

# Examination of a CSP Style Formal Definition of VHDL

David L. Barton  
Intermetrics, Inc.  
7918 Jones Branch Dr.  
McLean, VA 22102  
(703) 827-2606  
dlb@wash.inmet.com

April 5, 1995

## Abstract

Various mechanisms have been investigated for producing a formal definition of VHDL. This paper will explore yet another: Communicating Sequential Processes, or CSP. This formalism has some advantages, both for VHDL in general, and in defining the relationship between a VHDL model, a WAVES dataset that specifies the test vectors for the model, and a tester specification.

## 1 Introduction

In the past few years, there has been a tremendous amount of investigation concerning a formal semantics for VHDL. Various approaches have been tried, including a denotational approach [3, 10, 11], operational [4], functional [1], and net based approaches [2]. Both dynamic and static [12] semantic definitions have been investigated.

This paper explores another approach: a formal definition of VHDL based upon the Communicating Sequential Processes, or CSP, formalism. This formalism was developed by C.A.R. Hoare during the 1980's. The defining work for CSP is [6]. CSP is normally associated with Occam<sup>tm</sup> and communicating

between processes using the send (!) and receive (?) operations. In fact, communication is built on top of a complete and workable formalism for concurrent processes within CSP. It is to this basic formalism that we will look as a source for a formal semantics of VHDL.

As will be explained below, CSP is defined in terms of the traces — the observable behavior defined as a sequence of events — of the processes of a description. This is ideal for VHDL; the semantics of VHDL is codified in the values of its signals, and any implementation that gives equivalent signal values is appropriate. Given that a signal is a series of time / value pairs, and that each pair can be represented as an event (as can process activation), we have most of the mechanism we need to define the dynamic semantics of VHDL.

Indeed, we have more than that. WAVES, the Waveform and Vector Exchange Specification [7], is defined in terms of traces as well. This allows us to specify precisely, not just a single simulation, but all possible simulations of a VHDL model against any waveform that meets a WAVES specification. As will be shown, this specification can be expressed using operations defined at present within CSP.

Moreover, if a tester is described with a CSP speci-

fication, then all possible waveforms that can run on the tester and that meet the WAVES specification of the waveform may be similarly expressed. This means that the waveform translation problem has been formally specified, and reduces to choosing one of the vectors that satisfy the combined specification. This will be explored further below.

This paper will examine the dynamic semantics of VHDL. We anticipate no real contribution for CSP in the area of a static semantics (although it would not be impossible; see [6] for a discussion of defining languages using CSP). Those interested in a static semantic definition should refer to [12].

The structure of this paper is as follows. We assume that the reader is familiar with both WAVES [7] and with VHDL [8]. A brief survey of CSP will be given in the next section. This will be followed by a discussion of representing the basic constituents of a VHDL simulation in CSP. Special attention will be given to processes, and specifically to the sequential part of processes (which is skipped in most other treatments). Following this, we will discuss the relationship between a CSP specification of a VHDL description, a WAVES dataset, and a tester. We then discuss how a formal semantics for VHDL will be expressed in terms of CSP. Finally, we conclude and give some indications for future research.

## 2 A Brief Survey of CSP

The material in this section is taken from the standard text by Hoare [6]. The unit of action in CSP is the process. A VHDL process has a corresponding CSP process; however, the CSP process is also considerably finer grained, and many individual VHDL constructs will also correspond to a CSP process, right down to a variable assignment statement.

The action of a process is observed by the outside world as a series of events; indeed, *anything* that may be observed by the outside world in a process is an event by definition. Each process has a fixed (possibly

infinite) set of events in which it can engage. This set is called the alphabet of the process. The selection of the alphabet is extremely important, not just for the sake of specifying what the process can do, but also for the sake of specifying the coordination between different processes.

Because the event is the unit of process observation in CSP, the processes are defined in terms of the events in which they can engage. For example, the process that engages in event  $a$  and then acts like process  $P$  is expressed as follows:

$$(a \rightarrow P)$$

This is pronounced “ $a$  then  $P$ ”; the  $\rightarrow$  symbol identifies the prefix operation, in the sense that  $a$  is the prefix of the process. The alphabet of this process is whatever the alphabet of  $P$  is plus the symbol of  $a$ ; symbolically,  $\alpha P \cup \{a\}$ .

Processes can also be defined recursively, via normal recursive style equations. For example, we can define an equation that gives the femtosecond ticking of the VHDL simulation clock:

$$CLOCK = (fs \rightarrow CLOCK)$$

which means, intuitively, that  $CLOCK$  is the process that accepts an femtosecond tick, then acts like the process  $CLOCK$  again. This simple process definition defines the VHDL clock, minus delta cycles. To introduce delta cycles, we need to introduce the operation of choice. A true VHDL clock may either tick a delta cycle or a femtosecond advance of simulation time. It therefore has to make a choice. This is expressed as follows:

$$CLOCK = (fs \rightarrow CLOCK \mid d \rightarrow CLOCK)$$

where  $d$  is the event corresponding to a delta cycle. This process statement fully defines the VHDL clock. (In fact, we need to introduce a constraint that states that there is *some* process activation between delta cycles; this will be dealt with later). This recursion can be extended to a mutually recursive definition between an arbitrary number of processes if necessary.

The choice operation extends to an arbitrary number of choices. Each choice can be followed by a different process specification, if desired. Completely generally, if  $B$  is a set of events, and  $P$  is a set of processes, where each element of  $B$  can be used to select one of the processes in  $B$ , then the general choice operation is expressed as:

$$(x : B \rightarrow P(x))$$

The  $x$  in the above statement is bound within (local to) the general choice operation. The above statement is equivalent to:

$$(y : B \rightarrow P(y))$$

and both of these are equivalent to stringing out a series of prefix operations, one for each element of  $B$ , each followed by the  $\rightarrow$  operator and the respective element of  $P$ , all separated by the choice bars  $|$ .

We have not yet specified how processes interact. The combination of processes  $P$  and  $Q$  operating together, is expressed as:

$$P||Q$$

This has the following meaning. For any event in  $\alpha P \cup \alpha Q$ , both processes must execute the event together. Events not in  $\alpha P \cup \alpha Q$  may be executed by either process without cooperating with the other in any way. Such a definition corresponds precisely to what we are interested in with VHDL. Internal events, such as assignments to internal variables, may be executed without cooperation with other processes. Events that affect other processes must be coordinated by cooperative execution.

This indicates why the selection of an alphabet is so important. If an event is a member of the alphabet of two separate processes, and those processes are interacting, they must cooperate in the execution of that event. Alphabet definition is extremely important in the definition of a formal semantic for VHDL. Alphabets may be changed by defining a function that, for each event symbol in the alphabet of a given process,

maps the event symbol into another symbol in another set  $A$ . Given such a function  $f$  whose type is  $\alpha P \rightarrow A$ , we (by abuse of notation) also apply  $f$  to  $P$ ; thus, the process with each symbol in the alphabet of  $P$  transformed by  $f$  is expressed by  $f(P)$ . This is referred to as a change of symbol, and it is vital to the production of a large number of processes from a single process. Change of symbol can be used to define the generate statement of VHDL.

This change of symbol is extended even more powerfully by labeling the process. If a process is labeled, then each event that the process executes can also be labeled by the process name. This allows us to distinguish the alphabets of the processes, to the extent we wish, by prefixing the names of the events by the label. Such a labeled process is denoted by:

$$l : P$$

where  $l$  is the label and  $P$  is the process. This enables a generate statement to label the events of the duplicated process in order to distinguish them, and cooperate as necessary (using indexed labels from the language). This labeling can be extended to multiple labels if necessary (as in nested generates).

One other mechanism of alphabet control needs to be addressed before we continue. This is concealment. Given a process  $P$ , and a set of events  $A$  that we want to ignore in  $P$ , we expressed the process that acts like  $P$  except that each of the events  $A$  is hidden as:

$$P \setminus C$$

All events in  $C$  may not be viewed in the combined process, and any symbol in  $C$  may be used again if desired.

This concludes our initial examination of CSP operations; we will return to some other specialized operations later. Dr. Hoare's text [6] also specifies a number of mathematical laws for each operation, and an implementation strategy. We will not discuss these in this paper (although we will appeal to the existence of an implementation strategy). We make a brief mention here of the use of traces to define the mathematics of the processes.

Processes are defined by the trace of events that the process executes. Thus, the process that represents the execution of the entire VHDL process is represented by the trace of the events that the process executes. The contents of this trace are defined by the mathematical definition of the process operations given above (and others in the text). This definition is given in [6]; here, we limit ourselves to referring to the definition in terms of traces for later discussion.

One additional fact concerning traces: traces have operations on them just as do processes. One such operation is restriction, which is the equivalent of hiding in processes. A trace can be restricted the events in a given set, just as processes can hide all the events in a given set. If  $t$  is the trace of a process, and  $A$  is a set of events that we wish to observe within the trace, then:

$$t \upharpoonright A$$

is the trace that is formed by removing each event from trace  $t$  except those that are in the set  $A$ . This allows us to limit our observation of the events in a trace to a specific set.

### 3 Signals and the Simulation Cycle

As discussed in the previous section, a VHDL signal is a series of time / value pairs, that signify the posting of a given value on a signal. The process that corresponds to the entire VHDL description has an associated trace; however, this trace records all events in the simulation. We propose, as a part of the formal definition of the language, to prefix each event associated with a signal with the name of that signal. If  $trace(D)$  is the trace defined by the VHDL description  $D$ , and  $s$  is a signal that is a part of  $D$ , with  $E(s)$  the set of all events that are prefixed with the signal name  $s$ , then  $(trace(D) \upharpoonright E(s))$  is the events associated with signal  $s$ , and defines the value of signal  $s$ .

We therefore have arrived at a decision concerning how we will represent the values of signal  $s$ . We need to decide how we will represent simulation time. The first impulse is to represent time as a series of events, one for each simulation time in which a process is resumed (or in the CSP terminology, executes and emits an event).

The main reason for resisting this impulse is a better opportunity. VHDL has chosen a centrally monitored simulation cycle for several important reasons, which include efficiency and the ability to pin down a given event to a specific time without complications.

However, for formal simplicity, we may choose another approach. We choose to represent time as a series of femtosecond and delta cycle ticks. This has one extremely important benefit: it completely decentralizes the control of simulation time. Each process now counts femtosecond and delta cycle ticks on its own, and awakens when its internal state indicates resumption. Any implementation based on this idea would be hopelessly inefficient; however, as a formal treatment it has tremendous advantages. No queues of processes ready for resumption are necessary; indeed, the concept of “resumption” is not really necessary. Each process executes based on its local determination of the state of the signals that it reads and the (global, because represented by a single process) simulation time. Using this definition of time, the formal definition is freed from any dependence upon a central simulation controller (beyond the central simulation clock).

This introduces a complication in the definition of the system clock. Each process must participate in the definition of each time tick. Therefore, each process must decide (individually) when to execute each time clock tick, including both femtosecond and delta cycle ticks. The intuitive question is how the system as a whole decides when each macro cycle is complete; i.e., when to stop matching delta cycle ticks and start matching femtosecond cycle ticks.

Recall that the only events that can execute are those that all processes who have the events in their alpha-

bets are ready to execute. This means that a single, “central” process must not be ready to accept a delta cycle tick when no process is ready to resume after a delta cycle. Each individual process must be ready to accept both a femtosecond tick and a delta cycle tick in all situations in which they are not ready to resume; otherwise, time cannot advance.

This indicates that the selection must be made by the central time clock. This is accomplished by including all process events (the event that represents the activation of a process) in the alphabet of the clock process. Assume that the alphabet of the clock process is so extended. The clock process must therefore cooperate in the activation of any process. These considerations come together in the following definition:

$$\begin{aligned}
 FCLOCK &= (fs \rightarrow FCLOCK \mid \\
 &\quad P(x) \rightarrow PCLOCK) \\
 PCLOCK &= (P(x) \rightarrow PCLOCK \mid \\
 &\quad fs \rightarrow FCLOCK \mid \\
 &\quad PD(x) \rightarrow DCLOCK) \\
 DCLOCK &= (P(x) \rightarrow DCLOCK \mid \\
 &\quad PD(x) \rightarrow DCLOCK \mid \\
 &\quad d \rightarrow PCLOCK)
 \end{aligned}$$

where  $P(x)$  represents the resumption of any individual process that is prepared to execute, and  $PD(x)$  represents a “marker” event that is executed by any process that is about to execute a delta cycle (i.e., a `wait for 0ns` statement). There are three cases; after a femtosecond tick, the clock can match either another femtosecond tick or a process activation. After a process activation, the clock can match either another process activation, a femtosecond tick (indicating the end of the delta cycle and an advance in simulation time), or a special delta marker event (indicating a coming delta cycle). After a delta cycle marker event, the clock can match a process activation, another delta cycle marker, or a delta cycle tick (indicating the end of the delta cycle). This defines the fact that a delta cycle can only be initiated by a process resuming at the same simulation time, and nothing else. This clock definition, along with the cooperation of each of the component processes (dis-

cussed later), defines the entire simulation cycle in a very distributed manner.

This deals with a limited part of the simulation cycle. The rest of the simulation cycle concerns the appropriate determination of the driving and resolved value of each signal, using resolution functions and type conversion functions as necessary. These portions of the simulation cycle can be defined purely by functional dependencies. In the intuitive “flow up” (determining the driving values) and “flow down” (determining the resolved values), the dependence of the various driving and resolved values on other values — the partial order defined in the LRM — can be expressed by a careful definition of the functional relationships between the different drivers and signals. This relationship is dealt with more extensively in [1], and we will not go into it further here.

One additional question needs mentioning. This concerns the representation of drivers, signals, resolution functions, and type conversion functions. There are two choices for each of these: define them in terms of functional relationships, as given in [1], or define them as processes (what was once called “implicit processes”, a definitional approach that was rejected in favor of the one currently in the LRM). At this time, there is not enough information to make this choice. The deciding factor should be formal and manipulative simplicity, and it is not clear which approach will eventually turn out to be simpler. We leave this as a subject for further research.

## 4 Defining Processes

A VHDL process is defined as an equivalent CSP process. In the simplest cases, such as concurrent signal assignment statements, this is done with little more mechanism than that done for a VHDL signal assignment statement, requiring only an explicit handling of the femtosecond and delta cycle ticks. In the following, we assume a process based implementation for drivers, since it simplifies the nature of the nand gate. The classic simple two-input nand statement, corre-

sponding to  $c \leftarrow a \text{ nand } b$ , could be implemented as follows:

$$\begin{aligned} \text{NAND} = & (fs \rightarrow \text{NAND} \mid \\ & d \rightarrow \text{NAND} \mid \\ & a : x \rightarrow c : dr : (x \text{ nand } y) \rightarrow \text{NAND} \mid \\ & b : y \rightarrow c : dr : (x \text{ nand } y) \rightarrow \text{NAND}) \end{aligned}$$

where  $x$  and  $y$  are internal storage indications that note the values of the  $a$  and  $b$  signals, respectively, and  $c : dr$  is a label indicating the driver associated with the process assigning to signal  $c$ . Note that the signal events and the driver event are all prefixed with the name of their respective signal or driver.

This concludes our discussion of the concurrent part of the process definition, which is that part most often dealt with in such abbreviated formal language definitions for VHDL. The next question concerns the more standard sequential statements. We may divide these into separate situations: wait statements, and all the rest.

A process statement with multiple wait statements can be defined as a number of mutually recursive process equations; see above for some examples. Each wait statement is a separate, named process equation. The form of each of these equations is based upon the resumption protocol specified by the wait statement; multiple signal lists, timing, or conditions (where the process, as usual, wakes up to evaluate the condition and goes back to sleep if it is false).

For “all the rest”, we refer the reader to Chapter 5 of [6], which deals with sequential processes. We give a brief synopsis of that material here. Sequential composition of processes is implemented in CSP by modeling the successful termination of a process explicitly. Successful termination is signaled by a unique event with a unique symbol,  $\checkmark$ . Given two sequential processes (i.e., two processes that signal their termination using  $\checkmark$ ), the process  $P; Q$  is defined to be that process which acts like  $P$  until it terminates, and then acts like  $Q$ . This provides the base for an entire sequential semantics.

Assignment in particular is handled by the normal

assignment notation:  $x := e$  sets the variable  $x$  to the value of the expression  $e$ . The laws given in the text reflect the normal substitution semantics. Conditionals are also defined:  $P \leftarrow b \rightarrow Q$  means that process which acts like  $P$  if  $b$  is true, and otherwise acts like process  $Q$ . Loops are defined similarly using recursion and conditionals.

Global variables create some different problems (as may be expected). The normal assignment semantics given in the text do not apply, as they are oriented (as are most assignment semantics) to a single thread of control. Rather than attempt to define a completely new operation, global variables may be defined in CSP by assigning a process to each shared variable, prefixed by the variable name. Such a process will be defined as follows:

$$\begin{aligned} \text{GLOB}(x) = & (x : \text{GLOB} \rightarrow \text{GLOB}(x) \mid \\ & \text{assign} : y : \text{GLOB} \rightarrow \text{GLOB}(y)) \end{aligned}$$

where  $\text{GLOB}.x$  is that event that matches a value of  $x$  in variable  $\text{GLOB}$ . Using a separate process assures atomic assignment and reads. No other control is implied, which matches the (extremely unwise) semantics in the present VHDL definition.

This completes our brief description of the CSP process that defines a given VHDL process, entity / architecture pair, or description. A true definition of VHDL in terms of CSP is beyond the scope of this paper (and the unfunded effort of the author). We now proceed to some of the implications of such a definition.

## 5 VHDL, CSP, WAVES, and Test

VHDL specifies a digital device. WAVES specifies a test for that device, in the form of a number of test waveforms that agree with (or conform to) the WAVES specification. A tester is, in some sense, another device; however, it can also be thought of as a larger waveform specification that represents all the

possible waveforms that can be checked by the tester. All of these can be represented as processes; indeed, if formally defined in the appropriate manner, by CSP processes. This gives us the ability to manipulate the results of using the VHDL description, the WAVES dataset, and the tester together.

Before proceeding, we note that WAVES [7] is defined in terms of traces of events. It is commonly stated that WAVES is defined as a VHDL subset, but this is not quite correct. WAVES is *implemented* in terms of a VHDL subset. A careful examination of the WAVES definition results in a definition of the conforming waveforms in terms of the traces of events defined by the WAVES vectors. This is deliberate; it allows tester manufacturers to implement WAVES without implementing VHDL.

This allows us to easily extend the WAVES definition to CSP. There is no process definition generating the waveforms; however, these processes can be derived fairly easily (since the original trace semantics was written with CSP in mind). Note that, in spite of being syntactically a VHDL subset and implementable within a VHDL simulator, the existence of a formal semantics for VHDL is not sufficient to describe WAVES. This is because the VHDL subset definition *models* the WAVES waveform rather than implementing it directly. A window is modeled in the VHDL implementation as an event that signals the opening of a result window, and an event that signals the close of that window. For a CSP definition, this can be more directly modeled as a process that matches a given event within the window and does not match the event outside the window.

With this in mind, we will proceed to examine some of the CSP operations on the various definitions. In the following,  $D$  represents a VHDL description,  $W$  represents a WAVES test dataset, and  $T$  represents a CSP definition of a tester. Given this,

$$D \parallel W$$

represents the set of all possible waveforms that meet the WAVES specifications and also simulate properly

in the VHDL definition. If

$$(D \parallel W) = W$$

then *any* waveform meeting the WAVES specification simulates correctly (the desired situation). If

$$(D \parallel W) \subset W$$

then there are waveforms in the WAVES definition that do *not* simulate correctly on the VHDL description. This may or may not be an error; however, it certainly means that if we implement the WAVES dataset on a tester, we may find a device that conforms to the VHDL description that does not pass the test on the tester. This is obviously undesirable, and indicates that either the dataset needs to be refined or the VHDL implementation is incorrect in some regard. A top-down design methodology using both WAVES and VHDL admits either interpretation; see [5] for more discussion.

We can now investigate the tester half of the equation. The expression

$$W \parallel T$$

represents all test vectors that agree with the WAVES dataset that also is implementable on the tester. Given the reality of the situation, we may presume that

$$(W \parallel T) \subset W$$

or that the tester cannot implement all possible waveforms specified by the WAVES dataset. As long as

$$(W \parallel T) \subseteq (W \parallel D)$$

then we are operationally OK; everything that meets the WAVES dataset and can be implemented on the tester correctly executes on the VHDL description.

Any of the above statements are theorems that can be proven or not proven using the laws of CSP. We therefore have the ability to make statements about the relationship between a description, its tests, and

the implementation of those tests and verify those relationships using the CSP calculus.

Moreover, [6] gives sample implementations for each of the CSP process operations, in terms of producing a sample trace for that process. These implementations are given in a subset of LISP. We can use these implementation as hints concerning how to find something very important: a trace that conforms to  $D \parallel W \parallel T$ . This ability effectively solves the waveform translation problem, which is of considerable commercial interest.

There are significant implementation and efficiency issues here. A tester description will certainly have a large number of non-deterministic choices involved (corresponding to mapping different test pins to different tester resources). Non-deterministic searches are NP complete, regardless of implementation hints. Thus, some intelligent heuristics for such choices will be vital to efficiently selecting an implementable trace that corresponds to a WAVES dataset. Nevertheless, a formal definition and a place to start such investigations is a significant step forwards.

## 6 The Form of a CSP Definition of VHDL

Before concluding, we need to say a brief word about the form of a formal definition for VHDL using CSP. There are, of course, many possible forms of such a definition. This paper has done nothing more than give a very brief sketch of a mapping between VHDL language constructs and CSP processes. The form of the specification of that mapping remains to be decided upon. Of course, whoever does the mapping will make his or her own decision. We would like to put forward a suggestion.

We would suggest that the mapping be given very operationally (that is, the form of the mapping is given operationally, not the mapping itself). A transformational grammar, very similar to a YACC style input file, has two benefits. The first benefit is that it does,

indeed, give a full specification of the translation from VHDL into CSP processes. This is the definitional benefit. The second is that such a description, modified only slightly, can serve as input to a compiler that will take a VHDL description and produce the equivalent CSP process.

Therefore, the first benefit allows the user to make statements about the behavior of the VHDL description, and prove those statements true or false (or fail to prove either). The second benefit allows the user to translate the VHDL description into a CSP process. This is not particularly useful for simulation purposes, given efficiency concerns (recall the form of the system clock). It is useful for feeding the definition into tools, such as waveform translators, that may be quite useful in areas besides simulation.

A YACC description is too informal for such a specification. The reader is referred to the standard works on compiler specification and formal semantic definitions for more information.

## 7 Conclusion

We have presented an examination of the use of CSP (or, rather, a translation from VHDL into CSP) in the formal definition of VHDL. Such a formal definition has several very real benefits. First, and foremost, the distributed nature of the event matching semantics allows us to completely separate the formal definition from a central “simulation controller”, or “simulation kernel”. The closest thing to a simulation kernel is the central clock process. This process does little more than enforce the rule that delta cycles must be separated by actual process resumptions, and they end when no more process resumptions remain for a given simulation time.

In fact, the use of CSP alone has some disadvantages for the purely mathematical portion of the definition. Even [6] assumes the implementation of common mathematical functions. A complete CSP definition is certainly possible, and may not even be diffi-

cult; however, it would share many of the difficulties of a complete axiomatization of number theory using (for example) the Peano axioms.

We anticipate using a blend of a CSP characterization of processes and a functional characterization of the mathematics of the language. See [1] for an explanation of the functional characterization of parts of the VHDL language. Some of this division is clear; for example, operations and function calls should be defined using a functional formalization, while processes and assignment should be defined using a CSP formalism. There is a middle ground between the two where the decision is not so clear; for example, it is not clear without attempting some examples whether it is best to define the operation of resolution functions functional, as in [1], or as an “implicit process” using CSP.

We have also shown relationships between a CSP representation of a VHDL model, a WAVES dataset, and a CSP specification of a tester. These relationships are of tremendous benefit in terms of making formally defined statements about the waveforms and testers; for example, “What does it mean for a VHDL description to completely implement a WAVES dataset? What does it mean for a WAVES dataset to be implementable with respect to a specific tester?”

In terms of future research, a formal definition opens the door to make similar statements about other VHDL models. First and foremost on the list for future research is the actual definition of VHDL using CSP. Without this, any potential benefits are pure speculation. A mechanical translator from VHDL into the equivalent CSP process would also be advantageous.

Given these, there are several areas of research that immediately suggest themselves. One is immediately connected to the RASSP program. Performance models are a vital part of the RASSP methodology [9]. A missing piece in the effective use of performance models is the question of assuring that a behavioral VHDL model conforms to a VHDL performance model. Given a change of alphabet, we can

precisely formulate the question using CSP. If  $P$  is a performance model,  $B$  is a behavioral model, and  $f$  a function that translates all values from  $B$  into events for  $P$ , then (assuming that the performance timing characteristics are looser than the behavioral ones):

$$f(B) \parallel P = f(B)$$

precisely captures the statement that the behavioral model must correspond to the performance model. Such a statement is difficult to make in many other formalisms.

Similar statements can be made in other areas. Where software, and in particular parallel software, is specified in CSP or in a language defined in terms of CSP, we can make statements about the hardware / software system using the CSP models.

Specifications for a VHDL (or software, or other) models can be made using predicates on the process trace, as specified in [6]. These predicates can be made with a normal predicate calculus based language. In particular, we would suggest the SDDL under development within the DASC for this purpose. Investigating the relationship between these two notations is another fertile area for future research.

## References

- [1] David L. Barton. A functional characterization of elements of the VHDL simulation cycle. In *Using VHDL for Electronic Product Design*, pages 91–96. VHDL User’s Group, April 1991.
- [2] W. Damm, B. Josko, and R. Schlor. A net-based semantics for VHDL. In *Proceedings of the European Design Automation Conference with EURO-VHDL ’93*, pages 514–519. CCH, September 1993.
- [3] K. C. Davis. A denotational definition of the VHDL simulation kernel. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages*, pages 509–521, 1993.

- [4] Ivan V. Filippenko. VHDL verification in the State Delta Verification System (SVDS). In *Proceedings of the 1991 International Workshop on Formal Verification in VLSI Design*, January 1991.
- [5] Christopher J. Flynn, Fredrick G. Hall, James P. Hanna, and Mark T. Pronobis. Using WAVES in a top down design methodology. In *Notebook of Sessions: VHDL International Users Forum, Fall Conference, 1994*. VHDL International, November 1993.
- [6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [7] Institute of Electrical and Electronic Engineers. *IEEE Standard for Waveform and Vector Exchange (WAVES)*. Number IEEE Std. 1029.1-1991. Institute of Electrical and Electronic Engineers, New York, New York, 1991.
- [8] Institute of Electrical and Electronic Engineers. *IEEE Standard VHDL Language Reference Manual (VHDL-LRM)*. Number IEEE Std 1076-1993. Institute of Electrical and Electronic Engineers, New York, New York, 1993.
- [9] Fred Rose, Todd Steeves, and Todd Carpenter. VHDL performance modeling. In *Proceedings of the First Annual RASSP Conference*. ARPA / Tri-Service RASSP Program, August 1994.
- [10] John P. van Tassel. The semantics of VHDL with VAL and HOL: Towards practical verification tools. Technical Report 196, University of Cambridge Computer Laboratory, June 1990.
- [11] John P. van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proff Assistant*. PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England, July 1993.
- [12] Phillip A. Wilsey. Developing a formal semantic definition of VHDL. In Jean Mermet, editor, *VHDL for Simulation, Synthesis, and Formal Proofs of Hardware*, pages 243–256. Kluwer Academic Publishers, Boston, MA, 1992.