A Formalization of a Subset of VHDL in the Boyer-Moore Logic *

David M. Russinoff
Computational Logic, Inc.
1717 West Sixth Street, Suite 290, Austin, TX 78703
russ@cli.com

September 20, 1994

Abstract

We present a mathematical definition of a hardware description language that admits a semantics-preserving translation to a subset of VHDL. The language is based on the VHDL model of event-driven simulation and includes behavioral and structural circuit descriptions, the basic VHDL propagation delay mechanisms, and both zero and nonzero delays. It has been formally encoded in the computational logic of Boyer and Moore, which provides a LISP implementation as well as a facility for mechanical proof-checking. We prove a number of basic properties of the simulator, which we apply to the analysis of gate-level designs of a one-bit adder and a d-flip-flop.

1 Introduction

The VHSIC Hardware Description Language [3] (VHDL) has gained wide acceptance as a tool for hardware design and simulation. However, the limitations of simulation as a method of design validation are well known. A formal verification system based on VHDL would therefore have clear practical value. Naturally, a prerequisite for any such system is a precise understanding of VHDL semantics. Unfortunately, VHDL was not intended for formal analysis; it is not surprising that its semantics are complicated and obscure.

The first objective of this paper is a rigorous exposition of a core subset of VHDL that is small enough to admit a clear and simple semantic definition, but extensive enough to provide realistic gate-level descriptions of interesting circuits. Thus, we avoid complicated language constructs and focus on the VHDL models of time, signal behavior, propagation delay, and event-driven simulation. Our subset includes only two types of VHDL statements:

*This work was sponsored in part at Computational Logic, Inc. by National Aeronautics and Space Administration Langley Research Center (NAS1-18878). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., NASA Langley Research Center, or the U.S. Government.

(1) concurrent signal assignment statements of the form

$$s \Leftarrow \mathit{expr}\ \ [\texttt{transport}]\ \texttt{after}\ n\ \texttt{ps.},$$

where s is a port of mode `out`, *expr* is a Boolean expression involving only ports of mode `in`, and n may be either 0 or positive;

(2) component instantiation statements of the form

$$\mathit{label: component}\ \texttt{port map}\ (s_1, \ldots, s_k),$$

where each s_i is either a port of mode `in`, `out`, or `buffer`, or a declared signal.

While other formalization efforts (e.g., [2, 6, 8]) have addressed comparable or larger VHDL subsets, relatively little progress has been reported in the formal verification of VHDL programs. Our second objective is to demonstrate how our approach provides for the proof of correctness of interesting behavioral specifications of the programs in our subset. To this end, we present some relevant general theorems, which we apply to several example circuits.

Since we are not concerned with syntactic issues, our language definition is based on an abstract syntax that is more amenable to direct formal analysis than that described in [3]. The correspondence between the two is given by a translator from our language to VHDL, which is described elsewhere [4]. Here, we concentrate on a mathematical treatment of the abstract language. This begins in Section 3, where we present the notions of *time* and *waveform*, on which the semantics of the language are based. We also define two waveform transformations that embody the main propagation delay modes of VHDL, *transport* and *inertial*, and derive their fundamental properties.

In Section 4, we describe the form and execution of *behavioral modules*, which are used to model gates and also to specify abstractly the behavior of circuits. Section 5 discusses *structural modules*, which provide hierarchical descriptions of circuits in terms of connections among their components. For the purpose of illustration, we exhibit the actual VHDL code generated by the translator for modules of both types.

The semantics of the language are given by an interpreter function, *sim*, which produces a list of waveforms that represent the output generated by a module in response to a given list of input waveforms. The definition of *sim* is presented in Section 6, along with a number of basic results pertaining to its behavior. Finally, in Section 7, we derive behavioral specifications of two modules: a one-bit adder and a d-flip-flop.

The design of our language is based on S-expressions, the data objects of LISP, which are defined in Section 2. This choice was motivated by our desire to support its analysis with the use of the Nqthm system of Boyer and Moore [1]. Nqthm is based on a constructive formal logic for which the intended model is the domain of S-expressions. Thus, there is a correspondence between the formulas of this logic and informal propositions about S-expressions. A user of the system may extend the logic by adding axioms that correspond to definitions of computable functions over this domain.

Mechanical support for the Nqthm logic is provided by a LISP implementation that includes (1) an evaluator that computes values of functions defined in the logic, and (2) a theorem prover that may be used to derive logical consequences of the axioms. Since these theorems may be interpreted as propositions about functions of S-expressions, the prover may be used to verify (formally and mechanically) the correctness of properties of these functions that have been derived by traditional (informal) mathematical methods.

All of the functions involved in the construction of our language, which we shall describe informally, meet the computability requirement for encoding as Nqthm definitions [1]. In fact, we have developed an Nqthm theory that formalizes these functions, including the module recognizers that form the syntax of the language, the interpreter that constitutes its semantics, and various procedures for deriving behavioral specifications of its programs. Thus, we have a complete LISP implementation of our language, provided by the Nqthm evaluator.

Moreover, all of our results, which are justified by informal (but mathematically rigorous) proofs, correspond in a natural way to Nqthm formulas. Thus, these proofs could, in principle, be checked mechanically by the Nqthm prover. At the time of this writing, significant progress has been made toward this objective; its completion remains a goal of our research.

Another benefit of the Nqthm formalization is that it provides a basis for a LISP implementation of the translator from our language to VHDL [4]. This potentially allows commercial VHDL synthesis tools to be used to implement our programs in silicon. As another application of more immediate interest, we have actually executed the translations of many of our programs using the Vantage VHDL simulator. For the simulations that we have tested, which include all of those described herein, the Vantage results were identical to those produced by our LISP-based interpreter. Since the official description of VHDL [3] is often ambiguous, this offers useful evidence that we have achieved our goal of semantically capturing the VHDL subset in which we are interested.

2 S-expressions

Along with the set $\mathbf{N}$ of natural numbers, we posit a set $\mathbf{B} = \{\mathcal{T}, \mathcal{F}\}$ and an infinite set $\mathbf{L}$, the elements of which are called *Boolean* and *literal atoms*, respectively. These three sets are assumed to be pairwise disjoint, and any element of their union is called an *atom*. We further assume that no atom is an ordered pair of atoms, and we recursively define an *S-expression* to be an atom or an ordered pair of S-expressions. $\mathbf{S}$ denotes the set of all S-expressions. Three basic operations on $\mathbf{S}$ are defined: If $z = (x, y) \in \mathbf{S} \times \mathbf{S}$, then $car(z) = x$, $cdr(z) = y$, and $cons(x, y) = z$.

We also assume the existence of various distinct literal atoms, which we shall mention as we proceed. Among these is the atom `INFINITY`. We define a *generalized number* to be an atom that is either `INFINITY` or an element of $\mathbf{N}$. Both the order relation and the addition operation on $\mathbf{N}$ are extended to the set of generalized numbers in the natural manner: for any $n \in \mathbf{N}$, $n < \texttt{INFINITY}$ and $n + \texttt{INFINITY} = \texttt{INFINITY} + n = \texttt{INFINITY}$.

A *list* is an S-expression that is either the literal atom `NIL` or an ordered pair $z \in \mathbf{S} \times \mathbf{S}$ such that $cdr(z)$ is a list. The list `NIL` is denoted alternatively as (), and a non-`NIL` list z is denoted as $(a_1 \ \ldots \ a_n)$, where $a_1 = car(z)$ and $(a_2 \ \ldots \ a_n)$ denotes $cdr(z)$. In this case, n is the *length* of z, and $a_1, \ldots, a_n$ are its *members*. For $1 \leq i \leq n$, $nth(i, z)$ is defined to be a_i. A list is a *bit vector* if each of its members is a Boolean atom.

A function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is an *n-ary Boolean function*. The following Boolean functions are called *elementary*: the 0-ary functions *t0* and *f0*, with values $\mathcal{T}$ and $\mathcal{F}$, respectively; the unary function *not*1; the binary functions *and2*, *or2*, *nand2*, *nor2*, *xor2*; and the ternary functions *and3*, *or3*, *nand3*, *nor3*, *xor3*. The definitions of these functions are assumed to be understood.

For the purpose of encoding Boolean function calls, we also assume that each elementary Boolean function f is associated with a unique literal atom $\bar{f}$ that is denoted with the same name as f. Thus, the function *not1* is associated with the literal atom $\overline{not1} =$ `NOT1`. We define a *Boolean term* over a list L of distinct literal atoms to be an S-expression that is either (a) a member of L, or (b) a list $(\bar{f}\,\tau_1\,\ldots\,\tau_n)$, where f is an n-ary elementary Boolean function and each τ_i is an Boolean term over L.

Let $L = (s_1\,\ldots\,s_k)$ be a list of distinct literal atoms and let $V = (v_1\,\ldots\,v_k)$ be a bit vector. Then $pairlist(L, V)$ is the list $A = ((s_1, v_1)\,\ldots\,(s_k, v_k))$, which is called an *association list*. If τ is a Boolean term over L, then we define $eval(\tau, A)$ to be (a) v_i, if $\tau = s_i$, or (b) $f(eval(\tau_1, A), \ldots, eval(\tau_n, A))$, if $\tau = (\bar{f}\,\tau_1\,\ldots\,\tau_n)$.

3 Waveforms

Let $\mathbf{T}$ be the quotient set determined by the equivalence relation on $\mathbf{N} \cup \mathbf{N}{\times}\mathbf{N}$ that identifies each $n \in \mathbf{N}$ with the pair $(n, 0) \in \mathbf{N} \times \mathbf{N}$. An element of $\mathbf{T}$ is called a *time object*. Thus, any element of $\mathbf{N}$ or $\mathbf{N}{\times}\mathbf{N}$ denotes a unique time object, with the understanding that for $n \in \mathbf{N}$, n and $(n, 0)$ denote the same object.

The motivation for this ordered-pair model of time is the need to provide records of the behavior of zero-delay devices. The components of a time object (n, k) may be interpreted as follows: n represents the number of time units, which we arbitrarily take to be picoseconds, that have elapsed since the start of a simulation; k represents the number of successive delta cycles that have occurred during the current time unit.

Thus, $\mathbf{T}$ is ordered according to the lexicographic order on $\mathbf{N}{\times}\mathbf{N}$, which is consistent with the natural ordering of $\mathbf{N}$: for time objects $t_1 = (n_1, k_1)$ and $t_2 = (n_2, k_2)$, $t_1 \leq t_2$ iff $n_1 \leq n_2$ and either $n_1 < n_2$ or $k_1 \leq k_2$. Thus the minimum element of $\mathbf{T}$ is the time object that is denoted alternatively as 0 or $(0, 0)$. For $t_1, t_2 \in \mathbf{T}$, the interval $\{t \in \mathbf{T} : t_1 \leq t < t_2\}$ will be denoted as $[t_1, t_2)$.

An *event* is an ordered pair $e = (v, t)$, where $v = value(e) \in \mathbf{B}$ and $t = time(e) \in \mathbf{T}$. Let $w = ((v_n, t_n)\,\ldots\,(v_0, t_0))$ be a list of events. If $t_i > t_{i-1}$ and $v_i \neq v_{i-1}$ for $0 < i \leq n$, and $t_0 = 0$, then w is a *waveform*. Note that according to this definition, successive events of a waveform must have different values; in VHDL terminology, all transactions are events. This restriction is consistent with the absence of implicit signals from our subset: since there is no way to detect transactions other than events (e.g., by means of the *ACTIVE* and *TRANSACTION* attributes), they may be ignored.

We define $\hat{w} : \mathbf{T} \rightarrow \mathbf{B}$ by $\hat{w}(t) = v_j$, where j is the greatest value of i satisfying $t_i \leq t$; $\hat{w}(t)$ is called the *value of* w *at* t. Note that $\hat{w}_1 = \hat{w}_2$ iff $w_1 = w_2$. If $t = t_j$, then we shall say that w *has a new value at* t. We also define the *history of* w *relative to* t to be the waveform $hist(w, t) = ((v_j, t_j)\,\ldots\,(v_0, t_0))$.

A *packet* is a list of waveforms, $p = (w_1\,\ldots\,w_n)$, $n \geq 0$. For any $t \in \mathbf{T}$, the *value of* p *at* t is the bit vector $\hat{p}(t) = (\hat{w}_1(t)\,\ldots\,\hat{w}_n(t))$; p *has a new value at* t if any member of p does. The *history* of p relative to t is the packet $hist(p, t) = (hist(w_1, t)\,\ldots\,hist(w_n, t))$.

The behavior of each signal occurring in a circuit will be modeled as a waveform. During the course of a simulation, these waveforms are updated at various times. When a waveform is considered in the context of a current time t_0, each of its members e is viewed as a past, current, or future event, according to the relationship between $time(e)$ and t_0. Past and present events are immutable, but future events are subject to deletion as they are superceded by newly scheduled events, as described below.

Whenever a new event e is to be scheduled for a signal, $time(e)$ is computed from the current time $t_0 = (n, k)$ and a delay $d \in \mathbf{N}$ that is associated with the signal, by means of an addition operation from $\mathbf{T}\times\mathbf{N}$ to $\mathbf{T}$, defined as follows:

$$(n,k) \oplus d = \begin{cases} (n+d, 0) & \text{if } d \neq 0 \\ (n, k+1) & \text{if } d = 0. \end{cases}$$

Thus, regardless of delay, when a new event $e = (v, t_v)$ is scheduled on a waveform w at time t_0, we have $t_0 < t_v$. The scheduling may be performed by either of two procedures, corresponding to the *transport* and *inertial* delay modes of VHDL. Note that the definitions of these procedures are somewhat different from the processes described in[3], due to our restricted notion of *waveform*.

Transport delay is the simpler of the two: each event (v', t') with $t' \geq t_v$ is deleted from w, and e is then *consed* to the result, unless that result already has value v at t_v. The updated waveform w' is computed as the value of $transport(w, v, t_v)$, which is defined recursively as follows:

(1) Let $car(w) = (v_f, t_f)$. If $t_f \geq t_v$, then $w' = transport(cdr(w), v, t_v)$; otherwise:

(2) If $v_f = v$, then $w' = w$; otherwise:

(3) $w' = cons((v, t_v), w)$.

Alternatively, w' may be described in terms of the function $\hat{w}'$:

$$\hat{w}'(t) = \begin{cases} v & \text{if } t \geq t_v \\ \hat{w}(t) & \text{if } t < t_v. \end{cases}$$

Inertial delay is somewhat more complicated: every event (v', t') with $t' > t_0$ is deleted from w, and if $\hat{w}(t_0) \neq v$, then a single event with value v is consed to the result. If $\hat{w}(t_v) = v$, then the time of this event is the time of the last event of w that precedes t_v; otherwise, it is t_v. Note that this procedure takes the current time t_0 as an additional argument, and requires that $t_0 < t_v$. The recursive definition of $w' = inertial(w, v, t_0, t_v)$ is given as follows:

(1) Let $\bar{w} = hist(w, t_0)$. If $\hat{w}(t_0) = v$, then $w' = \bar{w}$; otherwise:

(2) Let $car(w) = (v_f, t_f)$. If $t_f \geq t_v$, then $w' = inertial(cdr(w), v, t_0, t_v)$; otherwise:

(3) If $v_f = v$, then $w' = cons((v, t_f), \bar{w})$; otherwise:

(4) $w' = cons((v, t_v), \bar{w})$.

The following is a useful summary of both propagation functions. Each result may be proved by a straightforward induction. Note that (b) is consistent with our earlier informal observation that past and present events are immutable:

Lemma 3.1 *Let w be a waveform, let t_0, t_1, and t_v be time objects with $t_0 < t_v$, and let w' be either $transport(w, v, t_v)$ or $inertial(w, v, t_0, t_v)$. Then*

(a) $\hat{w}'(t) = v$ for $t \geq t_v$;

(b) $\hat{w}'(t) = \hat{w}(t)$ for $t \leq t_0$;

(c) if $t_1 \leq t_0 \leq t_2 \leq t_v$ and $\hat{w}(t) = u$ for $t \in [t_1, t_2)$, then $\hat{w}'(t) = u$ for $t \in [t_1, t_2)$.

A similar induction shows that both procedures are "idempotent" in the following sense:

Lemma 3.2 *If w is a waveform and t_0, t_v, t'_0, t'_v are time objects with $t_0 < t_v$, $t'_0 < t'_v$, $t_0 < t'_0$, and $t_v < t'_v$, then*

(a) $transport(transport(w, v, t_v), v, t'_v) = transport(w, v, t_v)$;

(b) $inertial(inertial(w, v, t_0, t_v), v, t'_0, t'_v) = inertial(w, v, t_0, t_v)$.

4 Behavioral Modules

The simplest programs of our language are the behavioral modules, which contain explicit information concerning propagation delay and the functional dependence of outputs on inputs.

A *behavioral module* is a list $M =$ (BEHAV I O T P D), where

(1) BEHAV is the identifying literal atom for modules of this type;

(2) $I = I(M) = (r_1 \ \ldots \ r_m)$ is a list of literal atoms called the *inputs* of M;

(3) $O = O(M) = (s_1 \ \ldots \ s_n)$ is a list of literal atoms called the *outputs* of M;

(4) $T = T(M) = (\tau_1 \ \ldots \ \tau_n)$ is a list of elementary Boolean terms over $I(M)$, called the *output terms* of M;

(5) $D = D(M) = (d_1 \ \ldots \ d_n)$ is a list of natural numbers, the *delays* of M;

(6) $P = P(M) = (p_1 \ \ldots \ p_n)$ is a list of literal atoms called the *propagation modes* of M, each of which is either TRANSPORT or INERTIAL.

The members of the list $(r_1 \ \ldots \ r_m \ s_1 \ \ldots \ s_n)$ are required to be distinct and are called the *signals* of M.

Note that each output is associated with a term, a mode, and a delay. If every term is either an atom or a list of atoms, (i.e., contains no nested function calls), then M is *primitive.*

Gates are generally modeled as primitive modules with inertial delays. For example, we represent a simple 2-input nand gate as the primitive module nand2:

```
(BEHAV (A B) (C) ((NAND2 A B)) (2000) (INERTIAL))
```

We may define a similar behavioral module, with n inputs and 1 output, corresponding to each elementary n-ary Boolean function, arbitrarily taking the delay to be 2000 in each case. In the sequel, we shall refer to these primitive modules without explicitly listing their definitions.

Transport mode is often used to model wires along which pulses of arbitrarily small duration are propagated to the delayed signal. For the purpose of illustration, the following primitive module m is defined to have one output of each propagation mode:

```
ENTITY m IS
 PORT(a,b: IN BIT; c,d: OUT BIT)
END m;

ARCHITECTURE m OF m IS
BEGIN
  c <= a NAND b AFTER 2 NS;
  d <= TRANSPORT NOT a AFTER 5 NS;
END m;
```

(a)

```
ENTITY adder2 IS
 PORT (a,b,c: IN BIT; l,h: OUT BIT)
END adder2;

ARCHITECTURE adder2 OF adder2 IS
  COMPONENT nand
    PORT(a,b: IN BIT; l,h: OUT BIT);
  END COMPONENT;
  SIGNAL t1,t2,t3,t4,t5,t6,t7: BIT;
BEGIN
  I1: nand PORT MAP (a,b,t1);
  I2: nand PORT MAP (a,t1,t2);
  I3: nand PORT MAP (b,t1,t3);
  I4: nand PORT MAP (t2,t3,t4);
  I5: nand PORT MAP (c,t4,t5);
  I6: nand PORT MAP (c,t5,t7);
  I7: nand PORT MAP (t5,t4,t6);
  I8: nand PORT MAP (t5,t1,h);
  I9: nand PORT MAP (t7,t6,l);
END adder2;
```

(b)

Figure 1: VHDL Code

```
(BEHAV (A B) (C D) ((NAND2 A B) (NOT1 A)) (2000 5000) (INERTIAL TRANSPORT))
```

The VHDL code corresponding to a behavioral module consists of

(a) an entity declaration, consisting of a port clause listing the input signals as ports of mode `IN` and the output signals as ports of mode `OUT`, all of type `BIT`;

(b) an architecture body, consisting of a concurrent signal assignment statement corresponding to each output signal.

The code (generated by our translator) for the module `m` defined above is displayed in Figure 1(a). Note that our time units are interpreted by the translator as picoseconds, and hence the delays are expressed as 2 and 5 nanoseconds. Note also that there is no mention of inertial delay in the translation, since this is the VHDL default mode.

Another example of a behavioral module is the 1-bit adder `adder1`:

```
(BEHAV (A B C) (L H)
  ((XOR3 A B C) (OR2 (AND2 A (OR2 B C)) (AND2 B C)))
  (12000 10000)
  (INERTIAL INERTIAL))
```

The two outputs of this module represent the 2-bit sum of the three input bits. Since the higher-order "carry" output bit is not expressed as an elementary function of the inputs, this is not a primitive module.

Let $s = nth(j, O(M))$ be an output of a behavioral module M. Let $\tau = nth(j, T(M))$ be the corresponding term. For any bit vector V of the same length as $I(M)$, we define the *combinational value* of s w.r.t. V as $cv(s, V, M) = eval(\tau, pairlist(I(M), V))$.

We shall say that a list of waveforms is an *input* (resp., *output*) *packet* for a module M if it has the same length as $I(M)$ (resp., $O(M)$). The semantics of behavioral modules are defined by a function *exec* of four arguments: (1) a module M, (2) an input packet p_{in} for M, (3) an output packet $p_{out} = (w_1 \ldots w_n)$ for M, and (4) a time object t_0. The value of $exec(M, p_{in}, p_{out}, t_0)$ is the updated output packet $p'_{out} = (w'_1 \ldots w'_n)$ that results from "executing" M at t_0. It is defined as follows: For $i = 1, \ldots, n$, let v_i be the combinational value of $nth(i, O(M))$ w.r.t. $\hat{p}_{in}(t_0)$, and let $t_i = t_0 \oplus nth(i, D(M))$. Then w'_i is either $transport(w_i, v_i, t_i)$ or $inertial(w_i, v_i, t_0, t_i)$, according to $nth(i, P(M))$.

Our first observation concerning the behavior of *exec* is that its value depends only on the current values of the input:

Lemma 4.1 *Let p_1 and p_2 be input packets and let p_{out} be an output packet for a behavioral module M. For any $t_0 \in \mathbf{T}$, if $\hat{p}_1(t_0) = \hat{p}_2(t_0)$, then $exec(M, p_1, p_{out}, t_0) = exec(M, p_2, p_{out}, t_0)$.*

Two other basic properties may be derived as consequences of Lemmas 3.1(b) and 3.2:

Lemma 4.2 *Let p_{in} and p_{out} be an input packet and an output packet for a behavioral module M. For any $t_0 \in \mathbf{T}$, $hist(exec(M, p_{in}, p_{out}, t_0), t_0) = hist(p_{out}, t_0)$.*

Lemma 4.3 *Let p_{in} and p_{out} be an input packet and an output packet for a behavioral module M and let t_0 and t_1 be time objects. If $t_0 < t_1$ and $\hat{p}_{in}(t_0) = \hat{p}_{in}(t_1)$, then $exec(M, p_{in}, exec(M, p_{in}, p_{out}, t_0), t_1) = exec(M, p_{in}, p_{out}, t_0)$.*

5 Structural Modules

Our language also includes modules that represent hierarchically constructed circuits. These structures contain information concerning interconnections among the modules of which they are composed.

A *structural module* is a list $M = (\texttt{STRUCT}\ I\ O\ S\ LI\ LO)$, where

(1) `STRUCT` is the identifying literal atom for modules of this type;

(2) $I = I(M) = (r_1 \ldots r_m)$ is a list of literal atoms called the *(global) inputs* of M;

(3) $O = O(M) = (s_1 \ldots s_n)$ is a list of literal atoms called the *(global) outputs* of M;

(4) $S = S(M) = (\mu_1 \ldots \mu_k)$ is a list of (structural or behavioral) modules, called the *submodules* of M;

(5) $LI = LI(M) = (A_1 \ldots A_k)$, where for $j = 1, \ldots, k$, $A_j = (a_{j1} \ldots a_{jm_j})$ is a list of literal atoms called the j^{th} *local inputs* of M, and m_j is the length of $I(\mu_j)$;

(6) $LO = (B_1 \ldots B_k)$, where for $j = 1, \ldots, k$, $B_j = (b_{j1} \ldots b_{jn_j})$ is a list of literal atoms called the j^{th} *local outputs* of M, and n_j is the length of $O(\mu_j)$.

The members of the list $(r_1 \ldots r_m\, b_{11} \ldots b_{1n_1} \ldots b_{k1} \ldots b_{kn_k})$, consisting of the global inputs and all local outputs, are required to be distinct and are called the *signals* of M. There is no such constraint on the global outputs or local inputs, but each local input must be a signal of M, and each global output must be a local output.

Note that the local inputs and outputs of M correspond to its submodules. Thus, intuitively, the submodules of a structure generate signals that are distinct from each other and from the structure's inputs. Each signal may be connected to arbitrarily many submodule inputs. A signal other than a global input may serve as any number of global outputs, but global inputs and outputs are distinct.

One additional constraint must be imposed on structural modules: in order to ensure that any simulation (as defined in the next section) of a module terminates, our structures are required to be free of zero-delay cyclic paths. Several preliminary definitions will be needed in order to make this notion precise.

We shall define a computable function that measures the (possibly infinite) maximum length of any path of signals within a structure along which the total delay is 0. The definition will be based on an auxiliary function, $\delta(M, s, E, L)$, the arguments of which are to be understood as follows:

(1) M may be either the top-level structure or one of its components at any level of the hierarchy;

(2) s is a signal of M;

(3) $E = (e_1 \ldots e_n)$ is a list of generalized numbers corresponding to $O(M)$. For each i, e_i is intended to represent the maximum length of any path that starts at the i^{th} output and leads out of M. Such a list is called an *environment* for M;

(4) L is a list of signals of M, each of which is known to lie on some infinite path.

Under these assumptions, we may think of $\delta = \delta(M, s, E, L)$ as the maximum length of a path starting at s. It is computed recursively as follows:

(1) If s is a member of L, then $\delta =$ `INFINITY`. Otherwise:

(2) Let $\Delta_1 = max\{e_i : s = s_i\}$, where $O(M) = (s_1 \ldots s_n)$. (The maximum of the null set is taken to be 0.)

(3) Suppose M is behavioral. Let $D(M) = (d_1 \ldots d_n)$. If s is an input of M and some $d_i > 0$, then let $\Delta_2 = 1 + max\{e_i : d_i = 0\}$; otherwise, $\Delta_2 = 0$.

(4) Suppose M is structural with $S(M) = (\mu_1 \ldots \mu_k)$. For $1 \le i \le k$, let $nth(i, LI(M)) = (a_{i1} \ldots a_{im_i})$, $nth(i, LO(M)) = (b_{i1} \ldots b_{in_i})$, $I(\mu_i) = (\alpha_{i1} \ldots \alpha_{im_i})$, and let E_i be the environment $(\epsilon_{i1} \ldots \epsilon_{in_i})$ for μ_i, where for $1 \le k \le n_i$, $\epsilon_{ik} = \delta(M, b_{ik}, E, cons(s, L))$. Let $\delta_{ij} = \delta(\mu_i, \alpha_{ij}, E_i, \texttt{NIL})$ for $i = 1, \ldots, k$ and $j = 1, \ldots, m_i$. Let $\Delta_2 = max\{\delta_{ij} : s = a_{ij}\}$.

(5) $\delta = max(\Delta_1, \Delta_2)$.

The function Δ is defined by by $\Delta(M, s, E) = \delta(M, s, E, \texttt{NIL})$. Next, we define the *relative δ-depth* of a module M with respect to an environment E to be the number ρ computed as follows:

(1) Let D_0 be the maximum value of $\Delta(M, s, E)$ over all signals s of M. If M is behavioral, then $\rho = D_0$. Otherwise:

(2) Let M be structural with $S(M) = (\mu_1 \ldots \mu_k)$. For $1 \leq i \leq k$, let $nth(i, LO(M)) = (b_{i1} \ldots b_{in_i})$ and let D_i be the relative δ-depth of μ_i with respect to the environment $(\Delta(M, b_{i1}, E) \ldots \Delta(M, b_{in_i}, E))$. Then $\rho = max(D_0, D_1, \ldots, D_k)$.

Finally, we define the *δ-depth* of M to be its relative δ-depth with respect to the environment $(0 \ldots 0)$. This represents the length of the longest 0-delay path through M. If it is not `INFINITY`, we shall say that M is *δ-acyclic*. All structural modules in our language are required to have this property.

Although we have gone to considerable effort to formalize the VHDL "delta delay" mechanism, the examples in which we are interested exhibit only positive delays. Our first example is the structural module `adder2`, composed of nine nand gates and intended as a gate-level "implementation" of the behavioral module `adder1`:

```
(STRUCT (A B C) (L H)
  (nand2 nand2 nand2 nand2 nand2 nand2 nand2 nand2 nand2)
  ((A B) (A T1) (B T1) (T2 T3) (C T4) (T5 T4) (C T5) (T5 T1) (T7 T6))
  ((T1) (T2) (T3) (T4) (T5) (T6) (T7) (H) (L)))
```

The VHDL code corresponding to a structural module consists of

(a) an entity declaration, consisting of a port clause listing the inputs as ports of mode `IN` and each output as a port, either of mode `BUFFER`, if it occurs as a local input, or of mode `OUT`, if it does not;

(b) an architecture body, consisting of a component declaration corresponding to each module that occurs as a submodule, a signal declaration corresponding to each local output that it not a global output (and hence does not already occur as a port), and a component instantiation statement corresponding to each submodule.

The code for `adder2` is shown in Figure 1(b), and a circuit diagram appears in Figure 2(b). Later, we shall compare the behaviors of `adder1` and `adder2`.

Of course, a signal path may be cyclic, provided that some signal in the path is associated with a positive delay. This is an important feature of our language, as it allows the modeling of state-holding devices. Figure 2(a) shows a clocked d-flip-flop, which is modeled by the structural module `dff`:

```
(STRUCT
 (CLK D)
 (Q QN)
 (nand2 nand2 nand3 nand2 nand2 nand2)
 ((B2 B1) (A1 CLK) (B1 CLK B2) (A2 D) (B1 QN) (Q A2))
 ((A1) (B1) (A2) (B2) (Q) (QN) (A2)))
```

The submodules include five 2-input nand gates and a 3-input nand gate `nand3`, which is similarly defined with an inertial delay of 2000.

We shall define the semantics of structural modules by means of a function *step*, based on the *exec* function of Section 4. Note that the notions of input and output packets may be naturally applied to any module. For a structural module M, however, instead

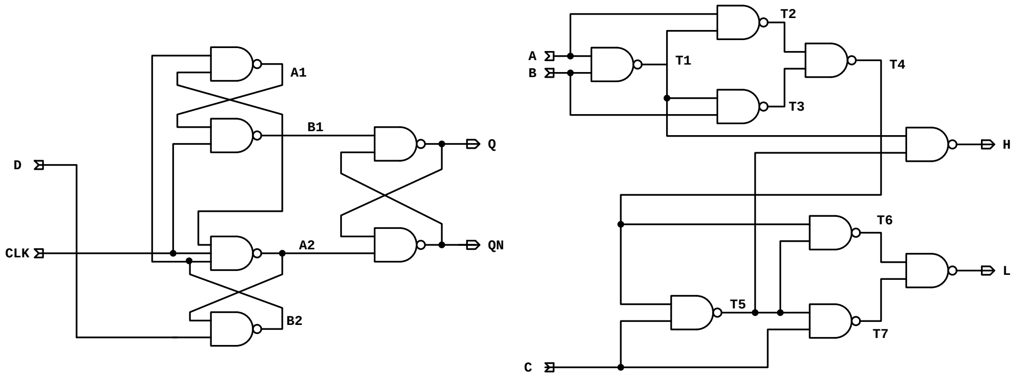

Figure 2: (a) D-Flip-Flop (b) 1-Bit Adder

of a simple output packet, the third argument of *step* must be an object that consists of a waveform corresponding to each signal generated by each component of M. Thus, for any module M, we define a *bundle for M* to be a list B such that (a) if M is behavioral, then B is an output packet for M; (b) if M is a structure with $S(M) = (\mu_1 \, \ldots \, \mu_k)$, then $B = (\beta_1 \, \ldots \, \beta_k)$, where β_i is a bundle for μ_i, $i = 1, \ldots, k$.

Let B be a bundle for a module M and let s be a signal of M that is not an input of M. The *waveform for s determined by B* is the waveform w that is computed as follows: (a) if M is behavioral and $s = nth(j, O(M))$, then $w = nth(j, B)$; (b) if M is structural and $s = nth(j, nth(i, LO(M)))$, then w is the waveform for $nth(j, O(nth(i, S(M)))$ determined by $nth(i, B)$.

The *output packet for M determined by B*, denoted as $outp(M, B)$, is defined as follows: (a) if M is behavioral, then $outp(M, B) = B$; (b) if M is structural with $O(M) = (s_1 \, \ldots \, s_n)$, then $outp(M, B) = (w_1 \, \ldots \, w_n)$, where for $1 \leq j \leq n$, w_j is the waveform for s_j determined by B.

Let M be a structural module with $nth(i, LI(M)) = (a_{i1} \, \ldots \, a_{in_i})$. Let p be an input packet and let B be a bundle for M. The i^{th} *input packet determined by p and B*, denoted as $inp(i, M, p, B)$, is the input packet $(w_1 \, \ldots \, w_m)$ for $nth(i, S(M))$, where for $1 \leq j \leq m$, w_j is computed as follows: (a) if s_j is a global input $nth(k, I(M))$, then $w_j = nth(k, p)$; (b) if s_j is a local output, then w_j is the waveform for s_j determined by B.

We may now define *step*. Let p and B be an input packet and a bundle, respectively, for an arbitrary module M, and let $t \in \mathbf{T}$. Then $step(M, p, B, t)$ is the bundle B', defined as follows: (a) if M is behavioral, then $B' = exec(M, p, B, t)$ if p has a new value at t, and $B' = B$ if not; (b) if M is structural with $S(M) = (\mu_1 \, \ldots \, \mu_k)$ and $B = (\beta_1 \, \ldots \, \beta_k)$, then $B' = (\beta'_1 \, \ldots \, \beta'_k)$, where $\beta'_i = step(\mu_i, inp(i, M, p, B), \beta_i, t)$.

Thus, the execution of a structure at time t amounts to the execution of each behavioral component for which the value of some input signal changes at t.

We have the following generalization of Lemma 4.1:

Lemma 5.1 *Let p_1 and p_2 be input packets and let B be a bundle for a module M. Let $t_0 \in \mathbf{T}$. If $hist(p_1, t_0) = hist(p_2, t_0)$, then $step(M, p_1, B, t_0) = step(M, p_2, B, t_0)$.*

The *history* of a structural bundle $(\beta_1 \ldots \beta_k)$ relative to a time t is recursively defined as $hist(B,t) = (hist(\beta_1, t) \ \ldots \ hist(\beta_k, t))$. Lemma 4.2 may be generalized as follows:

Lemma 5.2 *Let p and B be an input packet and a bundle for a module M. For any $t_0 \in \mathbf{T}$, $hist(step(M,p,B,t_0),t_0) = hist(B,t_0)$.*

6 Simulation

Let p and B be an input packet and a bundle for a module M. For any $t \in \mathbf{T}$, we define $t_{next}(t,p,B,M)$ to be the minimum element of the set of all $t' \in \mathbf{T}$ that occur as times of events in the waveforms of p and B and that satisfy $t' > t$, if this set is nonempty; otherwise, $t_{next}(t,p,B,M)$ is undefined.

A simulation of M consists of repeated applications of *step*, which are performed by the function *run*. For $t_0, t_f \in \mathbf{T}$, we define $run(M,p,B,t_0,t_f)$ to be the bundle B' that is computed recursively as follows: Let $t_{next} = t_{next}(t_0,p,B,M)$. If t_{next} is defined and $t_{next} \leq t_f$, then $B' = run(M,p,step(M,p,B,t_{next}),t_{next},t_f)$; otherwise, $B' = B$.

It is not obvious that this is a valid recursive definition, i.e., that it is satisfied by a unique function. This may be established by exhibiting some measure of the arguments that decreases with each recursive call. More precisely, it suffices to define a function *meas* such that under the assumptions imposed on the arguments of *run*,

$$meas(M,p,step(M,p,B,t_{next}),t_{next},t_f) \prec meas(M,p,B,t_0,t_f)$$

with respect to some well-founded order "$\prec$". (In fact, this is the requirement for admissibility of Nqthm function definitions.)

We may construct an appropriate measure based on a function $\phi(M,p,B)$ that computes an upper bound on the delta component of any time object that occurs in any waveform during the course of a simulation. For each signal s of M or any module occurring in M, this function computes the sum of (a) the length of the longest 0-delay path through M starting at s and (b) the largest delta component that occurs in the waveform of p or B that corresponds to s. $\phi(M,p,B)$ is the maximum of these sums. (We omit the actual recursive definition of ϕ, which parallels that of δ-*depth*.)

Now, if $t_0 = (m_i, k_i)$ and $t_f = (m_f, k_f)$, then we define

$$meas(M,p,B,t_0,t_f) = (m_f - m_i, \phi(M,p,B) - k_i).$$

It may be shown that with respect to the lexicographic order "$\prec$" on $\mathbf{N}\times\mathbf{N}$, this function satisfies the property stated above. Note that its definition, and hence that of *run*, ultimately depends on the assumption that M is δ-acyclic.

The function *meas* provides an induction scheme for deriving properties of *run*. The following, for example, is proved by induction as an immediate consequence of Lemma 5.2:

Lemma 6.1 *Let p and B be an input packet and a bundle for a module M. For any $t_0, t_f \in \mathbf{T}$, $hist(run(M,p,B,t_0,t_f),t_0) = hist(B,t_0)$.*

The next lemma, similarly proved by induction, provides for the decomposition of a simulation interval:

Lemma 6.2 *If p and B are an input packet and a bundle for a module M, and $t_0 \leq t' \leq t_f$, then $run(M, p, B, t_0, t_f) = run(M, p, run(M, p, B, t_0, t'), t', t_f)$.*

Another property of run that is important in the analysis of circuit behavior is the following basic result, which describes the behavior of a structural module in terms of that of its components. It is interesting that its proof requires the two properties of $step$ that are stated in Lemmas 5.1 and 5.2, namely that module execution is neither predictive (with respect to input) nor retroactive (with respect to output).

Lemma 6.3 *Let p and $A = (\alpha_1 \ldots \alpha_k)$ be an input packet and a bundle for a structural module M with $S(M) = (\mu_1 \ldots \mu_k)$. Let $t_0, t_1 \in \mathbf{T}$ and $B = (\beta_1 \ldots \beta_k) = run(M, p, A, , t_0, t_1)$. Then $\beta_i = run(\mu_i, b_i, \alpha_i, t_0, t_1)$, where $b_i = inp(i, M, p, B)$, $i = 1, \ldots, k$.*

Proof: Let $A' = (\alpha'_1 \ldots \alpha'_k) = step(M, p, A, t')$, where $t' = t_{next}(t_0, p, A, M)$. Then by definition of $step$, $\alpha'_i = step(\mu_i, a_i, \alpha_i, t')$, where $a_i = inp(i, M, p, A)$, and by definition of run, $B = run(M, p, A', t', t_1)$. By induction, we may assume that $\beta_i = run(\mu_i, b_i, \alpha'_i, t', t_1)$.

It follows from Lemmas 5.2 and 6.1 that $hist(A, t') = hist(B, t')$. Consequently, $hist(a_i, t') = hist(b_i, t')$. By Lemma 5.1, $\alpha'_i = step(\mu_i, b_i, \alpha_i, t')$. Thus, we have $\beta_i = run(\mu_i, b_i, step(\mu_i, b_i, \alpha_i, t'), t', t_1)$.

Let $t'' = t_{next}(t_0, b_i, \alpha_i, \mu_i)$. Clearly, if t'' is defined, then $t'' \geq t'$. If $t'' = t'$, then

$$\begin{aligned} run(\mu_i, b_i, \alpha_i, t_0, t_1) &= run(\mu_i, b_i, step(\mu_i, b_i, \alpha_i, t''), t'', t_1) \\ &= run(\mu_i, b_i, step(\mu_i, b_i, \alpha_i, t'), t', t_1) \quad = \quad \beta_i. \end{aligned}$$

In the remaining case,

$$\begin{aligned} run(\mu_i, b_i, \alpha_i, t_0, t_1) &= run(\mu_i, b_i, \alpha_i, t', t_1) \\ &= run(\mu_i, b_i, step(\mu_i, b_i, \alpha_i, t'), t', t_1) \quad = \quad \beta_i. \ \Box \end{aligned}$$

The definition of our top-level simulation function sim depends on run as well as a function $init$, which generates an initial bundle from a module and an input packet. First, for a given module M, we define the bundle $B_0(M)$:

(1) If M is behavioral, then $B_0(M)$ is the output packet $(w_0 \ldots w_0)$ for M, where $w_0 = ((\mathcal{F}, 0))$.

(2) If M is structural and $S(M) = (\mu_1 \ldots \mu_k)$, then $B_0(M) = (B_0(\mu_1) \ldots B_0(\mu_k))$.

Thus, every waveform of $B_0(M)$ is the trivial w_0, which has the constant value $\hat{w}_0(t) = \mathcal{F}$. Prior to simulation, each of these waveforms is updated by executing every behavioral component of M. The result is the bundle $init(M, p)$, defined as follows:

(1) If M is behavioral, then $init(M, p) = exec(M, p, B_0(M), 0)$;

(2) If M is structural with $S(M) = (\mu_1 \ldots \mu_k)$, then
$init(M, p) = (init(\mu_1, inp(1, M, p, B_0(M))) \ldots init(\mu_k, inp(k, M, p, B_0(M))))$.

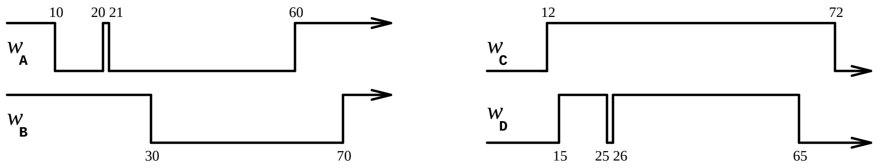

Figure 3: Simulation of m

Now, given an input packet p for M and a time object t, we define

$$sim(M, p, t) = run(M, p, init(M, p), 0, t).$$

We note the following restatements of Lemmas 6.2 and 6.3:

Lemma 6.4 *If p is an input packet for a module M, and $t_1 \leq t_2$, then $sim(M, p, t_2) = run(M, p, sim(M, p, t_1), t_1, t_2)$.*

Lemma 6.5 *Let p be an input packet for a structural module M with $S(M) = (\mu_1 \ldots \mu_k)$. Let $t \in \mathbf{T}$ and $B = (\beta_1 \ \ldots \ \beta_k) = sim(M, p, t)$. Then $\beta_i = sim(\mu_i, b_i, t)$, where $b_i = inp(i, M, p, B)$, $i = 1, \ldots, k$.*

As a simple example, a simulation of the primitive module m is illustrated in Figure 3. The waveforms corresponding to the inputs A and B are

$$w_{\mathtt{A}} = ((\mathcal{T}, 60000)\ (\mathcal{F}, 21000)\ (\mathcal{T}, 20000)\ (\mathcal{F}, 10000)\ (\mathcal{T}, 0))$$

and

$$w_{\mathtt{B}} = ((\mathcal{T}, 70000)\ (\mathcal{F}, 30000)\ (\mathcal{T}, 0)),$$

respectively. These are shown along with the waveforms

$$w_{\mathtt{C}} = (((\mathcal{F}, 72000)\ (\mathcal{T}, 12000)\ (\mathcal{F}, 0)))$$

and

$$w_{\mathtt{D}} = ((\mathcal{F}, 65000)\ (\mathcal{T}, 26000)\ (\mathcal{F}, 25000)\ (\mathcal{T}, 15000)\ (\mathcal{F}, 0))$$

of the output $sim(\mathtt{m}, (w_{\mathtt{A}} w_{\mathtt{B}}), 80000) = (w_{\mathtt{C}}\ w_{\mathtt{D}})$.

This example exhibits a fundamental difference between transport and inertial delay: an input pulse of duration less than the delay, as occurs in $w_{\mathtt{A}}$, is not reflected in an inertial output.

All of the simulation results that we report herein were produced by the Nqthm implementation of *sim* and have been matched with the output of the corresponding Vantage simulations of the VHDL translations of these modules. One further observation is warranted, however, in support of the claim that our language definition adheres to the VHDL standard [3]. There is an apparent discrepancy between the definition of *sim* and the standard: in our language, each output waveform of a behavioral module is updated whenever there is a change in *any* input value. In VHDL, on the other hand, in the absence of any instruction to the contrary (i.e., an explicit "sensitivity list"), a signal's waveform is updated only in response to changes in those inputs on which the signal is functionally dependent.

Consider, for example, the output `D` of the module `m`. The VHDL code corresponding to this signal (Figure 1) is executed only in response to events of the input waveform $w_{\mathtt{A}}$. However, according to our definitions of *exec* and *step*, its waveform is also updated whenever the value of `B` changes, e.g., at time 30000 in our example.

Nonetheless, as illustrated in Figure 3, the behavior of this output signal is completely independent from that of `B`, in accordance with the VHDL standard. In order to understand this, consider the waveform w that represents this signal before the execution of `m` at time 21000. The updated waveform after this execution is $w' = transport(w, \mathcal{T}, 26000)$. Although w' is further updated when the value of `B` changes at 30000, the value of `(NOT1 A)` remains $\mathcal{T}$, and hence, by Lemma 3.2, the resulting waveform is $transport(w', \mathcal{T}, 35000) = w'$.

The above argument is based on the simple observation that at the time of any change in input during a simulation of a behavioral module, the output packet is the result of executing the module at that time. In fact, an interesting property of our simulator is that this holds true even when there is no input change, i.e, regardless of whether the execution actually occurs:

Lemma 6.6 *Let p be an input packet for a behavioral module M, let $t \in \mathbf{T}$, and let $B = sim(M, p, t)$. Then $B = exec(M, p, B, t)$.*

Proof: It is easily shown by induction and Lemma 4.3, that if $B_0 = exec(M, p, B_0, t_0)$ and $B_1 = run(M, p, B_0, t_0, t_1)$, then $B_1 = exec(M, p, B_1, t_1)$. The lemma is an instance of this result, with $t_0 = t$, $B_0 = init(M, p)$, $t_1 = t$, and $B_1 = B$. □

7 Analysis of Circuit Behavior

We begin our analysis of circuit behavior by considering the outputs of a behavioral module. In the case of transport delay, it may be shown that the value of an output with delay d, at time $t + d$, is simply the combinational value determined by the input values at time t. For inertial delay, a similar prediction may be made only if the combinational value remains stable over an interval of length d. We have the following general characterization:

Lemma 7.1 *Let $s = nth(j, O(M))$ be the j^{th} output of a behavioral module M, let $d = nth(j, D(M))$ be the corresponding delay, and let $w = nth(j, sim(M, p, t_f))$.*

Assume that for all $t \in [t_1, t_2)$, the combinational value of s w.r.t. $\hat{p}(t)$ is v, where $t_1 + d \leq t_2$ and $t_1 \leq t_f$. Then for all $t \in [t_1 + d, t_2 + d)$, $\hat{w}(t) = v$.

Proof: Let $p_1 = sim(M, p, t_1)$. Then according to Lemma 6.6, $p_1 = exec(M, p, p_1, t_1)$. It follows from Lemma 3.1(a) that the value of $nth(j, p_1)$ is v for all $t \geq t_1 + d$.

We claim that if p' is any output packet for M such that $nth(j, p')$ has value v throughout $[t_1 + d, t_2 + d)$, then so does $nth(j, run(M, p, p', t', t_f))$, for any $t' \geq t_1$. Once this claim is proved, the lemma will follow from Lemma 6.4 upon substituting p_1 and t_1 for p' and t'.

The claim is proved by induction. It suffices to show that if p has a new value at $t'' = t_{next}(t', p, p', M)$, and $p'' = exec(M, p, p', t'')$, then $nth(j, p'')$ has value v throughout $[t_1 + d, t_2 + d)$.

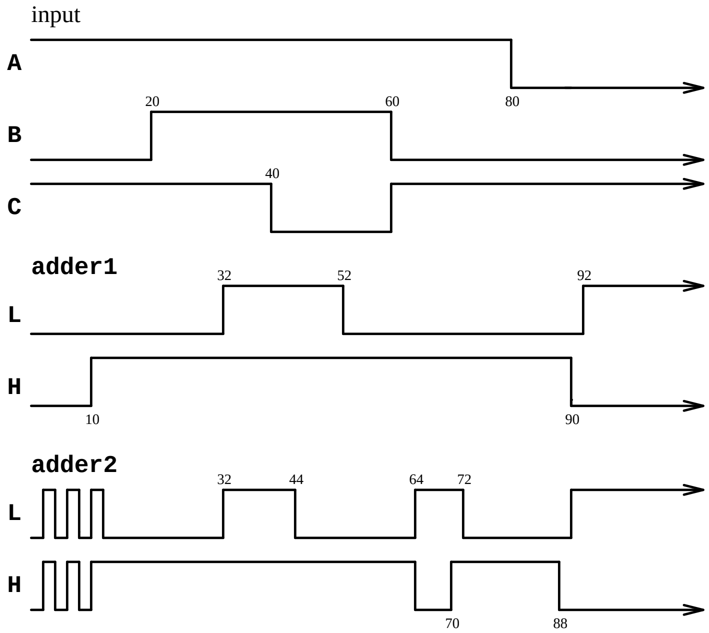

Figure 4: Simulation of `adder1` and `adder2`

If $t'' \geq t_2$, then the desired result follows from Lemma 3.1(c). Thus, we may assume $t'' < t_2$ and hence, the combinational value of s w.r.t. $\hat{p}(t'')$ is v. In this case, $nth(j, p'')$ has value v on $[t_1 + d, t'' + d)$ by Lemma 3.1(c), and on $[t'' + d, t_2 + d)$ by Lemma 3.1(a). $\Box$

In Figure 4, we illustrate Lemma 7.1 with a simulation of the behavioral module `adder1`. We also compare the result of this simulation with the corresponding output of the combinational structure `adder2`. Note, for example, that the first output `L` of `adder1`, with corresponding term `(XOR3 A B C)`, has the combinational value $\mathcal{F}$ throughout the interval from 40000 to 80000, and thus, since its delay is 12000, the actual value of the signal is $\mathcal{F}$ from 52000 to 92000. The behavior of `adder2` is somewhat more complicated, although for stable inputs, the two modules eventually produce the same values.

It is possible to establish similar behavioral specifications of combinational (i.e., acyclic) structures, such as `adder2`. The hypothesis of Lemma 7.1 must be strengthened, however, to require that *each input* maintain a constant value over a fixed interval, the length of which is the total delay along the longest path that connects the output to an input. The conclusion applies to the interval that results from shifting the lower and upper endpoints of this interval by the delays along the longest and shortest paths, respectively. In the case of `adder2`, these delays are 12000 and 4000 for output `L`, and 10000 and 4000 for output `H`. Thus, in the simulation displayed in Figure 4, for example, the input is constant over the interval $[20000, 40000)$, and hence the computed value of `L` is valid on the interval $[32000, 44000)$.

Proposition 7.1 *Let $p = (w_{\mathtt{A}}\, w_{\mathtt{B}}\, w_{\mathtt{C}})$ be an input packet for* `adder2`*, let $t_1, t_2, t_f \in \mathbf{T}$, and let $sim(\texttt{adder2}, p, t_f) = ((w_{\mathtt{T1}})\,(w_{\mathtt{T2}})\,(w_{\mathtt{T3}})\,(w_{\mathtt{T4}})\,(w_{\mathtt{T5}})\,(w_{\mathtt{T6}})\,(w_{\mathtt{T7}})\,(w_{\mathtt{H}})\,(w_{\mathtt{L}}))$. Let $t_1 \oplus 12000 \le t_2 \le t_f$. Suppose that for all $t \in [t_1, t_2)$, $\hat{w}_{\mathtt{A}}(t) = v_{\mathtt{A}}$, $\hat{w}_{\mathtt{B}}(t) = v_{\mathtt{B}}$, and $\hat{w}_{\mathtt{C}}(t) = v_{\mathtt{C}}$. Then for all $t \in [t_1 \oplus 12000, t_2 \oplus 4000)$, $\hat{w}_{\mathtt{L}}(t) = xor3(v_{\mathtt{A}}, v_{\mathtt{B}}, v_{\mathtt{C}})$; for all $t \in [t_1 \oplus 10000, t_2 \oplus 4000)$, $\hat{w}_{\mathtt{H}}(t) = or2(and2(v_{\mathtt{A}}, or2(v_{\mathtt{B}}, v_{\mathtt{C}})), and2(v_{\mathtt{B}}, v_{\mathtt{C}}))$.*

Proof: The proof is straightforward, involving successive applications of Lemma 7.1 to the signals along the various paths that connect inputs to outputs. We shall concentrate on $w_{\mathtt{L}}$; the analysis of $w_{\mathtt{H}}$ is similar. We may assume $t_1 \oplus 8000 \le t_2$. Applying Lemma 6.5 with $i = 1$, we have $(w_{\mathtt{T1}}) = sim(\texttt{nand2}, (w_{\mathtt{A}}\, w_{\mathtt{B}}), t_f)$. By Lemma 7.1, for $t \in [t_1 \oplus 2000, t_2 \oplus 2000)$, $\hat{w}_{\mathtt{T1}}(t) = nand2(v_{\mathtt{A}}, v_{\mathtt{B}})$. We abbreviate this value as $v_{\mathtt{T1}}$. Similar applications of the same two lemmas yield the following:
$\hat{w}_{\mathtt{T2}}(t) = nand2(v_{\mathtt{A}}, v_{\mathtt{T1}}) = v_{\mathtt{T2}}$ for $t \in [t_1 \oplus 4000, t_2 \oplus 2000)$;;
$\hat{w}_{\mathtt{T3}}(t) = nand2(v_{\mathtt{T1}}, v_{\mathtt{B}}) = v_{\mathtt{T3}}$ for $t \in [t_1 \oplus 4000, t_2 \oplus 2000)$;
$\hat{w}_{\mathtt{T4}}(t) = nand2(v_{\mathtt{T2}}, v_{\mathtt{T3}}) = v_{\mathtt{T4}}$ for $t \in [t_1 \oplus 6000, t_2 \oplus 4000)$;
$\hat{w}_{\mathtt{T5}}(t) = nand2(v_{\mathtt{T4}}, v_{\mathtt{C}}) = v_{\mathtt{T5}}$ for $t \in [t_1 \oplus 8000, t_2 \oplus 2000)$;
$\hat{w}_{\mathtt{T6}}(t) = nand2(v_{\mathtt{T4}}, v_{\mathtt{T5}}) = v_{\mathtt{T6}}$ for $t \in [t_1 \oplus 10000, t_2 \oplus 4000)$;
$\hat{w}_{\mathtt{T7}}(t) = nand2(v_{\mathtt{T5}}, v_{\mathtt{C}}) = v_{\mathtt{T7}}$ for $t \in [t_1 \oplus 10000, t_2 \oplus 2000)$;
$\hat{w}_{\mathtt{L}}(t) = nand2(v_{\mathtt{T6}}, v_{\mathtt{T7}})$ for $t \in [t_1 \oplus 12000, t_2 \oplus 4000)$.
Expansion of the abbreviations $v_{\mathtt{T1}}, \ldots, v_{\mathtt{T7}}$ yields an expression for $nand2(v_{\mathtt{T6}}, v_{\mathtt{T7}})$ that may be shown to be tautologically equivalent to $xor3(v_{\mathtt{A}}, v_{\mathtt{B}}, v_{\mathtt{C}})$. □

Similar reasoning may be applied to state-holding (i.e., cyclic) circuits, such as the flip-flop `dff` of Section 5. The intended behavior of this device depends on various assumptions concerning the regularity of its inputs:

(1) The `CLK` input is expected to behave as a regular clock pulse, with minimum high and low times (intervals during which the values $\mathcal{T}$ and $\mathcal{F}$, respectively, must be maintained) of 4 ns. and 6 ns.

(2) The value of the `D` input may not change too close to a *rising edge* (i.e., a time when `CLK` assumes the value $\mathcal{T}$). Specifically, a *setup time* (interval of stability preceding a rising edge) of 4 ns. and a *hold time* (interval of stability following a rising edge) of 2 ns. must be respected.

Under these conditions, an output event can occur only during a short interval following a rising edge. The value assumed by `Q` following a rising edge is the value of `D` at the edge, and the new value of `QN` is its negation.

In the sample simulation shown in Fig. 5, rising edges occur at multiples of 20 ns. During the first cycle, the behavior of `Q` is erratic as the circuit reaches a settled state. Over the next two cycles, `Q` behaves as described above. The behavior becomes erratic again during the cycle following the rising edge at 60 ns, because a change in `D` occurs too close to the edge.

A precise specification of `dff` is given by the following. Its proof is an elaboration of the informal argument found in [7]:

Proposition 7.2 *Let $p = (w_{\mathtt{CLK}}\, w_{\mathtt{D}})$ be an input packet for* `dff`*. Assume that*

$$\hat{w}_{\mathtt{CLK}}(t) = \begin{cases} \mathcal{F} & \text{for } t \in [t_-, t_+) \cup [t'_-, t'_+) \\ \mathcal{T} & \text{for } t \in [t_+, t'_-), \end{cases}$$

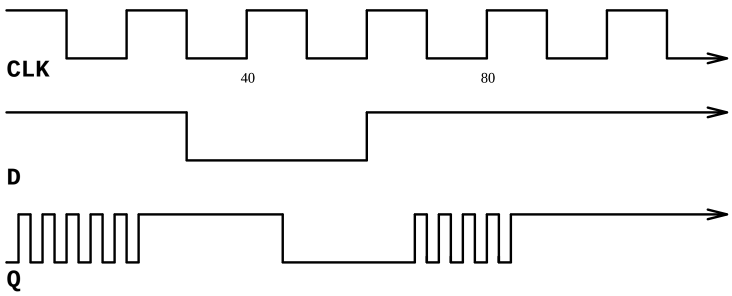

Figure 5: Simulation of dff

where $t_- \oplus 6000 \leq t_+$, $t_+ \oplus 4000 \leq t'_-$, *and* $t'_- \oplus 2000 \leq t'_+$. *Assume also that*

$$\hat{w}_{\mathtt{D}}(t) = v \text{ for } t \in [t_d, t_+ \oplus 2000),$$

where $t_d \oplus 4000 \leq t_+$. *Let* $sim(\mathtt{dff}, p, t_f) = ((w_{\mathtt{A1}})\ (w_{\mathtt{B1}})\ (w_{\mathtt{A2}})\ (w_{\mathtt{B2}})\ (w_{\mathtt{Q}})\ (w_{\mathtt{QN}}))$, *where* $t_f \geq t'_+$. *Then* $\hat{w}_{\mathtt{Q}}(t) = v$ *and* $\hat{w}_{\mathtt{QN}}(t) = not1(v)$ *for* $t \in [t_+ \oplus 6000, t'_+ \oplus 4000)$.

Proof: Applying Lemmas 7.1 and 6.5, we have $\hat{w}_{\mathtt{A2}}(t) = \hat{w}_{\mathtt{B1}}(t) = \mathcal{T}$ for $t \in [t_- \oplus 2000, t_+ \oplus 2000)$ and for $t \in [t'_- \oplus 2000, t'_+ \oplus 2000)$. Let $t_1 = max(t_- \oplus 2000, t_d)$. Then $t_1 \oplus 4000 \leq t_+$. Applying the same two lemmas again, we have $\hat{w}_{\mathtt{B2}}(t) = not1(v)$ for $t \in [t_1 \oplus 2000, t_+ \oplus 4000)$, and hence $\hat{w}_{\mathtt{A1}}(t) = v$ for $t \in [t_1 \oplus 4000, t_+ \oplus 4000)$.

We shall consider the case $v = \mathcal{F}$; the case $v = \mathcal{T}$ is similar. In this case, $\hat{w}_{\mathtt{B1}}(t) = \mathcal{T}$ for $t \in [t_1 \oplus 6000, t_+ \oplus 6000)$, and $\hat{w}_{\mathtt{A2}}(t) = \mathcal{F}$ for $t \in t_+ \oplus 2000, t_+ \oplus 6000)$.

Let t_2 be the least time such that $t_2 > t_+ \oplus 2000$ and some waveform in the set $\{w_{\mathtt{A1}}, w_{\mathtt{B1}}, w_{\mathtt{A2}}, w_{\mathtt{B2}}\}$ assumes a new value at t_2. Then $\hat{w}_{\mathtt{A1}}(t) = \hat{w}_{\mathtt{A2}}(t) = \mathcal{F}$ and $\hat{w}_{\mathtt{B1}}(t) = \hat{w}_{\mathtt{B2}}(t) = \mathcal{T}$ for $t \in [t_+ \oplus 2000, t_2)$. Since $t_2 \geq t_+ \oplus 4000$, it follows that $\hat{w}_{\mathtt{B1}}(t) = \hat{w}_{\mathtt{B2}}(t) = \mathcal{T}$ and $\hat{w}_{\mathtt{A1}}(t) = \mathcal{F}$ for $t \in [t_+ \oplus 4000, t_2 \oplus 4000)$. Similarly, $\hat{w}_{\mathtt{A2}}(t) = \mathcal{F}$ for $t \in [t_+ \oplus 4000, min(t_2 \oplus 4000, t'_- \oplus 2000))$. Thus, only $w_{\mathtt{A2}}$ can possibly assume a new value at t_2, and this requires that $t_2 \geq t'_- \oplus 2000$.

Thus, $\hat{w}_{\mathtt{B1}}(t) = \mathcal{T}$ and $\hat{w}_{\mathtt{A2}}(t) = \mathtt{F}$ for $t \in [t_+ \oplus 2000, t'_- \oplus 2000)$. It follows that $\hat{w}_{\mathtt{QN}}(t) = \mathcal{T}$ for $t \in [t_+ \oplus 4000, t'_- \oplus 4000)$, and hence $\hat{w}_{\mathtt{Q}}(t) = \mathcal{F}$ for $t \in [t_+ \oplus 6000, t'_- \oplus 6000)$. Let t_3 be the least time such that $t_3 > t_+ \oplus 6000$ and either $w_{\mathtt{Q}}$ or $w_{\mathtt{QN}}$ assumes a new value at t_3. By an argument similar to the above, it is easily shown that $t_3 \geq t'_+ \oplus 4000$. □

8 Discussion

The examples treated in the preceding section illustrate how the informal reasoning that is commonly employed by hardware designers in the analysis of circuits may be precisely formulated in our VHDL formalization. In a sequel to this paper [5], we further develop our proof methodology and systematically apply it to several significant classes of circuits. The generalization of Proposition 7.1 to arbitrary combinational circuits, for example, is straightforward. We also treat an interesting class of synchronous sequential circuits, based on a variation of the d-flip-flop and its characterization given by Lemma 7.2. As an application, we present a proof of a behavioral specification of a circuit consisting of two independently clocked processors that achieve asynchronous communication by means of a well known protocol.

We have attempted to capture accurately some of the basic semantic notions of VHDL. One consequence of our commitment to adhere to the VHDL standard is a faithful formalization of the "delta delay" mechanism, which provides for the simulation of zero-delay devices. Our exposition could be shortened significantly if we were to require all delays to be positive. In particular, the definitions of *time object*, *structural module*, and the simulator itself are greatly complicated by the delta delay feature.

On the other hand, in order to simplify the analysis, we have concentrated thus far on a subset of the language that is so limited as to be of little practical value for hardware engineers. In the future, we plan to extend this subset to include features that will allow more realistic designs: bidirectional ports, signal resolution, general waveform transactions, complex signal types, and sequential statements. We hope to show that our specification and verification methods may be effectively applied to these designs.

References

[1] Boyer, R. S. and Moore, J, *A Computational Logic Handbook*, Academic Press, Boston, 1988.

[2] Damm, W., A Formal Semantics for VHDL based on Interpreted Petri Nets, Technical Report, University of Oldenburg, 1992.

[3] Institute of Electrical and Electronic Engineers, *Draft Standard VHDL Language Reference Manual*, 1993.

[4] Kaufmann, M., A Translator from an HDL of David Russinoff to VHDL, Internal Note 278, Computational Logic, Inc., July 1993.

[5] Russinoff, D., "Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits", to appear in *Specification and Validation Methods*, edited by Egon Börger, Oxford University Press, 1994; also available as Technical Report 99, Computational Logic, Inc., May 1994.

[6] Sanchez, L. and Kloos, C. D., "Functional Description of VHDL", in *Segundo Congreso de Programacion Declarativa PRODE 93*, Spain, September 1993.

[7] Taub, H. and Schilling, D., *Digital Integrated Electronics*, McGraw-Hill, New York, 1977.

[8] Van Tassel, J., A Formalization of the VHDL Simulation Cycle, Technical Report 249, University of Cambridge Computer Laboratory, June 1992.