A Semantic Model of VHDL for Validating Rewriting Algebras*

Sheetanshu L. Pandey, Kothanda R. Subramanian and *Philip A. Wilsey*
Computer Architecture Design Lab, Dept. of ECECS, PO Box 210030
Univ. of Cincinnati, Cincinnati, OH 45221–0030 phil.wilsey@uc.edu

Abstract

This paper presents a formal model of the dynamic semantics of VHDL using interval temporal logic. The model uses a declarative style that provides a semantic definition of VHDL independent of the VHDL simulation cycle. Therefore, the model can be used as a platform for comparing alternative and possibly more efficient algorithms for simulating VHDL. Furthermore, optimization techniques for improving the performance of VHDL simulators can be validated against this model. To support this claim, we present a proof asserting the validity of process-folding. In contrast to past efforts that concentrate only on design verification, this model is also oriented towards CAD tool optimization. The model is comprehensive and characterizes most of the important features of elaborated VHDL.

1 Introduction

Hardware Description Languages (HDLs) are used extensively to specify, document, and verify hardware designs prior to fabrication. Apart from simulation and automated synthesis, formal proofs of equivalences between behavioral level specifications and implementation level specifications in HDLs are used to establish design correctness. Such proofs usually require a formal mathematical representation of the semantics of the HDL in consideration. In this paper, we are concerned with such a representation of VHDL which has come to be accepted as a standard HDL. Furthermore, the VHDL Language Reference Manual (LRM) [8] defines the semantics of VHDL in informal prose form which is sometimes ambiguous. Therefore a formal, well understood, and complete specification of the semantics for the language is necessary.

Most of the investigations into formal definitions of VHDL concentrate on defining the *dynamic semantics* [2, 3, 4, 5, 6, 11, 12] though some have in addition addressed the issues related to the *static semantics* [13]. The formal definitions developed so far characterize only a small subset of VHDL and rarely address issues related to the more complex features such as shared variables, guarded signals, component interconnections, and so on. Investigations into applications of formal methods in verification concentrate on simply proving equivalence between two design specifications at different levels of abstraction. Typically, a high level specification of a circuit is taken and its equivalence to a gate level specification of the same circuit is proved. However, no attempts towards investigating the application of formal methods in CAD tool optimization for VHDL have been made. In fact, the nature of the formal models developed so far do not support such investigations simply because they define VHDL semantics in terms of the simulation cycle prescribed by the LRM. They fail to separate the notion of the *meaning* of VHDL from the notion of *executing* a VHDL description by a specific algorithm.

The aim of this paper is to (1) define a comprehensive formal model of VHDL covering most of the important aspects of the dynamic semantics of VHDL, (2) define a model that views VHDL semantics in terms of the net effect of its evaluation rather than characterizing any particular method of simulation, (3) demonstrate the utility of the model in validating transformation techniques that may help in improving the performance of VHDL simulators and (4) capture the timing aspects of VHDL by using interval logic as the underlying specification language. The work presented herein is a part of a larger project to characterize the static and dynamic semantics of VHDL [14, 15, 16, 17] though this paper concentrates on the dynamic semantics.

The remainder of the paper is organized as follows. Section 2 reviews some related work. Section 3 presents the rudiments of Allen's interval temporal logic that is used as a framework for specifying the semantics of VHDL. Section 4 develops the mathematical representation of the static structures of VHDL and its related theories. These theories define a reduced form representation of VHDL on which the dynamic model (the topic of this paper) is based. Section 5 presents the dynamic semantics of elaborated VHDL. Section 6 provides a proof of the validity of process-folding, an optimization technique for parallel VHDL simulators. Finally, Section 7 contains some concluding remarks and identifies areas for future research.

2 Related Work

Since the late 1980s, there has been a growing interest in the application of formal methods to VHDL. Usually, a small subset of VHDL is identified and an operational or denotational characterization of the simulation cycle is provided.

Van Tassel [13] provides an operational semantics for a very

*This research was supported in part by the Advanced Research Projects Agency and monitored by the Air Force Wright Laboratory under contract number F33615–93–C–1315.

1089-6503/96 $5.00 © 1996 IEEE
Proceedings of EUROMICRO-22

limited subset of VHDL 87 and then embeds the semantics in an HOL mechanical proof assistant in order to prove properties of the VHDL descriptions and to show equivalences between two VHDL descriptions. Proofs for circuits such as parity checkers and flip flops are demonstrated. Only signals of kind boolean are characterized and most sequential statements are ignored. Gordon [7] uses pure logic to describe circuits directly rather than using a language such as VHDL. He demonstrates proofs of equivalences between implementations and their specifications for generic n-bit adders, sequential multipliers and flip-flops using the HOL proof assistant.

Davis [6] and Barton [2] provide a denotational specification of the simulation cycle for a limited subset of VHDL 87. No proof methods are investigated by these authors. Damm *et al* [5] use Interpreted Petri Nets to model designs specified in VHDL 87. The characterization does not include multiple entity-architecture pairs, delta delays and structural descriptions. Signals and variables of type bit only are considered. Olcoz and Colom [11] use Colored Petri Nets to model fully elaborated VHDL 87. Tools have been developed to automatically generate Petri Nets from VHDL descriptions and to perform analysis on these nets such as testing for *liveness* and *deadlock* using Petri Net theory.

Borger *et al* [3] provide a comprehensive semantics for VHDL 93 in terms of an operational characterization of VHDL using Distributed Evolving Algebra machines. Most of the new features of VHDL 93 such as postponed *process* statements, shared variables and pulse rejection limits are covered in this work. However, issues related to guarded signals and proof methodologies are not discussed. Borrione *et al* [4] define the semantics of VHDL in terms of a stream functional model and embed the semantics in the Prevail proof subsystem to prove equivalences between two VHDL descriptions, one at the specification level and one at the implementation level.

The semantic models described above are not broad enough in their scope with respect to the features of VHDL they characterize. On many occasions critical issues such as zero delay assignments, guarded signals and component interconnections have been ignored. Semantics of *exit* and *next* statements of VHDL are not available. Most of the formal definitions are for VHDL 87 and therefore do not characterize newer features such as postponed *process* statements and pulse-rejection limits.

Furthermore, not much progress has been made in validating transformation techniques on VHDL descriptions for CAD tool optimization. We will demonstrate a formal proof validating *process-folding* [10] which is an optimization technique for parallel VHDL simulators. It is also argued that since we do not bind our formal model to the LRM *simulation cycle*, different and possibly more efficient simulation algorithms could be investigated and validated against the model.

The notation used in this paper will be that of first order logic. An axiom of the form $\forall$ x: Q(x), P(x) must be read as: for all x such that Q(x) is satisfied, P(x) holds. Sometimes, for the sake of brevity an axiom of the form $\forall$ x: x $\in$ A, P(x) will be written as $\forall$ x $\in$ A, P(x). If no pre-conditions exist, then the axiom $\forall$ x: P(x) must be read as: for all x, P(x) holds. Usual notations for building sets and sequences will be used. References to the IEEE Standard VHDL LRM (Std 1076-1993) will be of the form (§ s, ¶ p, Ł l) where, s is the section number, p the page number and l the line number.

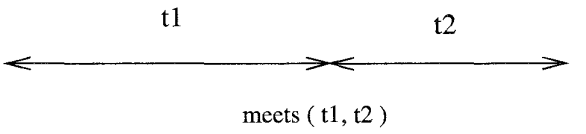

Figure 1. The Relation Meets

3 Notion of Time Intervals in VHDL

The underlying logic used in this work is Allen's interval temporal logic [1]. It organizes a universe by time intervals, relations between time intervals and by binding actions (or assertions) to time intervals. VHDL behavior can be defined in terms of time intervals and a set of actions that are performed in these time intervals. Interval temporal logic is axiomatized in terms of the single relation meets. A time interval t_1 is said to meet t_2 if t_1 is before t_2 such that there is no time interval separating them and if they do not overlap in any way (Figure 1). All other relations between time intervals are expressed in terms of meets. For example, the relation during can be axiomatized in terms of meets as

$$\text{during}(t_1, t_2) \equiv \exists\ t_3, t_4: t_3 + t_1 + t_4 = t_2.$$

There are six relations (depicted in Figure 2) other than meets that can hold between any two intervals. Informally, these relations can be interpreted as follows:

equals(t_1, t_2): t_1 and t_2 represent the same time interval,

before(t_1, t_2): t_1 finishes before t_2 begins and there is an interval t_3 that is between t_1 and t_2,

overlaps(t_1, t_2): t_1 begins before t_2 begins, t_2 begins before t_1 finishes, and t_1 finishes before t_2 finishes,

during(t_1, t_2): t_1 is fully contained within t_2,

starts(t_1, t_2): t_1 and t_2 begin together, but t_1 finishes before t_2 finishes, and

finishes(t_1, t_2): t_1 starts after t_2 starts, but they finish together.

Although the notions of time points and time moments are available in the logic, they are not required in this work. The critical issue is that for any given interval, t, there exists another interval for each predicate of the logic such that the predicate holds. That is, given an interval t, there is an interval that meets t, that equals t, that starts with t, and so on. Additionally, four other predicates that will be used in this work are defined below:

$$\text{in}(t_1, t_2) \Leftrightarrow \text{during}(t_1,t_2) \vee \text{starts}(t_1, t_2) \vee \text{finishes}(t_1, t_2),$$
$$\text{begins}(t_1, t_2) \Leftrightarrow \text{starts}(t_1, t_2) \vee \text{starts}(t_2, t_1) \vee \text{equals}(t_2, t_1),$$
$$\text{ends}(t_1, t_2) \Leftrightarrow \text{finishes}(t_1, t_2) \vee \text{finishes}(t_2, t_1) \vee \text{equals}(t_2, t_1),$$
$$\text{endsafter}(t_1, t_2) \Leftrightarrow \text{before}(t_2, t_1) \vee \text{overlaps}(t_2, t_1) \vee \text{meets}(t_2, t_1) \vee \text{starts}(t_2, t_1) \vee \text{during}(t_2, t_1).$$

Informally, the predicate in states that t_1 lies within the boundaries of t_2. The predicate begins states that t_1 and t_2 begin together. The predicate ends states that t_1 and t_2 finish together. The predicate endsafter asserts that t_1 ends after t_2 ends.

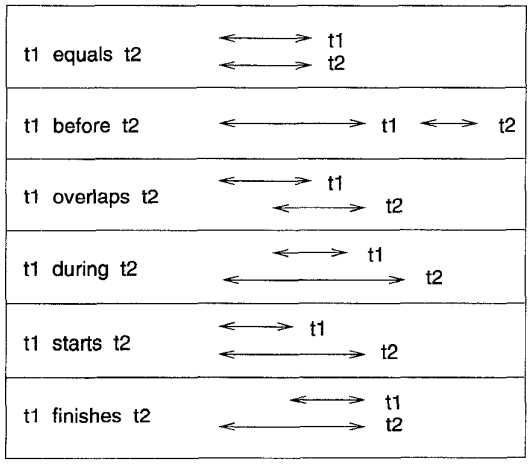

Figure 2. Relationships of Time Intervals

4 The Static Semantics

The dynamic semantic model (referred to as the *Dynamic Model*) is based on a mathematical representation of the static structures of VHDL such as signals, ports, sequential statements and so on. This representation is called the *Static Model*. In this section we present a brief overview of the Static Model and its affiliated theories. For a more detailed description of the Static Model and its application in CAD tool optimization, interested readers are urged to refer to [16] and the working document [15]. The Static Model uses standard mathematical constructs such as set, tuples and sequences to represent VHDL static structures. For example, the set of all declared signals[1] in a VHDL description is represented as the set Signals. Any declared signal sig, sig $\in$ Signals, is represented as the 6-tuple

sig = $\langle$ name, res-fn, data-type, signal-kind,
disconnection-delay, initial-value $\rangle$
signal-kind = [`register`| `bus`| `discrete`].

Informally, every declared signal has associated with it, a unique name, a resolution function res-fn mapping a set of values corresponding to multiple sources into a single value, a data-type, a disconnection-delay (specifying the delay involved in a null transaction for guarded signals) and an initial-value. If the declared signal is declared with a signal-kind then it is a guarded signal of the same kind, else it is an unguarded signal of kind `discrete`.

Throughout the remainder of this paper, other tuple definitions will be introduced as they are required. The affiliated theories include axioms defining *well-formedness* of VHDL descriptions. For example, the fact that a *process* statement with a sensitivity list cannot have a *wait* statement in it appears as an axiom in the rules for well-formedness. In addition, a *reduction algebra* has been developed, which defines a reduced form representation of VHDL. The reduction algebra is based on certain equivalences between static structures of VHDL that are defined in the LRM. Informally, these equivalences are: (i) Every concurrent statement in VHDL has an equivalent *process* statement representation, (ii) A *process* statement with a sensitivity list has an equivalent *process* statement with no sensitivity list and an added *wait* statement sensitive to the same signals as the former *process* statement and (iii) A *signal assignment* statement with multiple waveforms is equivalent to a sequence of the same *signal assignment* statements each with a single waveform as long as the order of the waveforms is preserved. Therefore the reduced form representation of VHDL does not include a representation for concurrent statements (except the *process* statement) and *signal assignment* statements with more than one waveform. The next section defines the dynamic semantic for this reduced form of VHDL.

5 The Dynamic Semantics

In this section, the Dynamic Model for the reduced form of VHDL (as defined by the reduction algebra of the Static Model) is presented as a declarative definition of its *state space* over time, independent of any method of simulation. The state space consists of the values of declared signals and ports in the description.[2] Since the net result of evaluating a VHDL description is reflected in the changes in the values of the signals in the description, their 'waveforms' represent a semantics of the description. Thus, the ultimate goal in the definition of the semantics is to define what value a signal would take in a given time interval (for a given VHDL description). In other words, we will define a snapshot of the entire simulation of VHDL in one step. In order to achieve this, we break up the VHDL simulation into time intervals and establish minimum constraints on the time intervals in which the *process* statements (and their sequential statements) can be evaluated. These constraints are then used to define the state space over time.

Since time plays an important role, we first define a set of reference time intervals against which other intervals can be constructed. A sequence of adjacent time intervals referred to as the *time keeper* (or simply TK) provides a convenient reference frame. Formally, TK is defined as

TK = (TK(i) : i $\geq$ 0 $\wedge$ meets(TK(i), TK(i+1))).

The notation TK(i) will be used to denote the i^{th} element of TK. It is important to understand that the time keeper plays a dual role here. It serves as a reference for defining the time periods of evaluating statements (the real time) as well as provides the notion of passage of *virtual*[3] time against which values of signals are defined. Crossing the boundaries of the time keeper intervals corresponds to moving ahead in virtual time. Therefore, we will

[1] Following the convention in the LRM, the term 'declared signals' refers only to data objects that are declared as signals in the VHDL description and the term 'signals' refers to declared signals as well as ports

[2] Files also form a part of the state space but they are not covered in this work

[3] The term *virtual* time was coined by Jefferson [9] to refer to simulation time in his description of the Time Warp paradigm for synchronizing parallel discrete event simulations.

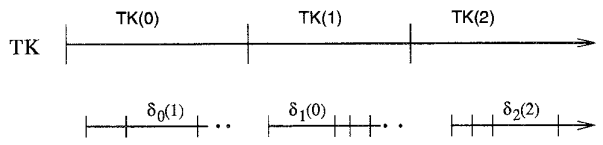

Figure 3. Delta Intervals

assume that each TK(i) will be associated with 1 femto-second, which is the smallest time unit defined in VHDL.

To model asynchronous transient activity in hardware circuits, *process* statements in VHDL are evaluated repeatedly in cycles (called *delta cycles* in the LRM) without advancing virtual time until stable values are reported. In the context of this work, the time intervals representing these cycles must be contained within a single TK(i). These time periods will be referred to as *delta intervals*. In general there is no upper bound on the number delta cycles that can occur before real time advances. Consequently, a sequence of an infinite number of delta intervals (corresponding to VHDL delta cycles) is associated with every TK(i) (Figure 3). This is formally stated as

$$\forall\ TK(i) : i \geq 0 \wedge TK(i) \in TK,$$
$$\exists\ \delta_i, \delta_i = (\ \delta_i(j) : j \geq 0 \wedge during(\delta_i(j), TK(i)) \wedge meets(\delta_i(j), \delta_i(j+1))\ )$$

where the j^{th} time interval in the sequence δ_i is referred to as $\delta_i(j)$. There is no virtual time difference between any two intervals in the sequence δ_i. Although delta intervals bear a close relation to the LRM delta cycles, these are not interchangeable concepts. Delta cycles merely occur within delta intervals and there need not be infinite delta cycles for every TK(i). It is important to note that we are not characterizing the simulation cycle of the LRM. *The delta interval must be viewed merely as time intervals in which the transient behavior of digital devices can be captured.* We now have a set of reference time intervals based on which the constraints on the evaluation intervals of VHDL statements can be established.

5.1 Evaluation of VHDL Statements

The constraints on the evaluation intervals of statements can be expressed in terms of the predicate EVAL(stmt, t) and the set EV(stmt, t) where stmt is a VHDL statement and t is a time interval. The predicate EVAL(stmt, t) will hold if the constraints for correct evaluation of stmt in t are met. The set EV(stmt, t) is a set of 2-tuples $\langle\ stmt_i, t_i\ \rangle$ where $stmt_i$ is a statement evaluated as the result of the evaluation of stmt and t_i is the time interval in which $stmt_i$ is evaluated. For example, in Figure 4 the statement s1 in *process* statement p1 is evaluated in an interval (say $\hat{t}$) contained within $\delta_0(0)$. Then EVAL(s1, $\hat{t}$) is `true` and EV(s1, $\hat{t}$) = $\{\langle\ s1, \hat{t}\ \rangle\}$. The set EV would be non-trivial for statements like the *case* statement whose evaluation results in the evaluation of a sequence of statements. We now define EVAL and EV for the *process* statement and some sequential statements.

5.1.1 The Process Statement

Elaboration of a VHDL description results in a collection of *process* statements that must be evaluated in parallel. A *process* statement pr, pr $\in$ Process-Statements, is represented as

process-stmt = $\langle$ name, postponement, ordered_statements, set-of-drivers $\rangle$
postponement = [`postponed` | `not-postponed`].

The sensitivity-list of the *process* statement is the list of signals and ports which the *process* statement is sensitive to. The postponement field indicates whether the *process* statement is a postponed one or not. Every *process* statement also has associated with it, a list of ordered_statements which have to be executed in order. The transaction-set is a set of 2-tuples associating signals to their corresponding transaction lists. Transaction lists will be dealt with in detail in Section 5.2.

The semantics of a *process* statement is defined a little differently from sequential statements because its evaluation occurs over an infinite interval.[4] Evaluation of *process* statements results in a repeated evaluation of the confined sequential statements. For example, in Figure 4, the evaluation of process p1 leads to the repeated evaluation of s1 and s2. Let us assume that s1 is evaluated in the intervals t1, t3, t5 and so on and that s2 is evaluated in the intervals t2, t4, t6 and so on (*the duration of these intervals depends on the constraints they must satisfy*). Then, it clear that the evaluation set of p1 (we refer to it as EV_{p1}) is equal to $\{\langle$ s1, t1 $\rangle, \langle$ s2, t2 $\rangle, \langle$ s1, t3 $\rangle, \ldots\}$. Furthermore, the following conditions must hold: (i) The predicates EVAL(s1, t1), EVAL(s2, t2), EVAL(s1, t3) and so on must be true and (ii) The predicates meets(t1, t2), meets(t2, t3), meets(t3, t4) and so on must be true. Note that, in general, the statement s1 could be an *if* statement or a *case* statement and therefore $\langle$ s1, t1 $\rangle$ should be replaced by EV(s1, t1) in EV_{p1}. Bearing these conditions in mind, the evaluation set EV_{pr} for a *process* statement pr can be defined as

$$\forall\ pr: pr \in \text{Process-Statements},$$
$$\exists\ EV_{pr}: EV_{pr} = \bigcup EV(stmt_i, t_i) \wedge$$
$$(during(t_0, \delta_0(0)) \vee overlaps(\delta_0(0), t_0)) \wedge meets(t_i, t_{i+1}) \wedge$$
$$stmt_0 = pr.ordered\_statements_0 \wedge$$
$$stmt_{i+1} = stmt_{(i+1)\ mod\ (\#pr.ordered\_statements)} \wedge$$
$$stmt_i \in \text{Sequential-Statements} \wedge EVAL(stmt_i, t_i)$$

where the symbol # denotes the size of a sequence and $(i+1)\ mod\ (\#pr.ordered\_statements)$ is equal to $i+1$ if $i < \#pr.ordered\_statements$ and 1 otherwise. The above definition also gives the order in which the statements are evaluated in a VHDL simulation. The initial statement of any process begins in the very first delta interval $\delta_0(0)$ (therefore all processes are fired in the first delta interval of TK(0) though they need not be fired simultaneously). Furthermore, the relation $during(t_0, \delta_0(0)) \vee overlaps(\delta_0(0), t_0)$ holds because the initial statement evaluation could be contained within $\delta_0(0)$ (as in the case of a signal assignment statement [refer to Section 5.1.2]) or extend beyond $\delta_0(0)$ (as in the case of a *wait* statement [refer to Section 5.1.3]). The following sections present the definitions of EVAL and EV for some sequential statements.

[4] It is assumed that simulation does not stop at any time and we ignore the LRM definition of TIME'HIGH.

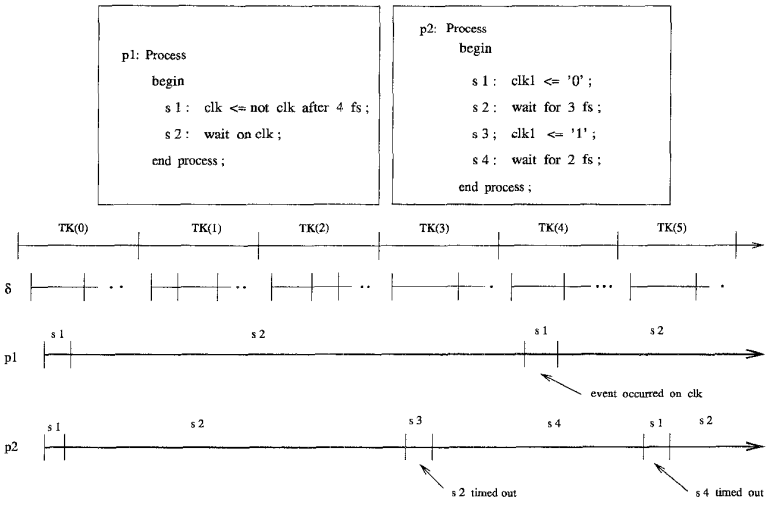

Figure 4. Parallel Evaluation of Process Statements

5.1.2 The Signal Assignment Statement

The *signal assignment* statement is used to update values of signals which represent wires in a digital circuit. A signal assignment statement sa, sa $\in$ Signal-Assignment, is formally represented as

$$sa = \langle \text{ name, destination, pulse-rejection, expr, delay } \rangle ,$$

where name is the unique name of the assignment, destination specifies the target signal, pulse-rejection specifies the pulse rejection limit of the assignment, expr defines the expression on the RHS of the signal assignment, and delay specifies the delay associated with the assignment. The delay is 0 femto-seconds if no delay is specified.

A signal assignment statement is constrained to be evaluated in a subinterval of some δ_i(j) following the proper order of the statements in a *process* statement. This constraint is formally stated in term of EVAL as

$$\text{EVAL(sa, t)} \equiv \exists \text{ i, j: during(t, } \delta_i(j)).$$

Informally, this formula merely indicates that evaluation of any signal assignment statement is wholly contained within a delta interval. However, the semantic definition is not yet complete. Execution of signal assignment statements affects the state space. These effects are captured in Sections 5.2 and 5.3. What remains is the definition of the evaluations set EV which is simply EV(sa, t) = $\{ \langle \text{ sa, t} \rangle \}$.

The definitions of EVAL and EV for *variable assignment, assert* and *report* statements are identical and are therefore not presented here. Moreover, any condition is also evaluated within a delta interval and therefore the definition of EVAL and EV for a condition evaluation is also the same. For lack of space, we have omitted definitions for the *if, case* and *loop* statements (refer to [15]).

5.1.3 The WAIT Statement

A wait statement causes the suspension of a *process* statement. It models circuit delays and sensitivity of components to signals. A wait statement wa, wa $\in$ Wait-Statements, is represented as

$$wa = \langle \text{ name, sensitivity-list, condition, timeout } \rangle ,$$

where name is the unique name of the port, sensitivity-list is a set of signals to which the wait statement is sensitive, condition is a boolean expression, and timeout is the timeout clause of the wait statement.

In order to terminate the evaluation of a wait statement, an event must occur on a signal or port in its sensitivity list and the condition must evaluate to `true`. If this does not happen within the time limit specified in the timeout clause, the evaluation terminates when the time limit has elapsed. For example in Figure 4, the evaluation of a wait statement s2 in *process* statement p2 terminates in $\delta_3(0)$ because the timeout condition is satisfied in that delta interval. This is modeled in terms of the predicate SATISFIES(wa, i, j, k, l) which is `true` for a wait statement wa whose evaluation begins in δ_i(j) and ends in δ_k(l). This is written formally as

$$\begin{aligned}&\text{SATISFIES(wa, i, j, k, l)} \equiv \\ &\quad i \leq k \wedge ((\exists\, s : s \in \text{Signals} \cup \text{Ports}, \\ &\quad s \in \text{wa.sensitivity-list} \wedge \text{EVENT}(s, \delta_k(l)) \wedge \\ &\quad \text{VAL(wa.condition, } \delta_k(l)) = \text{true}) \vee \\ &\quad (\text{wa.timeout} = k\text{-}i \wedge l = 0 \wedge k \neq i) \vee \\ &\quad (\text{wa.timeout} = 0 \wedge l = j\text{+}1 \wedge k = i)),\end{aligned}$$

where EVENT is a predicate (formally defined in Section 5.3.4) which is `true` whenever a VHDL event occurs on a signal. If the wait statement exists without any of the three clauses, its default condition is `true` and default timeout is TIME'HIGH - TIME'NOW (LRM § 8.1, ¶ 112, Ł 65). However, for our purposes, we simply assume that the timeout is infinity in this case.

Now the constraints on the time period of evaluation of a wait statement can be defined. For a non-postponed wait statement wa_n, EVAL(wa_n, t) is `true` if wa_n is evaluated in an interval t. The constraints on t are such that it begins in a particular delta interval δ_i(j) and ends in another delta interval δ_k(l) (*i.e.,* overlaps(δ_i(j), t) $\wedge$ overlaps(t, δ_k(l)) holds) such that the predicate SATISFIES(wa, i, j, k, l) is `true` and no intermediate delta interval δ_m(n) exists such that SATISFIES(wa, i, j, m, n) is `true`. Stated formally, a wait statement, wa_n in a non-postponed *process* statement evaluates in a time interval t, such that

$$\begin{aligned}\text{EVAL}(wa_n, t) \equiv\ & \exists\ i, j, k, l : \text{overlaps}(\delta_i(j), t) \wedge \\ & \text{SATISFIES}(wa, i, j, k, l) \wedge \text{overlaps}(t, \delta_k(l)) \wedge \\ \neg\ & \exists\ m, n: (m < k) \vee (m = k \wedge n < l), \\ & \text{SATISFIES}(wa, i, j, m, n)\end{aligned}$$

holds. The evaluation set EV(wa, t) for the wait statement is simply $\{ \langle wa, t \rangle \}$. The semantics for wait statements in postponed *process* statements has a similar definition (refer [15]).

5.2 Transaction Lists

Execution of signal and variable assignment statements affect values of signals and variables respectively. Variables are updated immediately whereas signals are updated after a specified delay. Therefore, in the case of signals, a mechanism is needed to store the value and delay information temporarily. The LRM defines *transaction lists* or *drivers* that store the necessary information for updating signals correctly. A specific algorithm (§ 8.5, ¶ 118) for maintaining these transaction lists is also provided. Previous semantic definitions for VHDL characterize this algorithm thus disallowing any attempts at investigating alternative techniques to achieve the same effect. This section provides a definition of transaction lists not by specifying an algorithm that adds and deletes its contents from time to time but by *declaratively specifying what its contents will be*, given a VHDL description. This definition bears a close resemblance to the LRM algorithm since the net effect of the algorithm on the transaction lists must be reflected in the definition. However, the definition does not characterize the algorithm itself. The definition of transaction lists is not necessary and we could directly define the *state space* but the axioms would be greatly simplified if we first defined transaction lists.

Associated with every *process* statement is a set of drivers for the signals to which assignments are made in the process. In Section 5.1.1 the representation of the *process* statement was provided. The representation for the set of drivers is

$$\begin{aligned}&\text{set-of-drivers} = \{\text{dr-set} : \text{dr-set} = \langle \text{destination, driver} \rangle\}, \\ &\text{driver} = \{\text{transaction} : \text{transaction} = \langle \text{v}, t_r, t_d \rangle\}\end{aligned}$$

where t_r denotes the reject pulse limit time interval, t_d denotes the time interval beginning with the signal assignment statement that posts the transaction and ends at the time that the transaction is slated to become a driver, and v is the value posted by the signal assignment. Figure 5 shows the relationships of t_d and t_r to one another and to the time interval of evaluation of the corresponding signal assignment statement s3. For a null assignment (applicable only for guarded signals) v takes on the value $null$.

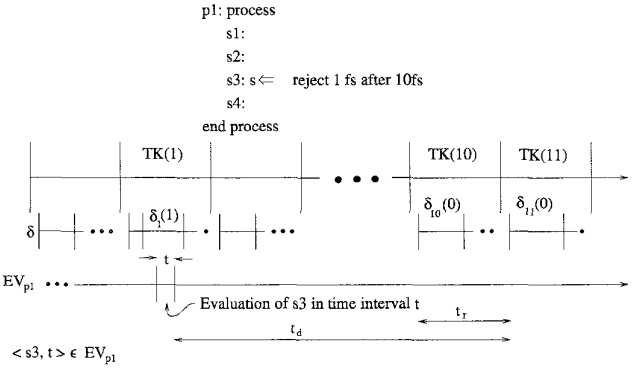

Figure 5. A Transaction

Since a distinct driver is associated with each unique signal destination of the signal assignments in each *process* statement, two processes may have drivers for the same signal. The formal definition of these drivers proceeds in two steps. First, a set containing all the transactions posted by the signal assignment is defined. Second, a set denoting the 'true' transactions (transactions not removed due to marking (§8.4.1, ¶117-119) to be applied for defining the signal's driving values) is defined. The set containing all transactions posted by a signal assignment is called ALL-TR. Formally, ALL-TR for every signal s and every process pr is defined as:

$$\begin{aligned}&\forall\ \text{pr: pr} \in \text{Process-Statements}, \\ &\forall\ \text{s: (s} \in \text{Signals} \cup \text{Ports} \wedge \exists\ \text{sa: sa} \in \text{Signal-Assignment} \wedge \\ &\quad \text{sa} \in \psi\ \text{(pr.ordered\_statements)} \wedge \text{sa.destination} = \text{s}), \\ &\quad \text{ALL-TR(s, pr)} = \\ &\quad\quad \{ \langle \text{v}, t_r, t_d \rangle : \exists\ \text{i, j, t : during(t, } \delta_i\text{(j))} \wedge \langle \text{sa, t} \rangle \in \text{EV}_{pr} \wedge \\ &\quad\quad\quad \text{meets(t, } t_d\text{)} \wedge \text{VAL(sa.expr, t)} = \text{v} \wedge \\ &\quad\quad\quad if\ \text{(sa.delay = 0)}\ then \\ &\quad\quad\quad\quad \text{meets } (t_d, \delta_i\text{(j+1)}) \wedge t_r = \emptyset \\ &\quad\quad\quad elseif\ \text{(sa.pulse-rejection = 0)}\ then \\ &\quad\quad\quad\quad \text{meets } (t_d, \delta_{i+sa.delay}\text{(0)}) \wedge t_r = \emptyset \\ &\quad\quad\quad else \\ &\quad\quad\quad\quad \text{begins}(\delta_{i+sa.delay-sa.pulse-rejection}\text{(0)}, t_r)\ \wedge \\ &\quad\quad\quad\quad \text{meets } (t_r, \delta_{i+sa.delay}\text{(0)}) \wedge \\ &\quad\quad\quad\quad \text{meets } (t_d, \delta_{i+sa.delay}\text{(0)}) \}.\end{aligned}$$

The above axiom merely states the relationships captured in Figure 5. If the specified delay is 0 femto-second, then t_d extends from the end of the statement evaluation interval t to the beginning of the very next delta interval. Since the pulse rejection limit is also 0 femto-second, $t_r = \emptyset$. If the specified delay is x femto-second (x > 0) and pulse rejection is 0 femto-second, then t_d extends from the end of t to the beginning of the delta cycle occuring after x femto-second (in real time) and $t_r = \emptyset$ as before. If the specified delay and the pulse rejection limit are positive non-zero values then the relations on t_d and t_r are as in the example depicted in Figure 5. Note that although ALL-TR(s, pr) has been specified for all signals and processes, it will be empty if no assignment is made to s in pr.

The LRM defines a marking scheme to be used for updating projected waveforms that can be informally stated as follows. If $\langle$ v, t_d, t_r $\rangle$ is a transaction posted by a signal assignment statement and $\langle$ v1, $t1_d$, $t1_r$ $\rangle$ is the transaction posted by a later assignment (to the same signal) such that the former transaction is projected at or after the time projected by the latter, then the relationship between t_d and $t1_d$ is given by during($t1_d$, t_d) $\vee$ finishes($t1_d$, t_d). In such a case, the former transaction is a false transaction and must not appear in the final transaction list. If the latter assignment is such that it projects a value *after* the former assignment but the former projection falls within the pulse rejection limit of the latter, then the relation between t_d, $t1_d$, and $t1_r$ is given by overlaps(t_d, $t1_d$) $\wedge$ overlaps(t_d, $t1_r$). The condition that must hold for the the former transaction to be a false one in this case is either that (1) the value projected by the latter is different from the value projected by the former or that (2) The value projected by the former is same as that projected by the latter and there exists a transaction that projects a different value at a time-stamp in between the earlier two time-stamps.

Any transaction in ALL-TR(pr, s) is a false transaction if it meets any of the conditions stated above and hence must not be included in the driver for s in pr. Furthermore, the initial values of the signals must also be included in the driver. Thus the formal definition of the drivers of signals for a *process* statement pr is given as

$$
\begin{aligned}
&\forall \text{ pr: pr} \in \text{Process-Statements,}\\
&\text{pr.set-of-drivers} =\\
&\quad \{ \langle \text{ s, driver } \rangle : \text{s} \in \text{Signals} \cup \text{Ports} \wedge\\
&\quad (\exists \text{ sa: sa} \in \text{Signal-Assignment} \wedge\\
&\quad \text{sa} \in \text{pr.ordered\_statements} \wedge \text{sa.destination} = \text{s}) \wedge\\
&\quad \text{driver} = \{ \langle \text{ v}, t_r, t_d \rangle : (\text{v} = \text{s.initial-value} \wedge t_r = \psi \wedge\\
&\qquad \text{meets}(t_d, \delta_0(0))) \vee\\
&\qquad (\langle \text{ v}, t_r, t_d \rangle \in \text{ALL-TR(s, pr)} \wedge\\
&\qquad \neg \exists \langle \text{ v1}, t1_r, t1_d \rangle \in \text{ALL-TR(s, pr)} - \{ \langle \text{ v}, t_r, t_d \rangle \},\\
&\qquad\quad \text{during}(t1_d, t_d) \vee \text{finishes}(t1_d, t_d) \vee\\
&\qquad\quad (\text{overlaps}(t_d, t1_r) \wedge\\
&\qquad\qquad (\text{v1} \neq \text{v} \vee (\text{v1} = \text{v} \wedge \exists \langle \text{ v2}, t2_r, t2_d \rangle \in \text{ALL-TR(s, pr)},\\
&\qquad\qquad\quad \text{v2} \neq \text{v1} \wedge \text{overlaps}(t_d, t2_d) \wedge\\
&\qquad\qquad\quad \text{overlaps}(t2_d, t1_d))))) \} \}.
\end{aligned}
$$

This says that a transaction $\langle$ v, t_d, t_r $\rangle \in$ ALL-TR is in the driver corresponding to s if there does not exist a transaction $\langle$ v1, $t1_d$, $t1_r$ $\rangle \in$ ALL-TR such that its presence would cause the deletion of the former transaction. The driver contains all transactions from ALL-TR except those that were false transactions. Additionally, the driver contains the tuple $\langle$ s.initial-value, t_d, $\emptyset$ $\rangle$ where meets(t_d, $\delta_0(0)$) holds. It is possible to conceive of algorithms that achieve the same final driver without following the LRM algorithm. They can be validated if it can be proved that they always lead to the correct driver.

5.3 The State Space

VHDL allows a hierarchical description of hardware and therefore signals and ports within a block may be associated as actuals with a formal port of a block at a lower level in the hierarchy. The Static Model represents these associations as the set Port-Associations. Formally, a port-association pa, pa $\in$ Port-Associations, is represented as $\langle$ formal, fn-f2a, actual, fn-a2f $\rangle$ where formal $\in$ Ports, fn-f2a is a type conversion function that maps the values of the formal's type to the values of the actual's type, actual $\in$ Signals $\cup$ Ports, and fn-a2f is a type conversion function mapping the values of the actual's type to values of the formal's type. In the case where no type conversion function(s) is specified in the original VHDL description, fn-a2f and/or fn-f2a will be a function that simply returns the input value.

If the value of a signal s at a lower level in the hierarchy changes, then these changes must also be reflected in the ports or signals that are associated (or connected) with s at a higher level. Similarly, changes at the top of the hierarchy must flow down to the lower levels. Upward flow of information is achieved by calculating the *driving value* of all signals (therefore driving values are not defined for ports of mode `in`) and downward flow is achieved by calculating *effective values* for all signals (therefore effective values are not defined for ports of mode `out`).

5.3.1 Driving Values

The driving value of a signal is defined in the LRM (§ 4.3.1.2, ¶ 55, Ł 183) as "the value that the signal provides as a source of other signals." A signal may have a number of sources associated with it at a particular time. The LRM defines a source to be "either a driver or an `out`, `inout`, `buffer` or `linkage` port of a component instance or of a block statement with which the signal is associated." Drivers were defined in the previous section.

For a signal s, if there is no *signal assignment* statement whose destination is s, and if s is not associated with any other port, it has no source. This may happen if a signal is declared in an architecture but no assignment is made to it or if a port is *not* of mode `in` and is at the lowest level in a hierarchy and no assignment is made to it. This is formally stated in terms of a predicate nullsources(s) as

$$
\begin{aligned}
&\text{nullsources(s)} \equiv \neg \exists \text{ sa: sa} \in \text{Signal-Assignment,}\\
&\qquad\qquad \text{sa.destination} = \text{s} \wedge\\
&\qquad\qquad \neg \exists \text{ pa: pa} \in \text{Port-Association, pa.actual} = \text{s}
\end{aligned}
$$

which states that nullsources(s) is `true` if s has no source and `false` otherwise. For a signal which has no source, its driving value is the initial value for all time. Formally,

$$
\begin{aligned}
&\forall \text{ s, t: s} \in (\text{s} \in \text{Signals} \vee (\text{s} \in \text{Ports} \wedge \text{s.mode} \neq \texttt{in})) \wedge \text{nullsources(s)},\\
&\quad \text{DRIVING-VAL(s, t)} = \text{s.initial-value}
\end{aligned}
$$

where DRIVING-VAL is a function similar to VAL and defines the driving value for a signal during a time interval[5]. For the remaining cases, the set of all drivers and ports that act as sources for a signal needs to be constructed before the driving value can be determined. Considering only drivers first, we define the value of a driver D in a time interval t. Referring to Figure 5, if $\langle$ 5, t_d, t_r $\rangle \in$ D, then for any time interval *t* if meets(t_d, t) $\vee$ before(t_d, t) holds, the value of D will be 5 in t if no other value is scheduled for *s* in between the upper bound of t_d and the lower bound of t by a different transaction. Thus, the function D-Val which returns the value of a driver is defined as

[5]The time intervals should be chosen such that the value of the signal is constant during the interval

$\forall$ D: $\langle$ s, D $\rangle \in \bigcup_{pr \in Process-Statements}$ pr.transaction-set,
D-Val(D, t) = v $\wedge$ $\langle$ v, t_d, t_r $\rangle \in$ D $\wedge$ (meets(t_d, t) $\vee$ before(t_d, t)) $\wedge$
$\neg\exists$ $\langle$ v', t'_d, t'_r $\rangle \in$ D $-$ { $\langle$ v, t_d, t_r $\rangle$ }, endsafter(t'_d, t_d) $\wedge$
endsafter(t, t'_d).

It should be noted that D-Val(D, t) (and hence all other functions defined below) is valid only for time intervals where the value of the driver is constant. We now define the set All-Drivers which is a collection of the values of all the drivers associated with a signal in a given time interval t. If the value is *null*, it is not included in the set and the corresponding driver is said to be "disconnected". Formally, this is stated as

$\forall$ s: ((s $\in$ Signals) $\vee$ (s $\in$ Ports $\wedge$ s.mode $\neq$ `in`)) $\wedge$ $\neg$ nullsources(s),
All-Drivers(s, t) =
{ v : $\langle$ s, D $\rangle \in \bigcup_{pr \in Process-Statements}$ pr.transaction-set $\wedge$
v = D-Val(D, t) $\wedge$ v $\neq$ `null` }.

The set All-Ports consists of values derived from any ports that may be associated with the given signal or port. This is formulated as

$\forall$ s: ((s $\in$ Signals) $\vee$ (s $\in$ Ports $\wedge$ s.mode $\neq$ `in`)) $\wedge$
$\neg$ nullsources(s), All-Ports(s, t) =
{ v : pa $\in$ Port-Associations $\wedge$ pa.actual = s $\wedge$
v = pa.fn-f2a(DRIVING-VAL(pa.formal, t)}.

The set of all sources can now be defined as the union of All-Drivers and All-Ports.The set All-Sources(s, t) could be empty if s has sources but they are all disconnected during the time interval t. In the case where s is not a resolved signal and is driven either by a single driver or a single port (but not both), the driving value is given by

$\forall$ s: ((s $\in$ Signals) $\vee$ (s $\in$ Ports $\wedge$ s.mode $\neq$ `in`)) $\wedge$ s.res-fn = ϕ $\wedge$
$\neg$ nullsources(s), DRIVING-VAL(s, t) = v $\wedge$ v $\in$ All-Sources(s, t).

It would be an error if the signal or port is unresolved and the set All-Sources has more than one element in it. In the case of resolved signals with at least one source, if All-Sources is empty and the signal kind is `register` then the value of the signal is unchanged from the previous time interval. Otherwise the driving value is determined by applying the resolution function to All-Sources. This is formally stated as

$\forall$ s: ((s $\in$ Signals) $\vee$ (s $\in$ Ports $\wedge$ s.mode $\neq$ `in`)) $\wedge$
s.res-fn $\neq$ ϕ $\wedge$ $\neg$ nullsources(s),
DRIVING-VAL(s, t) =
if (All-Sources = ϕ $\wedge$ s.signal-kind = `register` $\wedge$ $\exists$ $\hat{t}$, meets($\hat{t}$, t))
then DRIVING-VAL(s, $\hat{t}$)
else s.res-fn(All-Sources(s, t)).

5.3.2 Effective Values

The LRM (§ 4.3.1.2, ¶ 55, Ł 183) defines the effective value to be the value "obtainable by evaluating a reference to the signal within an expression." The effective value of declared signals, ports of mode `buffer` and unconnected ports of mode `inout` is equal to their driving values. Formally, this is expressed as

$\forall$ s: s $\in$ Signals $\vee$
(s $\in$ Ports $\wedge$ (s.mode = `buffer` $\vee$ (s.mode = `inout` $\wedge$
s.connection = unconnected))),
EFFECTIVE-VAL(s, t) = DRIVING-VAL(s, t).

For a connected port of mode `in` or `inout`, the effective value is defined to be " the effective value of the actual part of the association element that associates an actual with the signal." Note that the effective value flows downwards in the hierarchy of ports (*i.e.,* it flows from the actual to the formal). Formally, the transmission of effective values down the hierarchy (complete with type conversion) is expressed as

$\forall$ pa: pa $\in$ Port-Associations $\wedge$ (pa.formal.mode = `in` $\vee$
pa.formal.mode = `inout`),
EFFECTIVE-VAL(pa.formal, t) =
pa.fn-a2f (EFFECTIVE-VAL(pa.actual, t)).

In case the port is an unconnected port of mode `in`, the effective value is given by the default value associated with the port. Formally,

$\forall$ s, t: s $\in$ Ports $\wedge$ s.connection = unconnected $\wedge$ s.mode = `in`,
EFFECTIVE-VAL(s, t) = VAL(s.default-expr, t).

The definition of the effective values completes the definition of the state space. The function EFFECTIVE-VAL gives the value of any signal in a given time interval if the value is stable for that interval.

5.3.3 Waveforms of Signals

Having defined the effective values of signals for a given time interval t, their waveform over the entire simulation can be defined. The waveform for a signal is represented as a set of 2-tuples $\langle$ v, t $\rangle$ where v is the value of the signal in time interval t. Note that t must be chosen such that it is the largest interval in which the value of s is v. This set is formally defined as

Waveform(s) = { $\langle$ v, t $\rangle$: EFFECTIVE-VAL(s, t) = v $\wedge$
$\neg\exists$ $\hat{t}$: in(t, $\hat{t}$), EFFECTIVE-VAL(s, $\hat{t}$) = v }.

Informally, the above axioms states that a tuple $\langle$ v, t $\rangle$ belongs to the set if EFFECTIVE-VAL(s, t) = v and there is no larger interval $\hat{t}$ such that EFFECTIVE-VAL(s, $\hat{t}$) = v.

5.3.4 Events

A VHDL event is said to occur on a signal in a time interval δ_i(j) if its value in that interval is different from that of the previous interval. This can be formally stated as

$\forall$ s, i, j : s $\in$ Signals $\cup$ Ports $\wedge$ i $\geq$ 0 $\wedge$ j $\geq$ 0,
EVENT(s, δ_i(j)) $\equiv$ $\exists$ $\hat{t}$: meets($\hat{t}$, δ_i(j)) $\wedge$
DRIVING-VAL(s, δ_i(j)) $\neq$ DRIVING-VAL(s, $\hat{t}$)

5.4 Summary

The semantic model presented in this section describes VHDL behavior in terms of the definition of the state space evaluated by a VHDL description. A unique definition of transaction lists (or drivers) is provided. In particular, a set containing all the posted transactions (*for the entire simulation*) is defined and then conditions that result in the deletion of some of these transactions are defined. The transaction lists can be viewed merely as intermediate sets of time intervals that simplified the definitions of driving

and effective values of signals. Thus, our definition does not force VHDL simulators to maintain transaction lists as long as they satisfy the axioms of Section 5.3. For lack of space, the semantics for sequential statements such as *case* statements and *if* statements are not provided though a complete definition of the Dynamic model is available in the working document [15]. Semantics for variable assignments and postponed processes are also available.

6 Proof of Process-Folding

Communication overhead is a bottleneck in parallel simulation. In a parallel VHDL simulator, each *process* statement is executed as a parallel thread where each of these threads need to communicate with each other. Process-folding (proposed in [10]) is an optimization technique that combines two *process* statements into a single *process* statement, thereby reducing the number of threads in the simulation which in turn speeds up the simulation. Results have indicated an improvement by a factor of 2.2 after folding. In the following material, it will be proved that folding preserves the semantics of VHDL descriptions. The proof will be shown for only one type of folding as defined below.

Combination Method: Consider two process statements $PS_i = \langle N_i, P_i, S_i, D_i \rangle$ and $PS_j = \langle N_j, P_j, S_j, D_j \rangle$ where D_i and D_j are variable declarations and the other fields are as described in Section 7. These two process statements can be combined into a single process statement $PS_{ij} = \langle N_{ij}, P_{ij}, S_{ij}, D_{ij} \rangle$ if the following conditions hold:

1. $S_i = S1_i \frown W_i \frown S2_i$ such that $\forall s \in S1_i \cup S2_i$, $s \notin$ Wait-Statements and $W_i \in$ Wait-Statements.
2. $S_j = S1_j \frown W_j \frown S2_j$ such that $\forall s \in S1_j \cup S2_j$, $s \notin$ Wait-Statements and $W_j \in$ Wait-Statements.
3. $S_{ij} = S1_{ij} \frown W_{ij} \frown S2_{ij}$ where $S1_{ij} = S1_i \frown S1_j$ and $S2_{ij} = S2_i \frown S2_j$.
4. $W_{ij} = W_i = W_j = \langle \emptyset,$ `true`, TO $\rangle$, where TO is the timeout expression.
5. $D_{ij} = D_i \cup D_j$.
6. All signals, before or after the transformation are unresolved, *i.e.,* they have at most one source.
7. The initial value of the state space is the same.

6.1 Proof

In this paper we will merely present the outline of the proof. Formal Gentzen style proofs and their automated versions are available in the working document [15]. The essential premise of the proof is that two VHDL descriptions are logically equivalent if they evaluate the same state space. Therefore, to validate folding, it is sufficient to show that a driver for any arbitrary signal s before folding is 'similar' to the corresponding driver for s after folding. By 'similar' we mean that though the drivers may not be identical, they lead to the same waveform for s. Note that if $\langle v, t_d, t_r \rangle$ is a transaction in ALL-TR for s, then the point where t_d ends determines the point at which s is scheduled to acquire the value v. The point

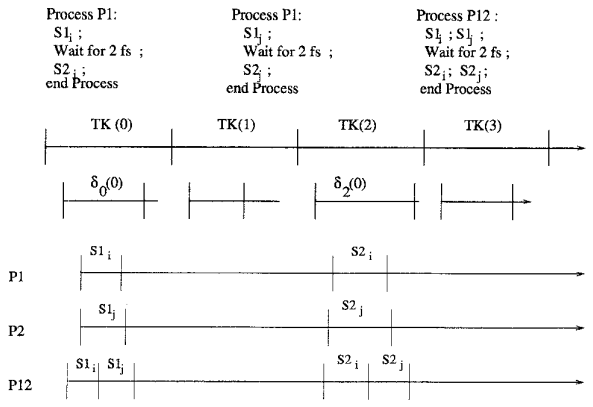

Figure 6. Execution of Folded Process

where t_d begins plays no role. Hence two drivers could be similar if their respective t_ds simply ended at the same point with no restriction on where they started.

Let there be an assignment to a signal s in *process* statement P1. Let the set of all transactions for s in P1 be ALL-TR(s, P1) and that in P12 be ALL-TR(s, P12) We will first argue that ALL-TR(s, P1) and ALL-TR(s, P12) are similar and then show that if a transaction is deleted from ALL-TR(s, P1) (due to marking), it will be deleted from ALL-TR(s, P12) also, thus proving that the resulting transaction sets are also similar.

Lemma 1 *For any signal s,* ALL-TR(s, P1) *(or* ALL-TR(s, P2)*) is similar to* ALL-TR(s, P12).

Proof: If the transactions by a signal assignment to s before and after folding are posted in the same delta interval δ_i(j), then the transactions are similar. This follows from the definition of t_d in Section 5.2 which states that the end point of t_d depends only on the delta interval in which it was posted and the delay specified in the signal assignment. The exact point at which the transaction is posted is not important. From Figure 6, which shows the parallel evaluation of the *process* statements, we see that if a signal assignment is evaluated in some δ_i(j) before folding, it will also be evaluated in the same δ_i(j) after folding and hence, all transactions posted by any assignment after folding, will be similar to those posted before folding.

Lemma 2 *If a transaction* $\langle$ v1, $t1_d$, $t1_r$ $\rangle$ *is deleted from the set* ALL-TR(s, P1) *for any signal* s, *then the corresponding transaction will also be deleted from the set* ALL-TR(s, P12).

Proof: There are three cases to be considered for the proof. From Section 5.2 we know that a transaction $\langle$ v1, $t1_d$, $t1_r$ $\rangle$ will be deleted if another transaction $\langle$ v1', $t1_d$', $t1_r$' $\rangle$ exists such that either of the following conditions hold: (1) during($t1_d$', $t1_d$), (2) finishes($t1_d$', $t1_d$) and (3) overlaps($t1_d$', $t1_r$). We can show that in general, for any two intervals t1 and t2, the relations during, finishes, and overlaps are preserved after folding. A formal proof of this available in the working document. Since the relations between time intervals

are preserved, if any transaction is deleted from ALL-TR(s, P1) (or ALL-TR(s, P2)) before folding, the corresponding transaction from ALL-TR(s, P12) will also be deleted.

Thus we have shown that for any signal, its drivers after folding will be similar to its drivers before folding and hence the state space is preserved. This completes the proof of the validity of the special case of process-folding described earlier.

7 Conclusions

In this paper, the semantics of VHDL was formalized in a declarative style using interval temporal logic. The definition captured the semantics of a vast subset of VHDL. Additionally, a validation of process-folding for CAD tool optimization was presented. This research aims at CAD tool optimization at two levels: (1) The static model defines a reduced form for VHDL thus simplifying the construction of a simulator for VHDL. For example, a VHDL simulator need not recognize any concurrent statements (except the *process* statement) or any transport delayed signal assignment. (2) The declarative style used in developing the dynamic model supports the investigation and validation of optimizing techniques such as rewriting of VHDL code and alternative simulation algorithms. For example, parallel simulators using distributed synchronization protocols such as time warp [9] can be constructed against this definition.

Though the dynamic model is self contained, procedures and functions still need to be characterized. Future work will also involve validation of a larger collection of optimizing techniques. Attempts are on currently to use the PVS theorem prover to automate proofs in interval logic.

References

[1] ALLEN, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM 26* (Nov. 1983), 832–843.

[2] BARTON, D. L. A functional characterization of elements of the VHDL simulation cycle. In *VHDL Users' Group Spring 1991 Conference* (Cincinnati, OH, April 1991), pp. 91–96.

[3] BORGER, E., GLASSER, U., AND MULLER, W. The semantics of behavioral VHDL '93 descriptions. In *Proceedings of EURO-DAC '94/EURO-VHDL '94* (Grenoble, France, September 1994).

[4] BORRIONE, D. D., PIERRE, L. V., AND ß ALEM, A. M. Formal verification of VHDL descriptions in the prevail environment. *IEEE Design & Test of Computers 9*, 2 (June 1992), 42–56.

[5] DAMM, W., JOSKO, B., AND ß CHLOR, R. A net-based semantics for VHDL. In *Proc. of the European Design Automation Conference with EURO-VHDL '93* (CCH Hamburg, Germany, September 1993), pp. 514–519.

[6] DAVIS, K. C. A denotational definition of the VHDL simulation kernel. *Proc. 11th Int. Symp. on Computer Hardware Description Languages* (1993), 509–521.

[7] GORDON, M. Why higher order logic is a good formalism for specifying and verifying hardware. *Formal Aspects of VLSI design* (1986), 153–177.

[8] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1993.

[9] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.

[10] MCBRAYER, T., AND WILSEY, P. A. Process combination to increase event granularity in parallel logic simulation. In *9th International Parallel Processing Symposium* (April 1995).

[11] OLCOZ, ß ., AND COLOM, J. M. Toward a formal semantics of IEEE Std. VHDL 1076. In *Proc. of the European Design Automation Conference with EURO-VHDL '93* (CCH Hamburg, Germany, September 1993), pp. 526–531.

[12] READ, ß . E. G., AND EDWARDS, M. D. A formal semantics of VHDL in Boyer-Moore Logic. In *Proc. of the Conf. on Concurrent Engineering and Electronic Design Automation* (Poole, Great Britain, 1994).

[13] VAN TASSEL, J. P. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England, July 1993.

[14] WILSEY, P. A. Developing a formal semantic definition of VHDL. In *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet, Ed. Kluwer Academic Publishers, The Netherlands, 1992, pp. 243–256.

[15] WILSEY, P. A. Formal models of digital systems compatible with VHDL, 1994. (available on the www at `http://www.ece.uc.edu/~paw/rassp/`).

[16] WILSEY, P. A., BENZ, D. M., AND PANDEY, ß . L. A model of VHDL for the analysis, transformation, and optimization of digital system designs. In *Conference on Hardware Description Languages (CHDL '95)* (August 1995).

[17] WILSEY, P. A., MCBRAYER, T. J., AND ß IMS, D. Towards a formal of VLSI systems compatible with VHDL. In *VLSI 91* (Amsterdam, The Netherlands, August 1991), A. Halaas and P. B. Denyer, Eds., Elsevier Science Publishers B. V. (North-Holland), pp. 225–236.