC = PROC EXPFALSE (M_S EXIT bottom)
```

2.13.9 ForLoop statement

The statement `(FORLOOP i L B)` is the abstract syntax of the statement `for i in L loop B endloop`. We can define the meaning of this statement by primitive recursion on the list `L`. If `L` is empty then the statement is a null statement. If not, we compose the meanings of:

1. Assign head of `L` to variable `i`.
2. Execute the body, `B`.
3. Do for-loop for `i` in the tail of `L`.

```
M_S (FORLOOP i [] B) = I
M_S (FORLOOP i (n:rest) B) =
    (M_S (VARASSIGN i (EXPCONSTANT (natVAL n))))
     .(M_SS B)
      .(M_S (FORLOOP i rest B))
```

2.13.10 Function meanings

When the translator recognizes the body of a function, F, with arguments x,y,z, and a body consisting of a statment sequence B, it enters the meaning of the function into the global environment by defining

```
FUNCMEANING (FUNCNAME "F") = FUNCEFFECT [x,y,z] (M_SS B)
```

The meaning of a function is of type VALUE → VALUE, and is obtained as follows. Given the values of the actuals as a single aggregate value v, we make a state (BIND nms v) that binds those values to the formals, nms. Then we evaluate the effect of the body in that state (with instance TOP) and get the value that is bound to the special variable name (FUNCRETURN ATOMIC).

```
FUNCEFFECT :: [NAME] ->PROCESS_TRANSFORMATION->VALUE->VALUE
FUNCEFFECT nms body v =
    CURRENT TOP (EFFECT body TOP (BIND nms v))
                    (FUNCRETURN ATOMIC)
```

The effect of the body is obtained by giving it a continuation which is the meaning of the EXIT statement (applied to the undefined continuation "bottom"). From the result we extract only updates to variables (functions may not have side effects on signals) and apply those updates to the initial state.

```
EFFECT :: PROCESS_TRANSFORMATION->INSTANCE->SIGSTATE->SIGSTATE
EFFECT body INST SS =
   __update SS (SIGUPDATES ((body (M_S EXIT bottom) ) INST SS))
SIGUPDATES <<S,T,CONT>> = S
```

The state (BIND nms v) is obtained by updating the uninitialized STARTSIG so that each name on the list nms is bound to the corresponding value in the aggregate v.

```
BIND [] v = STARTSIG
BIND [a] v = _update STARTSIG [<<a,VarVal v>>]
BIND nms (AGGREGATE v) = _update STARTSIG (BINDLIST nms v)
BINDLIST (a:b) (v:w) = <<a,VarVal v>>:(BINDLIST b w)
BINDLIST [] v = []
```

2.13.11 Statement sequences

Now, to give a meaning to a list of sequential statments, we compose their meanings to get a PROCESS-TRANSFORMATION. The "dot" operator is Caliban notation for function composition.

```
M_SS = FOLDR ((.).M_S) I
```

2.13.12 Process definition

A process with wait condition `w`, body `ss`, and instance `INST`, can be obtained via the following recursive definition.

```
MAKEPROCESS :: EXPRESSION->[STATEMENT]->INSTANCE->PROCESS
MAKEPROCESS w ss INST =
    PROC w (M_SS ss (NULLACTION (MAKEPROCESS w ss INST ))) INST
NULLACTION c INST s = <<[],[],c>>
```

If the body `ss` were null, then the process would produce no transactions and would be its own continuation. Reading the above definition with `(M_SS ss)` replaced by the identity, we see that it defines just such a process. If we let that process be the continuation on which the meaning `(M_SS ss)` acts, we see that the above definition defines the desired process. The `MAKEPROCESS` function is used by the translator to make the translations of concurrent statements such as process statements.

2.14 Useful general purpose functions

```
I F = F
K F X = F
C F X Y = F Y X

COND b X Y = b->X;Y

MERGE [] LST = LST
MERGE LST [] = LST
MERGE (A:As) (B:Bs) = (A<B)-> A : MERGE As (B:Bs)
                      (A=B)-> A : MERGE As Bs
                      B : MERGE (A:As) Bs

FOLDR F E [] = E
FOLDR F E (A:X) = F A (FOLDR F E X)

MAP F  [] = []
MAP F (A:X) = (F A): (MAP F X)
```

```
MAPOP F X [] = X
MAPOP F [] X = X
MAPOP F (A:X) (B:Y) = (F A B): (MAPOP F X Y)

FILTER P [] = []
FILTER P (A:X) = (P A) -> (A:FILTER P X); (FILTER P X)

ALLTRUE F [] = True
ALLTRUE F (A:X) = (F A) & (ALLTRUE F X)

MAP_AT L X = MAP (AT X) L
AT X F = F X

MERGE_LIST :: [[*]]->[*]
MERGE_LIST = FOLDR (MERGE) []


DIRECTION ::= UPTO | DOWNTO

FROMTO UPTO N M {N <= M} = N:(FROMTO UPTO (Succ N) M)
FROMTO UPTO N M  = []

FROMTO DOWNTO N M {N > M} = N:(FROMTO DOWNTO (N #- #1) M)
FROMTO DOWNTO N M {N = M} = [N]
FROMTO DOWNTO N M = []
```

3 The Translator

In defining our semantics we have made a *deep embedding* of VHDL up to the level of sequential statement sequences. This means that we have defined Caliban data types that are isomorphic to the abstract syntax of expressions, statements, and statement sequences. Because of this, the translator is mostly just a parser. We built our translator using yacc, starting with a public yacc grammar for VHDL obtained over the net.

As the parser recognizes each grammatical item, it writes the corresponding Caliban text to its output stream. So that the output will be readable, we make many abbreviations. For example, if the translator recognizes a waveform expression on line 20 of the file being translated, it will write a line like:

```
WAVEFORM20 = (WFEXP ... )
```

Then, for example, the signal assignment statement on that line, which contained the waveform expression might be translated as:

```
SIGASSIGN20 = SIGASSIGN INERTIAL clockstate WAVEFORM20
```

```
1: package clock_package is
2:    subtype output_type is BIT;
3: end clock_package;

5: entity clock_ent is
6:    generic (
7:        clock_delay : TIME := 1 ns
8:        );
9:    port (
10:       clock_out : out output_type
11:        );
12: end clock_ent;

14: architecture clock_arch of clock_ent is
15:    signal clockstate : BIT;
16: begin
17:    process (clockstate)
18:       begin
19:        if clockstate = '0' then
20:            clockstate <= '1' after clock_delay;
21:            clock_out <= '1' after clock_delay;
22:        else
23:            clockstate <= '0' after clock_delay;
24:            clock_out <= '0' after clock_delay;
25:        end if;
26:       end process;
27: end clock_arch;
```

Figure 1: VHDL clock.

With such abbreviations, each line of the translation is short and each abbreviation can be traced back to the original VHDL text.

We illustrate the translation process with the simple VHDL design for a clock in figure 1. (We have numbered the lines in order to refer to them later).

The translation is given in figure 2.

We will examine the translation line by line. The first line causes all the definitions of the vhdl semantics to be loaded.

```
FROM vhdl IMPORT ALL
```

We do not currently support the "with" and "use" clauses, so, currently, all packages are globally visible. Thus the lines 1-3 of the example are not translated

```
FROM vhdl IMPORT ALL
||*************************************************************
|| Declaration of entity clock_ent.
DEFAULT_VALUE "clock_delay" = CONSTVAL (EXPCONSTANT (natVAL #1))
ENTITY ::= clock_ent |+
||*************************************************************
|| Architecture clock_arch of clock_ent.
clockstate = SIGNALNAME [] "clockstate" ATOMIC
SIGLIST17 = [clockstate]
CONDITION19 = (BINARY (EXPNAME clockstate)
                  EQUALOP (EXPCONSTANT (bitVAL bit0)))
WAVEFORM20 = (WFEXP [<<(EXPNAME (GENNAME "clock_delay")),
                           (EXPCONSTANT (bitVAL bit1))>>])
SIGASSIGN20 = SIGASSIGN INERTIAL clockstate WAVEFORM20
SIGASSIGN21 =
    SIGASSIGN INERTIAL (PORTNAME "clock_out") WAVEFORM20
SS22 = [SIGASSIGN20, SIGASSIGN21]
WAVEFORM23 = (WFEXP [<<(EXPNAME (GENNAME "clock_delay")),
                          (EXPCONSTANT (bitVAL bit0))>>])
SIGASSIGN23 = SIGASSIGN INERTIAL clockstate WAVEFORM23
SIGASSIGN24 =
    SIGASSIGN INERTIAL (PORTNAME "clock_out") WAVEFORM23
SS25 = [SIGASSIGN23, SIGASSIGN24]
IFSTMT25 = IF_THEN_ELSE CONDITION19 SS22 SS25
SS26 = [IFSTMT25]
WAIT26 = (NAMESEVENT SIGLIST17)
PROCESS26 = MAKEPROCESS WAIT26 (SS26)
CONCURRENT27 = [PROCESS26]
ENTITYMEANING clock_ent = CONCURRENT27
```

Figure 2: Translation of VHDL clock.

but merely install the declaration of "output_type" in the internal symbol table of the translator.

Lines 5-12 of the example produce the following output.

```
||*************************************************************
|| Declaration of entity clock_ent.
DEFAULT_VALUE "clock_delay" = CONSTVAL (EXPCONSTANT (natVAL #1))
ENTITY ::= clock_ent |+
```

"clock_ent" is added to the enumerated type of all entities. "clock_delay" is added to the internal symbol table as a generic and its default value is defined. "clock_out" is added to the internal symbol table as a port name, but this generates no output from the translator.

Lines 14 and 15 generate the following:

```
||*************************************************************
|| Architecture clock_arch of clock_ent.
clockstate = SIGNALNAME [] "clockstate" ATOMIC
```

"clockstate" is defined as an abbreviation for the atomic signal name in the type NAME.

The sensitivity list on line 17 generates this line:

```
SIGLIST17 = [clockstate]
```

The condition from the if-then-else starting on line 19 generates:

```
CONDITION19 = (BINARY (EXPNAME clockstate)
                  EQUALOP (EXPCONSTANT (bitVAL bit0)))
```

The waveform, `'1' after clock_delay`, on line 20 is translated:

```
WAVEFORM20 = (WFEXP [<<(EXPNAME (GENNAME "clock_delay")),
                          (EXPCONSTANT (bitVAL bit1))>>])
```

Now the signal assignments on lines 20 and 21 are recognized and combined into a statement-sequence that is terminated by the "else" on line 22.

```
SIGASSIGN20 = SIGASSIGN INERTIAL clockstate WAVEFORM20
SIGASSIGN21 =
    SIGASSIGN INERTIAL (PORTNAME "clock_out") WAVEFORM20
SS22 = [SIGASSIGN20, SIGASSIGN21]
```

Note that the waveform on line 21 is identical to the waveform on line 20, so its abbreviation is reused (this is implemented via a hashing algorithm in the translator).

Similarly, lines 23-24 are translated as follows:

```
WAVEFORM23 = (WFEXP [<<(EXPNAME (GENNAME "clock_delay")),
                          (EXPCONSTANT (bitVAL bit0))>>])
SIGASSIGN23 = SIGASSIGN INERTIAL clockstate WAVEFORM23
```

```
SIGASSIGN24 =
    SIGASSIGN INERTIAL (PORTNAME "clock_out") WAVEFORM23
SS25 = [SIGASSIGN23, SIGASSIGN24]
```

The if-then-else ends on line 25 and is recognized.

```
IFSTMT25 = IF_THEN_ELSE CONDITION19 SS22 SS25
```

The body of the process ends at line 26. It is a statement-sequence containing the single if-then-else statement.

```
SS26 = [IFSTMT25]
```

Now, the process will be waiting for an event on its sensitivity list. The following expression evaluates to true exactly in this case.

```
WAIT26 = (NAMESEVENT SIGLIST17)
```

The process that ends at line 26, is defined with the appropriate wait-condition and body. The result, PROCESS26, has the type INSTANCE → PROCESS because the instance has not yet been supplied.

```
PROCESS26 = MAKEPROCESS WAIT26 (SS26)
```

The entity ends at line 27. Its body is a sequence of concurrent processes containing only the single process. The function ENTITYMEANING is defined by the translator to map ENTITY to [INSTANCE → PROCESS]. This make an entity into something that when supplied with an instance will be a list of processes.

```
CONCURRENT27 = [PROCESS26]
ENTITYMEANING clock_ent = CONCURRENT27
```

This example illustrates the translation of sequential statements and process statements. The main structuring mechanism in VHDL is the component instantiation statement, so we will continue our example to illustrate the translation of component instantiation.

Suppose we wish to build an entity that contains two clocks, one that ticks twice as fast as the other, and that has two output ports with the two clock outputs on them. Then the VHDL code in figure 3 appended to our previous example defines such an entity.

Again, we will examine the translation line by line. Lines 29-31 generate the declaration of "twoclock" as an entity.

```
ENTITY ::= twoclock |+
```

The constant declaration on line 34 defines "two".

```
two = CONSTVAL (EXPCONSTANT (natVAL #2))
```

```
29: entity twoclock is
30:    port (out1,out2 : out output_type);
31: end twoclock;

33: architecture foo of twoclock is
34:    constant two : TIME := 2 ns;
35: begin
36:    clock1 : clock_ent
37:             GENERIC MAP (clock_delay => two)
38:             PORT MAP (clock_out => out1);
39:    clock2 : clock_ent PORT MAP (clock_out => out2);
40: end foo;
```

Figure 3: An entity with two clocks.

The component instantiation that starts on line 36 will define an instance with a generic-map and a port map. The generic map is defined by the following:

```
MAP37 = [<<"clock_delay",(CONSTANT two)>>]
GENMAP37 = MAKE_GENMAP MAP37
```

The port map is recognized on line 38.

```
MAP38 = [<<"clock_out",(PORTNAME "out1")>>]
```

Now the label "clock1" is used to abbreviate the following partial instance. It has the type `INSTANCE→INSTANCE` because it has not been applied to its parent instance.

```
clock1 = Inst "clock1" MAP38 GENMAP37
```

Similarly, we form the partial instance for "clock2"

```
MAP39 = [<<"clock_out",(PORTNAME "out2")>>]
clock2 = Inst "clock2" MAP39 []
```

Now the entity ends at line 40 and the translation reads:

```
CONCURRENT40 =
        (MAP (\f->f.clock1)(ENTITYMEANING clock_ent))
        ++ (MAP (\f->f.clock2)(ENTITYMEANING clock_ent))
ENTITYMEANING twoclock = CONCURRENT40
```

The body of "twoclock" is the two component instantiation statements. The body is supposed to have the type: `[INSTANCE → PROCESS]`, that is, it should be a list of things which when supplied with an instance are processes. Now `(ENTITYMEANING clock_ent)` is already such a list. We have composed each member of this list with the partial instance "clock1" resulting in the list:

```
(MAP (\f->f.clock1)(ENTITYMEANING clock_ent))
```

Now a member of this list when supplied with a "parent" instance `X` will form the complete instance `(clock1 X)` and supply that instance to form the processes in "clock_ent". The second clock instance is formed in the same way, and the two lists have been appended.

Finally, how do we define a set of processes that we can actually simulate? One argument to the translator is the name of the top level entity. This is like telling an ADA compiler which library unit is the main program. For example, if the "testbed" entity for a design is called "test", and we call the translator with that name, we get the following lines of output:

```
INITLIST = MAP INIT (MAP_AT (ENTITYMEANING test) TOP)
STARTSTATE = Simulate <<STARTSIG,INITLIST,[#0]>>
```

The first line takes the meaning of the entity "test" and supplies it with the instance `TOP`. The result is a list of processes. Then the function `INIT` is applied to each process on the list. The definition of `INIT` is:

```
INIT (PROC W A INST) = PROC EXPTRUE A INST
```

This means that `(INIT P)` is a process identical to P except that it is waiting on "true". In VHDL, during the first simulation cycle, all the processes are active, and this is our way of achieving that behavior.

The second line:

```
STARTSTATE = Simulate <<STARTSIG,INITLIST,[#0]>>
```

then starts the simulation in an uninitialized starting signal-state, with the list of "activated" processes, and an δ-event scheduled.

In the next section we will give an example of symbolic simulation of a simple state machine.

4 Initial Results

In our initial experiments with our semantics, we used the example of a simple state machine given in figure 4. The example is taken from Dennis Morton's thesis, [7], which we got from Draper Labs and which contains a partial VHDL design of the Scoreboard. The example defines a machine with inputs "clock" and "control" (as well as "reset"). It has an internal signal "state" which can take one of four values. When the clock is active, then as a function of the control and the internal state, a next state and some output values are computed.

In order to simulate this example, we had to provide the "testbed" in figure 6. In this testbed, we are driving the clock and control wires with instances of the clock-entity used in the previous examples. The clock connected to the control-wire ticks at half the rate of the clock connected to the clock-wire.

```
package state_machine_package is
        type state_type is (s0,s1,s2,s3);
        subtype control_type is BIT;
        subtype output_type is BIT;

        constant clock_active : control_type;
        constant control_active : control_type;
        constant output_active : output_type;
end state_machine_package;

package body state_machine_package is
        constant clock_active : control_type := '1';
        constant control_active : control_type := '1';
        constant output_active : output_type := '1';
end state_machine_package;

entity state_machine is
        generic ( output_delay : TIME := 1 ns;
                  state_delay : TIME := 1 ns);
        port ( control,reset : in control_type;
               clock : in control_type;
               out1,out2 : out output_type);
end state_machine;
```

Figure 4: Declaration of a simple VHDL state machine.

```
architecture best of state_machine is
        signal state : state_type;
begin
    machine : process (clock,reset)
        variable next_state : state_type;
    begin
        if reset = control_active then
            next_state := s0;
        elsif clock = clock_active and clock'event then
        case state is
        when s0 =>
            out1 <= not output_active after output_delay;
            out2 <= not output_active after output_delay;
            if control = control_active then
                next_state := s1;
                out1 <= output_active after output_delay;
                out2 <= not output_active after output_delay;
            end if;

        when s1 =>
            if control = control_active then
                next_state := s2;
                out1 <= not output_active after output_delay;
                out2 <= output_active after output_delay;
            end if;

        when s2 =>
            if control = control_active then
                next_state := s1;
                out1 <= output_active after output_delay;
                out2 <= not output_active after output_delay;
            else
                next_state := s0;
                out1 <= not output_active after output_delay;
                out2 <= not output_active after output_delay;
            end if;
        when others =>
            next_state := s0;
        end case;

    state <= next_state after state_delay;
    end if;
    end process;
end best;
```


Figure 5: Body of the VHDL state machine.

```
entity test is end test;

architecture foo of test is
      signal clock_wire : output_type;
        signal control_wire : control_type;
        signal reset_wire : control_type;
        signal out1_wire,out2_wire : output_type;
        constant two : TIME := 2 ns;
begin
    control_inst : clock_ent
            GENERIC MAP (clock_delay => two)
            PORT MAP (clock_out => control_wire);
    clock_inst : clock_ent PORT MAP (clock_out => clock_wire);
    machine_inst : state_machine PORT MAP (
                    control => control_wire,
                    reset => reset_wire,
                    clock => clock_wire,
                    out1 => out1_wire,
                    out2 => out2_wire);
end foo;
```

Figure 6: The testbed.

We can trace the values of a list of expressions over a simulation with the help of the following definitions (which are loaded along with the semantic definitions).

```
TRACE LST = _TRACE LST TOP STARTSTATE

_TRACE LST INST S  =
    DUMP LST INST S : _TRACE LST INST (Simulate S)

DUMP [] INST S = []
DUMP (A:B) INST S = (_EVAL INST S A): DUMP B INST S

_EVAL INST <<S1,Q,T>> = EVAL INST S1
```

The expression (TRACE LST) reduces to an infinite list of "dumps". The i^{th} "dump" consists of the values of the expressions in LST evaluated in the state S_i where S_0 is STARTSTATE and $S_{i+1} = SimulateS_i$.

Now, after translating the state-machine example into Caliban using the translator, we load the resulting definitions into Clio. Then if, for example, we want to trace the values of the control wire and the internal state of the machine, as well as the global clock, we can load the following definitions.

```
cw = EXPNAME (SIGNALNAME [] "control_wire" ATOMIC)
st = EXPNAME (SIGNALNAME ["machine_inst"] "state" ATOMIC)
```

We then ask Clio to reduce the expression (TRACE [CLOCK, cw,st]), and get the following output (in a matter of minutes):

```
[natVAL Zero, UNINITIALIZED, UNINITIALIZED]:
[natVAL (#1), UNINITIALIZED, UNINITIALIZED]:
[two, bitVAL bit0, UNINITIALIZED]:
[natVAL (#3), bitVAL bit0, UNINITIALIZED]:
[natVAL (#4), output_active, UNINITIALIZED]:
[natVAL (#5), output_active, state_typeVAL s0]:
[natVAL (#6), bitVAL bit0, state_typeVAL s0]:
[natVAL (#7), bitVAL bit0, state_typeVAL s0]:
[natVAL (#8), output_active, state_typeVAL s0]:
[natVAL (#9), output_active, state_typeVAL s0]:
[natVAL (#10), bitVAL bit0, state_typeVAL s0]:
[natVAL (#11), bitVAL bit0, state_typeVAL s1]:
[natVAL (#12), output_active, state_typeVAL s1]:
[natVAL (#13), output_active, state_typeVAL s1]:
[natVAL (#14), bitVAL bit0, state_typeVAL s1]:
[natVAL (#15), bitVAL bit0, state_typeVAL s2]:
[natVAL (#16), output_active, state_typeVAL s2]:
...
```

(Since "output_active" was defined as `bitVAL bit1` and "two" was defined as `natVAL (#2)`, Clio is using them as abbreviations in displaying the results.)

We see that the values become initialized, and that the control wire is "ticking" every two cycles. The user defined enumerated type has been added to the VALUE type by the following lines from the translation:

```
state_type ::= s0 | s1 | s2 | s3
VALUE ::= state_typeVAL state_type |+
```

We see the internal state of the state-machine changing correctly.

These initial results were encouraging, for, at least in a simple example, the operational semantics seem usable.

5 Application to Scoreboard

The Scoreboard circuit design was given to us as a collection of eleven VHDL files describing major blocks and averaging 504 lines of VHDL each. In addition a "shell" file of 4593 lines of VHDL connected the major blocks into the complete Scoreboard. Further, two packages, totaling 2791 lines of VHDL, contained library functions on the seven-valued lattice of bits and other predefined functions used by the Synopsis synthesis tools.

Our plan was to characterize the behavior of each major block at a level of abstraction similar to the level of abstraction at which components are described when using Spectool. Spectool, [10], is the hardware description tool, developed at ORA, which we used to describe and verify the simplified Scoreboard design in our previous work [12]. The overall structure of the VHDL design of the Scoreboard does not differ much from the Spectool design we verified. Therefore, once we had a "Spectool-like" description of the behavior of each major block, we should be able to modify and reuse most of our earlier work to create the proof of the VHDL design.

We began our work on the actual Scoreboard design by completing the translator until it could translate all of the major blocks, shell, and Synopsis packages into Caliban without any modification and result in translations that could be read (viz. abbreviations were used to keep expressions short). After this had been accomplished, we began the process of characterizing the behavior of each major block, beginning with the "voterblock". This block was given as 717 lines of VHDL. We hoped to develop a "push-button" method to extract the abstract behavior of a block such as the voterblock from the translation of the VHDL code. Once this method was working on the voterblock, it could be applied to all the other blocks and we could begin modifying our earlier proof. As skilled users of our system, we could have begun by writing the abstract behavioral description of each block by hand and then using Clio to prove that the translation of the VHDL implemented the abstraction. However, since our goal was more ambitious, namely, to develop a general method for verifying VHDL designs

that would be usable by less skilled users, we did not pursue this approach. As a result, we have not verified the VHDL Scoreboard design, because we have not yet been able to develop a simple method to extract the abstract behavior of a block from the VHDL. We will next describe the level of abstraction we are attempting to extract from the VHDL code and the method we used, unsuccessfully, to extract it. We then discuss the features of the VHDL operational model that caused this method to fail, and finally describe the methods that we feel will overcome these problems.

6 Abstraction

We will briefly discuss the level of abstraction at which components are described in Spectool, and illustrate the discussion by considering a simple latch. We will then describe the same latch in VHDL and see what abstraction is needed.

In Spectool, a component has an internal state, a set of input ports, a set of output ports, and supports a set of *actions*. Each action is characterized by three things:

1. A function from (state x input values) to next state.

2. A list of functions from (state x input values) to output values, one for each output port.

3. A delay.

In the Spectool model, each component has two implicit input ports, one connected to the CLOCK, and one connected to the CONTROLLER. In operation a component acts as follows. When the clock ticks, each component reads its control input and, if the control input is an action (as opposed to "no-op"), it changes its state and outputs by applying the appropriate functions for that action to the current values of its input ports. The changes are made with the delay given for that action, and in the meantime the internal state and outputs become undefined.

For example, a simple latch is described by the following information. The type "data" is a type parameter.

- internal state: latchstate :: data
- inputs : latchin :: data
- outputs : latchout :: data
- actions: set
- nextstate set s in = in
- nextout set s in = in
- delay set = 1

To design a latch in VHDL with the same behavior (see figure 7) we must make the clock and control inputs explicit. Also, since we do not have a simple data abstraction mechanism (such as private types), we must declare the type "data" to be something. For the example we chose an enumerated type with eight values.

Now, we can make a "testbed" entity to drive the clock, control, and latchin input ports of the latch and then run the simulation as we did for the state-machine example. However, in order to characterize the behavior of the latch, we must prove a theorem that describes the behavior when the latch is part of an arbitrary state that satisfies some minimal assumptions.

In formulating the minimal set of assumptions about the context in which the latch will be embedded, we find that many assumptions that are implicit in the Spectool operational model must be stated explicitly for the VHDL operational model. We defined the following "generic" assumption list that captures most of these assumptions.

```
Ready 's' 'inst' 'X' :==
  PROPER 's'
  & SINGLE_PORT_CONNECTIONS 'inst'
  & 'CLOCKNAME inst'='SIGNALNAME X "clock" ATOMIC'
  & 'X'~='INST_NAME inst'
  & CLOCK_TICK 's' 'inst'
  & ALL_INTERNAL_QUIET 's' 'inst'
  & DEFAULT_VALUE_TOTAL
  & FUNCMEANING_TOTAL
  & '!!inst'='True' & '!!X'='True'
```

The `PROPER` predicate was previously defined and says that names of signals get signal-values and names of variables get simple values. The predicate `SINGLE_PORT_CONNECTIONS` says that the given instance does not connect two

```
package latch_package is
        type action is (set,noop);
        type data is (x1,x2,x3,x4,x5,x6,x7,x8);
end latch_package;

entity latch is
       port ( latchin: in data;
              clk: in bit;
              control: in action;
              latchout: out data
            );
end latch;

architecture latch of latch is
signal latchstate: data;
begin
latch: process
       begin
            wait until (clk='1') and (control=set);
            latchstate <= latchin after 1 ns;
       end process;
latchout <= latchstate ;
end latch;
```

Figure 7: VHDL latch.

ports to the same external signal. This is certainly not required by VHDL, but is satisfied by most designs. In Spectool this restriction is imposed only on output ports; in fact, no two output ports even of possibly distinct components may be connected to the same signal because Spectool does not allow multiple drivers of a signal. The next three assumptions say that the input port "clk" is, in the given instance, connected to some signal called "clock" which is external to the component (so its instance name `X` differs from the instance name of the component) and which has a driver that is a `CLOCK_TICK`. A `CLOCK_TICK` waveform is one that flips between bits '1' and '0' with unit delay. The predicate `ALL_INTERNAL_QUIET` is an invariant that says that no signal internal to the component has any pending transactions on its driver at the current simulation cycle. Finally, we must assume that all functions have well-defined bodies and all generics have well-defined default values and that the given instance and the instance of the "clock" are well-defined. All of the assumptions included in the `Ready` predicate are either implicit in the Spectool model or are part of a global invariant called `Proper_state` that Spectool generates.

Now to extract the abstract behavior of the latch, we simulate an arbitrary instance `@inst` of it in an arbitrary state `@s` satisfying the `Ready` assumptions (for some otherwise arbitrary @X). We run the simulation over one clock cycle; we take the clock cycle to include the simulation cycle in which the clock goes low (bit '0') and the subsequent cycles until the clock has gone high (bit '1') and will go low in the next cycle. Note that this may include many more than the two simulation cycles during which the clock goes low and goes high, because there may be many signal assignments with zero delay that cause extra δ-transitions. In the latch example this does not happen, however, so a clock cycle is two simulation cycles, and each simulation cycle may be thought of as a single clock *phase*. Since the latch has a unit delay, it has a delay of one clock phase. Spectool also allows a clock cycle to be divided into phases. So the VHDL latch corresponds to the Spectool latch if the number of phases in the Spectool model is two per cycle.

In order to see the full behavior of the VHDL latch, we must simulate the arbitrary instance for one and a half cycles to account for the one phase delay. We then abstract the state by extracting only the values of the internal state and output ports.

To give a bit more detail, we define the starting state to consist of an arbitrary signal state, the processes that constitute the body of the latch instance, and a list of scheduled clock ticks:

```
StartLatch s inst =
    <<s,ENTITY_PROCS latch inst,[#1,#2,#3,#4,#5]>>
```

We define the abstraction function to take the current values of the state and output:

```
LatchAbs inst <<s,ps,ts>> =
```

```
MAP (CURRENT inst s) [latchstate, PORTNAME "latchout"]
```

We make the **Ready** assumption:

```
Ready '@s' '@inst' '@X'
```

And then reduce the expression:

```
LatchAbs @inst (Iterate #3 Simulate (StartLatch @s @inst))
```

Here is the resulting expression for the internal state:

```
((CURRENTVAL (ADVANCE (#2) @s)
      (PORTCONNECT @inst (PORTNAME ("control"))))
  = (actionVAL set)) ->
 (CURRENTVAL (ADVANCE (#2) @s)
      (PORTCONNECT @inst (PORTNAME ("latchin")))
  );
 CURRENTVAL @s (REALNAME @inst @s latchstate)
```

Since our clock cycle starts in the clock high phase, after one unit of time has elapsed the clock goes low, then after two units of time, the clock goes high. At this point, if the value of the signal connected to "control" is "set", then the value of the signal connected to "latchin" will become the new state (after one more phase). Otherwise the new state is whatever it was in the initial state **@s**. Clio makes the above process easy and also saves the result as a theorem when given the following command:

```
prove
  ? = 'LatchAbs inst (Iterate #3 Simulate (StartLatch s inst))',
        Ready 's' ' inst' 'X'
```

Thus, to summarize, our abstraction method was:

1. Define an abstraction function that extracts the values of internal signals and output ports.
2. Assume the generic **Ready** predicate on arbitrary state and instance.
3. Simulate the processes of the entity over one clock cycle and apply the abstraction function to the resulting state.
4. Express the results as a theorem.

The method just presented seems complex, however, it is a standardized procedure and the steps are well supported by Clio. Thus we can make it into a semi-automatic "cook-book" procedure, or, by extending the translator to have it automatically define the abstraction function and necessary Clio commands, make it a fully automatic, "push-button" procedure. Unfortunately, even though this general procedure works (to some extent) for this simple latch, it suffers from some severe drawbacks which make it ineffective for the components of the Scoreboard. We discuss these problems in the next section.

7 Problems

There is a flaw in the abstraction method used on the simple latch example defined in the previous section. The theorem we derive says that the latch, *when executed as the only active entity* in an arbitrary state (satisfying the **Ready** condition) will exhibit the given abstract behavior. The theorem we want would say that the latch processes, when executed in combination with an arbitrary set of additional process, in an arbitrary **Ready** state will exhibit the correct abstract behavior. In fact, some *non-interference* condition on the additional processes will, in general, be necessary to guarantee correct behavior. For example, we might require that no other process may drive the signal to which the "latchout" is connected. But, even after assuming such a non-interference condition, our method will not work.

The problem is that an arbitrary set of additional processes, even non-interfering ones, can generate an arbitrary number of δ-delays and hence an arbitrary number of simulation cycles during one clock cycle. Now, we may argue that since the additional processes are non-interfering, the additional δ-delays have no effect on the signals which are part of the abstract behavior of the latch. This argument is correct, but it must be formalized in order to make a proof.

We may choose to ignore this problem by reasoning that if we derive the correct abstract behavior of a component in isolation, then we can correctly use this behavior at the more abstract level supported by Spectool when we connect the components together. The justification for this would be a "meta-theorem" that a certain non-interference condition is sufficient. This "meta-theorem" could be proved inside or outside the system. It would be most desirable to have a non-interference condition that could be checked syntactically. Then the user could derive an abstract behavior for each entity, and the system could make the non-interference check. Then the user can proceed to derive and verify the behavior of the composite system as if it had been described in Spectool (or any other abstract HDL). If the non-interference condition permits standard design styles to pass, then this will still be a useful verification style.

Proceeding on the assumption that the desired non-interference condition and "meta-theorem" could be found, we continued deriving the abstract behavior of the first major component of the Scoreboard, namely the "voterblock". Even in isolation, that is, when simulating only the voterblock processes, in an arbitrary state, we have a problem with δ-delays. The basic structure of the voterblock (and all of the other components of the Scoreboard) is a pipeline which is written schematically in VHDL in figure 8.

The schematic pipeline consists of five processes. The first responds with a δ-delay to any change in the `inport`, and copies to `inreg` (actually several one-bit in ports are concatenated to form an aggregate). The second stage latches the `inreg` into the `inbuf` when the clock goes high. The third stage is sensitive to any change in the `inbuf` and computes the `outbuf` after another δ-delay. The

```
port ( inport in t1; outport out t2);
signal inreg, inbuf, outbuf, outreg;
begin
    inreg <= inport;
    inlatch:process
            begin
                wait until clk='1';
                inbuf <= inreg;
            end process;
    compute: process(inbuf)
             begin
                outbuf <= answers;
             end process;
    outlatch:process
             begin
                wait until clk='1';
                outreg <= outbuf;
             end process;
    outport <= outreg;
end
```

Figure 8: Schematic pipeline.

fourth stage moves the **outbuf** into the **outreg** when clock is high. The final stage responds to any change in **outreg** and updates the **outport** after another δ-delay.

Now, even simulating this pipeline as the only collection of active processes, in an arbitrary state, we have the following problem. The signal connected to **inport** may have an arbitrary waveform in its driver (or drivers). As we simulate the pipeline, this waveform may cause the **inport** to change, and this introduces another δ-delay due to the first stage of the pipeline. Because the computation of **outbuf** is based on **inbuf**, and **inbuf** is clocked, the only change to **inport** that affects the results of the computation is the change that happened just prior to **clk = '1'**, but, again, this is another "meta-theorem". In fact, the instance of this "meta-theorem" that we need can be derived by our simulation method by considering cases. Since the pipeline is short, only a few simulation cycles other than δ-cycles are needed. The waveform on the **inport** driver cannot change in two consecutive δ-cycles. So, there is a bound on the number of changes it can make. Even if this bound is as low as two, this still leads to four simulation histories since it can either change or not change at two points during the simulation, generating or not generating an extra δ-delay. Of course, the extra δ-delays are easily seen not to affect the final behavior, so perhaps checking all the cases will not be too inefficient. This is wishful thinking, as we have seen in practice. In fact, Clio generates all the cases as part of a large conditional expression. Unfortunately, the "voterblock" has not one but thirty-eight input ports, and the possible changes in the drivers of these ports cause an exponential explosion of cases which Clio cannot handle.

If we rely on the second "meta-theorem" that says that multiple events on the ports are irrelevant and only the event just prior to the rising clock is relevant, we may simplify the pipeline by combining the first two stages, deleting the signal **inreg** and assigning **inport** (or, actually, a concatenation of in ports) to **inbuf** directly.

```
inlatch:process
        begin
            wait until clk='1';
            inbuf <= inport;
        end process;
```

With this modification, we should have an equivalent VHDL entity.

Will we still have problems with δ-delays? Based on our experience with the "voterblock" we will not. In our first efforts to obtain the abstract behavior of the voterblock, we wanted to isolate what we felt were the key processes. These were the compute and output stages of the pipeline, and particularly the compute stage, which contained nested for-loops, case statements, and if-then-else statements. Therefore we commented out the signal assignment statements which constituted the first stage of the pipeline. This meant that the final values on the output ports would be functions of the values that were initially

in the registers corresponding to `inreg` rather than functions of the values on the ports corresponding to `inport`. Once we were able to derive the outputs as a function of the stage one registers, `inreg`, we thought it would be easy to put the first stage of the pipeline back in place, since it consists of simple unclocked, unconditional signal assignments of the form `inreg <= port0&port1...&port7`, and these assignments are essentially invariants.

Having removed the first stage of the pipeline, we were indeed able to derive the behavior of the voterblock. Modifications to the semantics had to be made, including rewriting them to use conditional function updates, in order to handle the complicated compute stage of the pipeline. Once we accomplished this much, with very little time left to complete our work, we were disappointed to discover a major problem with what we thought would be an easy step. At this point we could have modified the VHDL code we were given to combine the first two stages of each pipeline as described above. Then, given more time than we had left, we could have completed the verification. However, it would have been an unsatisfying accomplishment since the final proof would have contained two major gaps corresponding to the two unproved "meta-theorems" we relied upon. Therefore, we used the time remaining to understand the problems and to propose some solutions to them, which we discuss in the next section.

8 Proposed Solutions

The problems we encountered were caused by two things. The first is the lack of abstraction with respect to δ-delays. Since our basic state transition function, `Simulate`, exposes the δ-delays, properties of *runs* that we assert may not be invariant under the insertion of additional δ-transitions. Even if the property of a run is invariant in this sense, for example, a property that relates the state at one rising clock edge to the state at the next rising clock edge, the proof by symbolic execution of a fixed number of iterations of `Simulate` will not be.

The second cause is the lack of *composability* or *monotonicity* of the properties and their proofs. A property proved about a process in isolation, may not hold when the process is combined with other processes. This second cause is related to the first, for under suitable non-interference conditions the only way that additional processes can affect the behavior of a given process is by generating extra δ-transitions.

To address the first cause, we would like to state all requirements in a δ-invariant way. A simple way to do that is to introduce the notions of a *completed state* and the *distance* between two states. Recall that in our semantics, the basic state was defined:

```
type STATE = <<SIGSTATE,[PROCESS],[TIME]>>
```

The third component is a list of times at which a simulation cycle will occur. In particular, if the head of this list is zero, then the next simulation cycle is

a δ-transition. If the head of the list is not zero, then all δ-transitions for this point in the run have been completed. Let us call such a state a *completed state.* In Caliban:

```
COMPLETED <<S,P,T>> = (hd T ~= Zero)
```

As a special case, we will also call the first, uninitialized state completed.

Next recall that there is an expression which we abbreviated `CLOCK` whose value in any state is the elapsed time on the global clock since the start of the run. If S is a state, let t_S be the value of this expression `CLOCK` in state S. Then define the distance between two states, S_1 and S_2 to be the difference $dist(S_1, S_2) = t_{S_2} - t_{S_1}$. In Caliban:

```
DIST S1 S2 = (CLOCKVAL S2) #- (CLOCKVAL S1)
CLOCKVAL <<S,P,T>> = EVAL TOP S CLOCK
```

Since a δ-transition does not advance the global clock, the distance between two states is not affected by the addition of such transitions.

If S is a state and x is a name, let S_x stand for the current value of x in state S. Consider a single concurrent signal assignment statement:

```
x <= y ;
```

This translates to a single process, `p`, in our semantics. In any run starting from a state S_0 in which `p` is waiting and no other process ever drives the signal x, we can see that $S_x = S_y$ in any completed state S in the run. Informally, the proof is that initially $S_x = S_y$, and if y does not change then neither will x since no other process drives x, and if y does change, then the state is not completed because `p` becomes active and generates a δ-transition after which the values of x and y are equal. The formal proof would be by induction on the position of S in the run.

If we let $N(x)$ stand for the requirement that a process does not drive x, and we let $EQ(x, y)$ stand for the assertion just proved, that x and y are equal in all completed states of a run, then we have proved a judgment of the form:

$$p, N(x) \models EQ(x, y)$$

The meaning of this judgement is that any run of any collection of processes which includes p will satisfy $E(x, y)$ provided every process in the collection except p obeys the restriction $N(x)$. This type of judgement is by it very definition *monotone* under the addition of processes that satisfy the given non-interference condition $N(x)$. We might call $EQ(x, y)$ and *invariant equation.*

Consider next the assignment:

```
x <= y after 2 ns;
```

If this is translated as process q, and we impose the non-interference condition $N(x)$ on the set of additional processes, then we can prove that in any run, if S^1 and S^2 are completed states and the distance $dist(S^1, S^2)$ between them is $2ns$, then $S^2_x = S^1_y$. If we call this assertion $DEQ(x, y, 2)$ then we have the judgement:

$$q, N(x) \models DEQ(x, y, 2)$$

We can call $DEQ(x, y, n)$ a *delayed invariant equation.*

Such judgements can be combined, if we have:

$$(p, A \models \phi) \wedge (q, B \models \psi)$$

and p satisfies the non-interference condition B and q satisfies the condition A, then:

$$[p, q], [A, B] \models \phi \wedge \psi$$

In the VHDL verification system we propose, the judgements for simple processes like the concurrent signal assignments could be generated automatically and added as lemmas. Also non-interference conditions of the form "process p does not drive signal x" can be checked syntactically and added as lemmas. In the "voterblock" of the Scoreboard, there were thirty-two processes. Of these, twenty-eight were simple concurrent signal assignments that could be handled automatically. That leaves only four processes where computations were actually performed. We would like to be able to use the operational semantics to prove a new judgement for a process that the system cannot generate automatically.

Suppose that process p is such a process that reads from signals x and y and writes to signal z. If we simulate it in isolation on an arbitrary *completed* state S^1 for some number of iterations of `Simulate` and reach a completed state S^2 in which S^2_z, the value of z in S^2, is some function, $f(S^1_x, S^1_y)$ of the values of x and y in state S^1. Then, in some sense, we have proved the delayed invariant $DEQ(z, f(x, y), d)$ where $d = dist(S^1, S^2)$. If $L = [a, b, c, z]$ are all the signals other that the input signals x and y that are read or written by p during the simulation from S^1 to S^2, then our "meta-theorem" mentioned earlier says that as long as no other process interferes with any signal in L, then p together with these non-interfering processes will guarantee the delayed equation $DEQ(z, f(x, y), d)$. Thus, we should have proved the judgement:

$$p, N(L) \models DEQ(z, f(x, y), d)$$

For this kind of proof to be made rigorous, more work is needed, however the system we propose would have the advantages of both kinds of semantics, the operational and the logic based. The user would typically work at the higher level of abstraction provided by the judgements, but would be able to prove new judgements directly from the operational semantics when needed. In summary, we think that reasoning about processes and sets of processes by

proving judgements about them that can be composed under non-interference restrictions is the best way to verify complex VHDL designs. The operational model we have created gives a precise definition to the set of processes in a VHDL design, provides a meaning for the judgements, and provides a way to prove new judgements. Thus, although we could not use our operational semantics to verify the Scoreboard, we think it is valuable.

9 Conclusions

As a method for re-verifying the Scoreboard design, the approach we took, of creating a general method for verifying VHDL designs, was too ambitious and we did not complete the verification. If we had translated the VHDL by hand into the Spectool model, we would have completed the verification, but the connection between the Spectool model and the VHDL code would have been informal and there would have been a gap in the proof.

In the end, we think that the work we did is more valuable than if we had merely verified a particular design by ad hoc methods. Although we were discouraged by the problems we had in applying our methods to the Scoreboard, we now think that the approach proposed in the previous section, combining the logic-based and operational methods, is very promising. At present, the most impressive results in hardware verification have been on devices with a central clock in which all the components operate in lock step. No easily used methods have been developed to handle systems with interacting, independently clocked components such as a system containing a microprocessor, memory management unit, and co-processors. Such systems can be conveniently described in VHDL and the proposed verification system seems like a good way to reason about them.

As explained in the previous section, the operational semantics we defined will be used as the basis of the proposed system. Also, without the experience gained in trying to use a pure operational approach to verify a large VHDL design, we would not have been led to the proposed, mixed logic-based and operational method. We have also exposed, in a real example, the pitfalls that other formal methods practitioners must address, so we think that this report will contribute to the state of the art.

References

[1] William R. Bevier and William D. Young. "Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit". NASA Contractor Report 182099, November 1990.

[2] M. Bickford, C. Mills, and E.A. Schneider. "Clio: An Applicative Language-Based Verification System". Technical Report TR 89-13, ORA

Corporation, 301A Harris B. Dates Drive, Ithaca, NY 14850, September 1989.

[3] Boulton, Gordon, Gordon, Harrison, Herbert, and Van Tassel. "Experience with embedding hardware description languages in HOL". Technical report, Computer Laboratory, University of Cambridge, Cambridge, UK, 1992.

[4] Ricky W. Butler. "NASA Langley's Research Program in Formal Methods". In *Proceedings of the Sixth Annual Conference on Computer Assurance*, pages 157–162, June 1991.

[5] Avra Cohn. "Correctness Properties of the Viper Block Model: The Second Level". Technical Report 134, Computer Laboratory, University of Cambridge, Cambridge, U.K., May 1988.

[6] Warren A. Hunt. "FM8501: A Verified Microprocessor". In *IFIP WG 10.2 Workshop, From HDL to Guaranteed Correct Circuit Designs*, pages 85–114. North-Holland Publishing Co., 1986.

[7] Dennis Morton. "Hardware Modeling and Top-down Design Using VHDL". Technical Report CSDL-T-1082, The Charles Stark Draper Laboratory, 555 Technology Square, Cambridge, MA 02139, June 1991. MS Thesis, MIT, Cambridge, MA 02139.

[8] Natarajan Shankar. "Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm". NASA Contractor Report 4386, July 1991.

[9] Mandayam Srivas and Mark Bickford. "Formal Verification of a Pipelined Microprocessor". *IEEE Software*, September 1990.

[10] Mandayam Srivas and Mark Bickford. "Spectool: A Computer-Aided Verification Tool for Hardware Designs, Final Report, Contract No. F30602-89-D-0096". Technical report, Rome Laboratory, Griffis AFB, NY 13441, 1991. Authors' affiliation: ORA Corporation, 301A Dates Drive, Ithaca, NY 14850.

[11] Mandayam Srivas and Mark Bickford. "Verification of the FtCayuga Fault-Tolerant Microprocessor System Volume 1: A Case Study in Theorem Prover-Based Verification". NASA Contractor Report 4381, 1991.

[12] Mandayam Srivas and Mark Bickford. "Moving Formal Methods into Practice: Verifying the FTPP Scoreboard Phase 1 Results". NASA Contractor Report 189607, May 1992.

[13] J. P. Van Tassel. "A Formalization of the VHDL Simulation Cycle". Technical Report 249, Computer Laboratory, University of Cambridge, Cambridge, UK, 1992.

[14] John Van Tassel and David Hemmendinger. "Toward Formal Verification of VHDL Specifications". In *Luc Claesen, ed, Applied Formal Methods For Correct VLSI Design*, Leuven, Belgium, November 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 5, 1994	Contractor Report Jan 92 - Dec 92

4. TITLE AND SUBTITLE
Formal Semantics For a Subset of VHDL and its use in analysis of the FTPP Scoreboard circuit

5. FUNDING NUMBERS
C-NAS1-18972
TA-6
WU 505-64-10-13

6. AUTHOR(S)
Mark Bickford

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326

8. PERFORMING ORGANIZATION REPORT NUMBER
TM-92-0045

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191577

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: James L. Caldwell

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category: 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In the first part of the report, we give a detailed description of an operational semantics for a large subset of VHDL, the VHSIC Hardware Description Language. The semantics is written in the functional language Caliban, similar to Haskell, used by the theorem prover Clio. We also describe a translator from VHDL into Caliban semantics and give some examples of its use. In the second part of the report, we describe our experience in using the VHDL semantics to try to verify a large VHDL design. We were not able to complete the verification due to certain complexities of VHDL which we discuss. We propose a VHDL verification method that addresses the problems we encountered but which builds on the operational semantics described in the first part of the report.

14. SUBJECT TERMS
VHDL, Clio

15. NUMBER OF PAGES
67

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	