The Semantics of Timing Constructs in Hardware Description Languages

David C. Luckham
Youm Huh
Alec G. Stanculescu

Technical Report No. CSL-TR-86-303

Program Analysis and Verification Group Report No. 32

August 1986

The work was supported by the Advanced Research Projects Agency, Department of Defense, under Contract N00039-84-C-0211.

The Semantics of Timing Constructs in Hardware Description Languages

CSL-TR-86-303

David Luckham
Youm Huh
Alec G. Stanculescu

Program Analysis and Verification Group
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

Three different, approaches to the representation of time in high level hardware design Languages are described and compared. The first is the timed assignment statement of ADLIB/SABLE which anticipates future events. The second is the timed assignment, of VHDL which predicts future events and allows predictions to be preempted by other predictions. The third is a new proposed method of expressing time dependency by qualifying expressions so that. their values are required to be constant over a specified time interval. Examples comparing these three approaches are given. It is shown how time-qualified expressions could be introduced into a hardware description language. The possiblility of proving correctness of hardware models in this language is illustrated.

Keywords: Hardware description languages, timing construct, timed assignment, anticipatory semantics, preemptive semantics, timing qualifier, qualified expression, guarded assignment.

Computer Systems Laboratory
Stanford University

Table of Contents

List of Figures

1. introduction

In conventional hardware description languages such as ADLIB [1], HHDL [3], VHDL [5] [6], DABL [2], and TEGAS [4], timing behavior is expressed using assignment statements with a timing delay. The delay can be associated with the assignment statement such as in ADLIB and HHDL, or it can be specified in an input-output delay table such as in DABL. An assignment statement with timing **delay is** called **a timed** *assignment statement.*

A timed assignment statement in ADLIB is, in general, given **by**

assign X to Y delay t:

where "X" denotes either a value or an expression, "Y" denotes the net, and "t" denotes the time period. The net, which is also called a port, corresponds to a connecting point of a hardware module with the outside world. It can be of an arbitrary data type and contains the contribution of the hardware module to the value of the connection, The type of the value of "X" or the evaluation result of the expression "X" has to be compatible with the type of the net "Y". The time "t" can be either **an** expression or a value in the simulated time units.

The semantics of a timed assignment statement of ADLIB is that the net "Y" will have the value "X" after "t" simulated time units from the current simulated time. If "X" is given as an expression, it is evaluated at the current simulated time. This semantics is called anticipatory because execution of an assignment statement anticipates the value of the net "Y" in "t" time units in the future, and cannot be overridden prior to "t" becoming the current simulated time.

The semantics of a hardware description language is usually implemented by an event driven simulator. In order to help understand different approaches to the timing description, we briefly review the mechanism of the event driven simulator.

The event-driven simulation of a hardware description is carried out by iterating the following three steps. Step 1 is the execution of behavior descriptions of components constituting the hardware being simulated. Execution of **each component description generates some projected events, which are** actually generated by executing assignment statements in the component description. Projected events represent predictions of component contributions to the outside world. In step 2, projected events are placed in the event list. Although the placement can be implemented in various ways, this paper assumes a unique event list where events are ordered according to the simulated time at which events are intended to affect the environment. Step 3 consists of scheduling the earliest event (with respect to the simulated time) from **the** event list to become an actual event which means to actually influence the environment. The environment takes into account the contribution represented by the event.

After taking into account all events associated with a given simulated time, step 1 is repeated, for all descriptions which qualify. A description qualifies for step 1 if it is not suspended* and if any of its inputs were affected by step 3. At each unit of simulated time, all active (i.e., not suspended) descriptions with new input values are executed. The execution takes into account the "latest" contributions of all other descriptions.

Execution of a component description can trigger the execution of other component descriptions at the same simulated time. This is because predictions can be made with zero delay. Also, global and local activation conditions can change due to contributions made by component descriptions in step two. In order to deal with this problem, an infinitesimal time delay called delta *delay* is assumed for zero delay in the actual simulation. All events which are evaluated during the same step two, are said to belong to the same *delta simulated rime.* There are events predicted to happen at the same simulated time but at different delta simulated time.

One of the main difficulties in describing the timing characteristics of hardware is the fact that the description should generate an event only when it possesses enough information. Present hardware description languages do not enable a designer to write descriptions which wait for enough information (on input ports) before generating an event on an output port.

Anticipatory semantics, as embodied in the ADLIB timed assignment, permits description of timing delays. It is, however, inadequate for expressing timing behavior based on the time duration of an input signal. Preemptive semantics of timed assignment (defined in Chapter **4)** was introduced to express such behavior. However, the effect of a timed assignment under preemptive semantics is not known when the assignment is executed-it may be preempted later. Consequently semantics of behavior descriptions depend on assignment statements which will be executed in the future. As the possibility **of** more event preemptions increases, it gets more difficult to read and understand the behavior. In fact, preemptive semantics violates two fundamental principles of language **design—** programming languages or simulation languages-namely: (1) the effect of a statement should be a function of entities mentioned explicitly 'in the statement, and (2) the effect of a statement should be

[2] **A description** can be suspended by means of various mechanisms, for example, global activation conditions such as the "upon" construct or local activation conditions such as the "waitfor" construct in the case of **ADLIB** and HHDL. HHDL is the successor of ADLIB.

known upon termination of its execution. These principles are important in understanding models, and in laying any future foundation for mathematically proving their correctness. When these principles are violated, such as in the case of shared pointer structures, aliasing, variables shared between concurrent processes, etc, there are serious problems regarding readability and proof of correctness issues. Similarly, the effect of a preemptive assignment is often misunderstood and its use frequently results in incorrect modeling of designs.

In this paper, we review the current approaches to timing description by timed assignment with anticipatory or preemptive semantics. We will show how correct descriptions can be written, with difficulty, using these semantics. Finally, we will discuss a new language construct, called timing qualifiers, for expressing timing behavior.

2. Problem with Anticipatory Semantics

This chapter will show the problem associated with the anticipatory semantics for timing constructs in conventional hardware description languages such as ADLIB and HHDL by using the inverter example in Figure 2-1. This inverter example will be used throughout this paper to compare different approaches to timing description.

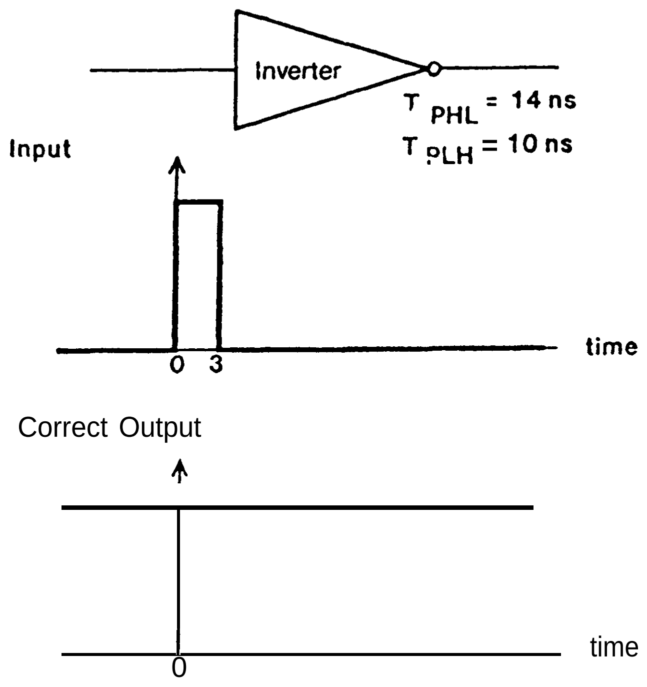


Figure 2-1: Inverter with different fall and rise propagation delays

The relevant characteristic of this inverter is that its propagation time for the output transition from high to low, T_{PHL} (= **14** ns), is longer than the propagation time for the low to high transition, T_{PLH} (= 10 ns). Their difference, 4 ns, represents an inertial delay for a positive pulse. One of the important requirements for this inverter is that a positive pulse applied to the input whose duration is shorter than **4** ns should not affect the output.

The description in pidgin ADLIB of the inverter in Figure 2-1 is not completely straightforward. The most obvious use of the ADLIB timed assignment would lead to the representation given in Figure 2-2.

The body of the above inverter description is executed whenever there is a change at the input "inport". Suppose the input changes from low to high at time t = 0 as shown in Figure 2-1. The statement labeled L1 will be executed, and consequently the output of the inverter will be scheduled

```
nettype two-value-net = (high, low);
comptype inverter:
    inward   inport: two-value-net;
    outward  outport: two-value-net;
begin
    if inport = high then
L1:     assign 1 ow  to outport delay 14
    else
L2:     assign high to outport delay 10:
end: {inverter}
```

Figure 2-2: Behavior model of inverter in ADLIB

to receive the value low at time t = 14. For the input change from high to low at t = 3, the statement at label L2 will be executed. As a result, the output is scheduled to receive the value high at time t = 13. Note that t = 13 is obtained by adding the delay 10 to the current simulated time t = 3. At this point, the scheduler has two events, El at t = 14 and E2 at t = 13. In this case, E2 is first executed at t = 13 followed by El at t = 14. Consequently, the simulation 'result will give the low as the value of the output at t = 14 although the correct output should be high. Therefore, the model in Figure 2-2 is inaccurate.

Four techniques to enable the correct description of the inverter will be discussed and compared in the following chapters. These techniques are use of virtual hardware, use of preemptive assignment statement, use of “waitfor” statement, and use of timing qualifiers.

3. Using Virtual Hardware

This chapter discusses the first approach to achieving correct simulations in spite of using anticipatory assignment statements. This solution relies on the concept of *virtual hardware* in the behavior model.

```
type      two-value = (high, low);
nettype  two-value-net = .two_value;
with  UTLpackage;
          {Utility package containing UTLactivePin to check if
           a net is changed or not}
comptype  inverter:
    inward          inport: two-value-net;
    outward        outport: two-value-net;'
    internal  pseudoPort: two-value-net:
    var                sent:  boolean;
begin
    if  (UTLactivePin(inport)  and  (inport  =  low))  then
        begin
            assign high to outport delay 10;
            sent := false:
        end
    else  if  UTLactivePin( inport) then
        begin
            assign low to pseudoPort delay 4;
            sent   :=   true:   .
        end
    else if sent then .
        begin      .     .
            assign low to outport delay 10;
            sent := false;
        end
end; {inverter}
```

Figure 3- 1: Correct behavior model of inverter using virtual hardware

Figure 3-l shows the revised model of the inverter in ADLIB with virtual ports which is a fictitious port and a part of the virtual hardware. "UTLactivePin" is the boolean function which returns true only if a given net is changed since the last execution of the inverter description. This function is included in the utility package called "UTLpackage" which is imported from the library. In Figure 3-1, a virtual port "pseudoPort" is introduced in addition to "inport" and “outport”. The virtual port "pseudoPort" is not connected to any external source. The model in Figure 3-l first identifies the exact transition. When the transition at "inport" is high to low at t = 0, "outport" is scheduled to receive the value low at t = 10, and the boolean variable “sent” is set to false. For the low to high transition at "inport" at time t = 0, the model sets “sent” to true and forces a new signal value on the virtual port "pseudoPort" at t = 4, where 4 is the difference between the falling and the rising propagation delays. The new signal value will arrive at the virtual port "pseudoPort" at t = 4. If there is no transition at "inport" between t = 0 and t = 4, i.e., the input remains high, the model is self-activated at t = 4 and a high is scheduled at “outport” at t = 14. Should a transition occur between t = 0 and t = 4, the boolean variable “sent” will be reset to false and the signal forced on "pseudoPort" will be ignored, and thus the possible incorrect result is prevented.

This approach achieves the non-anticipatory semantics by using virtual ports which enable the model to schedule its own activation. Until a model has all the necessary inputs to ensure. that an event scheduled in the future will be actually executed, the value of the virtual port is temporarily stored. It is used to activate the model at a later time, when the input values are adequate to determine the status of the event.

However, the model based on this approach is complicated due to the virtual hardware. Moreover, there is no separation between the virtual part of the model and the actual part. An architecture implementing this behavior would probably need to justify deletion (optimization) of the virtual parts, or else implement them,

4. Preemptive Semantics

The second approach is to change the semantics of the timed assignment statement from anticipatory to preemptive. Under preemptive semantics, predictions of future events are allowed to be preempted by other predictions. Some hardware description languages have adopted preemptive semantics. For example, the earlier version of VHDL (version 5.0) [5] used anticipatory semantics, and its disadvantages were discussed in a critique of VHDL [7]. The latest version of VHDL (version 7.2) [6] has changed to preemptive semantics. VHDL supports two different delays, transport delay and *inertial delay,* both of which have preemptive semantics.

A timed assignment statement is called a *signal assignment* in VHDL. A signal assignment statement with a transport time delay has the following syntactic form:

```
        Y <= transport X after t;
```

The above statement means that the value of the expression "X" is assigned to the port "Y" after•the transport time delay "t". However, the event projected by the above assignment can preempt other projected events according to the following semantics. It is assumed that an event E is generated as the result of executing the above timed assignment to the port "Y". If there exist events at "Y" caused by previously executed assignments and these events are projected to actually occur later than the event E, then the event E preempts these events. In other words, the preemptive semantics means that a later decision affecting an earlier future time preempts an earlier decision affecting a later time in the future.

Some devices are described more easily using preemptive semantics. than anticipatory semantics. For example, Figure 4- 1 shows a model of the inverter in Figure 2-1, which is described in VHDL (version 7.2) using transport delay.

Figure 4-1 consists of three different program units: a package for declaration of types, an entity interface, and an entity body. The package “type-declaration” provides the type declaration for two-value-net, which is maintained in the library. The entity interface imports the type declaration from the library, and uses it in declaring ports. The entity body contains a behavior description of the inverter.

Consider the previous case in which the anticipatory model (Figure 2-i) was incorrect. Suppose that a positive pulse as shown in Figure 2-l is apptied to the input of the inverter of Figure 4-1. At time t = 0, the input changes from low to high and statement L1 is executed. This generates event El,

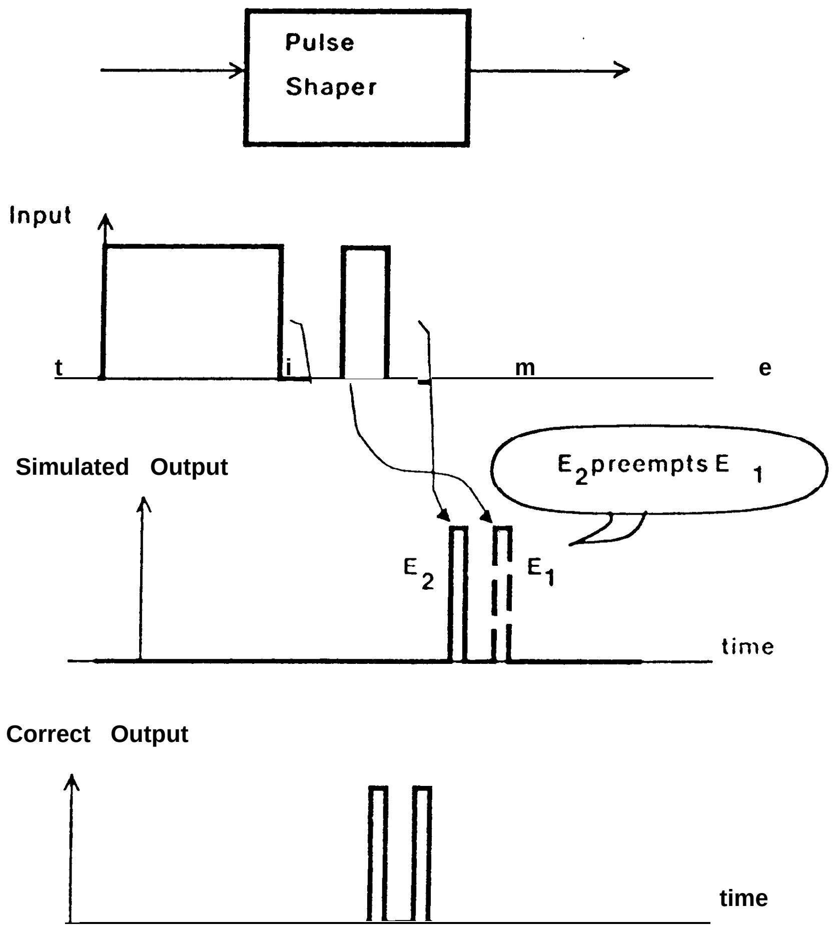
```
-- Package declaration
package type-declaration Is
    type
        two-value-net is (low, high):
end type-decl arat ion ;


-- Entity interface of inverter
with package type-declaration:
use type-declaration:
entity inverter
    (inport : in  two-value-net;
     outport: out two-value-net)
is
end inverter:


-- Entity body of inverter
architecture behavior-view of inverter Is
    block
    begin
L1:     outport <=  transport 1 ow  after 14 when inport = high else
L2:                                high after 10;
    end block ;
end behavior-view;
```

Figure 4- 1: Behavior model of inverter in VHDL

which sets the output of the inverter to low at time t = 14. At time t = 3, the input changes from high to low and statement L2 is executed. This generates event E2, which sets the inverter output to high at time t = 13. However, the projected time of El (t = 14) is later than that of E2 (t = 13). By the preemptive semantics of transport delay, event El is preempted and event E2 takes its place in the event list. Therefore, the behavior model wilt be correct in this case.

Unfortunately, preemption of events fails to be an improvement over anticipatory semantics in all cases. In fact, some devices are easier described using anticipatory semantics than preemptive' semantics. As an example of this, consider the pulse shaper presented in Figures 4-2 and Figure 4-3. The model is described In VHDL.

The pulse shaper in Figure 4-2 has an input called "inport" and an output port called “outport”. Every low to high transition applied at "inport" produces a positive pulse with the fixed width of a single time unit at' “outport” after a certain amount of delay from the high to tow transition at “inport”. The delay is equal to the time period during which the input signal stays high. To simplify the behavior description, we assume that overlapping of pulses at “outport” does not occur.



Figu re 4- 2: Pulse shaper behavior

```
-- Entity interface
withpackage Type-Declaration;
use Type-Declaration;
entity Pulse-Shaper
    (inport  : in two-value-net;
     outport : out two-value-net)
IS
end Pulse-Shaper:


-- Entity body
architecture behavior-view of Pulse-Shaper is
    block
        constant width: integer := 10 ns;
        variable pulseOrigin : simulated-time;
            -- simulated_time is a predefined type

        -- TIME is a primitive function which returns the current
        -- simulated time.
    begin
        process (inport)
        begin
            case inport is
                when low =>
                    outport <= transport high after (TIME-pulseorigin),
                                         low after (TIME-pulseOrigin+width):
                when high => pulseOrigin := TIME:
            end case:
        end process ;
    end block ;
end behavior-view;
```

Figure 4-3: Behavior model for the pulse shaper

When a particular sequence of pulses as shown in Figure 4-2 is applied to "inport" of the pulse shaper, the behavior model in Figure 4-3 will not provide the correct output. We assume that the first pulse starts at the time unit 0. For the first input pulse of the width of four time units, the behavior model will generate the event E, at "outport" after the delay of four time units from the high to low transition of the first pulse. Thus, E, will be projected at the time unit 8. For the second pulse with the width of one time unit, the behavior model will project the event E_2 at the time unit 6 which is obtained by adding the delay of one time unit to the time point of the high to low transition of the second pulse. Then, by the preemptive semantics, the event E_2 will preempt the event E,. Therefore, the actual simulation will provide a wrong result in this case. If anticipatory semantics is assumed, it is easily seen that the model is correct.

5. Stabilify of Expressions over Time intervals

A third approach to representing time dependency is to use anticipatory timed assignment statements in conjunction with language features which require the evaluation of expressions over time intervals. In many instances it is important in constructing a design to have an ability to express that the value of an expression must remain stable over a given time interval. The stability of expressions over time intervals can be represented by different language constructs.

In this chapter, we first discuss expressing the stability of expressions over time intervals by using the “waitfor” construct provided in ADLIB, and discuss the problems with this approach. Second, we propose a new language construct called *timing qualifiers.* We give a simple illustration of how timing qualifiers can be introduced into a hardware design language, and then provide examples showing the expressive power in comparison with the language without qualifiers. Finally we mention some issues concerning the semantics and implementation of timing qualifiers.

5.1 Stability of Expressions Using "Waitfor"

The “waitfor” statement-of ADLIB (or HHDL) has the following syntactic form:

```
<waitfor statement> ::= waitfor <boolean expression>;
```

Upon execution of a “waitfor” statement, the execution of the given model description is suspended until the associated condition specified, by <boolean expression> becomes true. In order to use the “waitfor” statement to check the stability of an expression over a time interval, the following steps are required. First, the origin of the time interval must be stored. Second, the condition of the “waitfor” statement must be built as a disjunction of two boolean expressions. One is obtained by equating the current time to the end of the time interval. The other boolean expression is obtained by equating the expression to be checked for stability to its value evaluated-at the origin of the time interval. Finally, a third step consists of checking whether the condition of the “waitfor” was met because the time interval is expired or because the expression was not stable over the time interval.

Figure 5-1 shows an ADLIB description of the inverter in Figure 2-2. This description uses the “waitfor” statement to check the stability of the input after a low to high transition.

The invet-ter description in Figure 5-1 is explained as follows. Each time the description is invoked, the current simulated time is saved in the local variable “origin”. If the invocation is due to a high to low transition of the input, the description predicts a high value at the output port in 10 units of

```
nettype two-value-net = (high, low):
comptype inverter;
    inward   inport: two-value-net;
    outward  outport: two-value-net:
    var      origin: simulatedTime;
begin
    origin := TIME;
    if inport = low then
L1:     assign high to outport delay 10;
    else
L2:     begin
            waitfor inport = low or TIME = origin + 4;
            if inport = high then
                assign low to outport delay 10:
        end
end: {inverter}
```

Figure 5-1: Behavior model of inverter using a “waitfor” statement

simulated time after the current time. In this case there is enough information to make such a prediction. No change in the input can possibly affect this prediction.

If the invocation is due to a low to high transition of the input, there is not enough information to predict an output transition. The description will wait (statement 12) either for a high to low input transition or for simulated time to progress four units. If the condition of the “waitfor” statement is met due to a change in the input, no transition in the output needs to be predicted because the output will not change value. In this case, the width of the input pulse was too narrow. If the condition of the “waitfor” statement was met because simulated time progressed four units without any change in the input, then a low value needs to be predicted at the output in 10 units from the time at which the condition is met. At this moment (four simulated time units after the input changed from low to high) no change in the input can possibly affect the predicted output transition to low.

The “waitfor” construct is not available in some hardware description languages such as VHDL. Also, the “waitfor” construct was originally intended for purposes other than stable evaluation; for example, description of synchronization among components. In fact, application of the “waitfor” construct to checking the stability of expressions over time intervals is prone to programming errors. The use of the “waitfor” construct requires low-level programming details in the description. For example, as shown in Figure 5-1, the user has to program low level details such as saving the current simulated time in origin, building a compound condition to be used in the “waitfor” statement, and checking which alternative made the “waitfor” condition true (i.e., the branch, “if inport = high

then..."). A more promising approach is to design a higher level language construct that requires only specification of time interval over which stability is to be measured, and does not force the user to program the actual measurement of the time interval.

5.2 Stability of Expressions Using Timing Qualifiers

For the purpose of specifying the stability of expressions over time intervals, we introduce a new language construct called a timing qualifier which applies to expressions. An expression with a timing qualifier is called a *qualified expression.* A qualified expression has the following syntactic form:

e_1 **during** $[e_2, e_3]$ **else** e_4

In the above, e, and e_4 are expressions of any type, and e_2 and e_3 are expressions whose values must be of the type integer or real. None of these expressions contains a qualifier. When control reaches the statement containing the qualified expression, the expressions e,, e_2, e_3 and e_4, are all evaluated. Then, the value of the qualified expression is e, if the value of e, remains constant during the simulated time interval from e_2 to e_3; otherwise, the value is e_4. Evaluation ceases whenever e, changes value or simulated time reaches the value e_3.

The bracketed expression $[e_2, e_3]$ denotes the time interval of the qualification. The time interval is bounded by the starting time point given by (current-time + e,) and the ending time point given by (current_time + e_3). The value of e_3 is normally larger than that of e_2. However, if the value of e_3 is less than or equal to the value of e_2, the qualified expression becomes a normal expression without qualification, i.e., e,. A negative value of e_2 or e_3 indicates the time point in the past from the current time. When the starting point of a time interval is at the current time, the expression e_2 may be omitted, and then the time interval can be denoted by only the future ending time e_3.

For example, the qualified expression given by

```
2*(X during t else 0)
```

is interpreted as follows. If the value of the variable X is stable for t units from the current time, the value of the above expression is (2.X); otherwise, the **value is 0.**

Essentially, the concept of qualified expression is a language design concept. It can be incorporated into languages in various ways, some very restricted, some more general. It requires strict scoping rules to define regions of applicability of timing qualifiers, consistent nesting of qualified expressions, etc. Restrictions must also be placed on where qualified expressions may occur in statements in order for them to make sense. The major advantage is that qualified expressions have a precise semantics. They require the value of an expression to be checked over a time interval during simulation. When their execution is completed, the values of qualified expressions and the computational effect of the statements in which they occur is known. This should be contrasted to the semantics of preemptive assignments, where it is not known what will happen when an assignment is executed.

5.3 Timed Assignment with Qualified Expressions

Below we illustrate a very simple extension to a timed assignment statement, whereby a restricted form of qualified expressions is used to define a new kind of assignment called a guarded assignment statement. Qualified expressions are allowed in guards to determine whether or not a timed assignment should be executed.

For qualified expressions placed in guards, application of timing qualifiers is restricted to only boolean expressions. We can adopt a simplified form of, qualified expression for the case of boolean expressions. We require that the starting time of a time interval, e_2, is always the current simulated time and the value of the else part, e_4, is always false. The qualified expression is

e_1 during [0, e_2] else 'false.

Since the starting time of the time interval and the else part are actually constants, they can be omitted, and the above expression is' simplified to the following form which is called a *qualified boolean expression:*

e_1 during e_2

Note that e, denotes a boolean expression and e_2 denotes-an expression returning a numeric value. When the ending time of the time interval is given by zero or a negative value, the above expression becomes a normal boolean expression e,. Thus, the normal boolean expression is a special form of a qualified boolean expression.

The semantics of a qualified boolean expression can be derived directly from the semantics discussed in Section 5.2 as follows. Both of expressions e, and e_2 are evaluated at the current simulated time, and the initial boolean value of e, is checked to be stable until the end of the time interval given by e_2. When the initial value of e, is true, if it remains true over the time interval, the value of the qualified expression is true; otherwise, it is false. Whenever the value of e, is false, the qualified expression evaluates to false immediately. The **evaluation of a qualified boolean expression** can terminate as soon as the boolean expression e, **evaluates to false during the time interval.**

Syntactic restrictions are also required on the occurrences of qualifiers in expressions. The qualifier **during** has the least precedence in terms of association in an expression, and a qualified boolean expression cannot contain another qualified expression, i.e., nesting of qualification is not allowed. A qualified boolean expression can be negated by prefixing the logical operator **not.** For example, the expression "**not (e, during** t,)" means logical negation of the evaluation result of the expression **"(e_1 during t_1)".**

In our example of a guarded timed assignment, a qualified boolean expression may appear in the guard. The guarded assignment statement has the following syntax:

```
<guarded assignment statement> ::=
            when <guard> => <timed assignment statement>
<guard> : : = <qualified' boolean expression>
                {<logical operator> <qual ified boolean expression>}
<qualified boolean expression> ::=
            [not] <boolean expression> during <expression>
<logical operator> ::= and | or
```

In the above BNF notation, bold faced words indicate reserved words, **"|"** denotes alternatives, the part enclosed by **"["** and **"]"** means optional, and the part enclosed by **"{ "** and **"}"** means zero or more occurrences.

Execution of a guarded assignment proceeds as follows. The guard is evaluated at each unit of simulated time specified by the time interval. The assignment is executed immediately after the time interval has elapsed only if the initial value of the boolean expression is true and its value remains true over the time interval. If the guard is evaluated to false at any simulated time instance during the time interval, then the assignment statement is not executed.

5.4 Guarded Assignments in Hardware Description

We now consider a fragment of a hardware description language illustrating the use of anticipatory semantics in timed assignments and timing qualifiers in the guards of guarded timed assignments. In this language, we permit several guarded assignment statements in a hardware description by means of a *sequential* **select** statement, which has the following form:

```
select
    when g1 => S,:
    when g2 □  > S2;
```

```
        when $g_n$ => S,:
    end select;
```

In the above, $g_1, g_2, \ldots g_n$ denote guards consisting of qualified boolean expressions, and each of S,, $S_2, \ldots, S_n$ represents a sequence of one or more timed assignment statements.

When the control reaches the **select** statement during simulation, all the guards are evaluated simultaneously. As soon as a guard evaluates to true, evaluation of guards stops and only assignment statements belonging to the guard with the value true will be executed. Assignment statements belonging to other guards will not be executed. If more than one guard is true at the same simulated time, one guard is selected arbitrarily and assignment statements belonging to that guard are all executed. If none of the guards is true, no assignment statement will be executed. (Note that other forms of **select** can be defined, for example *parallel* **select,** allowing concurrent execution of timed assignments when several guards are true.)

Figure 5-2 shows the behavior model of the inverter described in the proposed language.

```
entity inverter is

    type two-value-net is (high,' low):
    port
        inport : in   two-value-net:
        outport: out  two-value-net;
    end port ;

    select
L1:     when inport = low  => outport  := high  after 10:
L2:     when inport = high during 4 => outport :=  low after 10;
    end select;

end inverter:
```

Figure 5-2: Behavior model of inverter using a qualified conditional assignment

In Figure 5-2, the entity description of the inverter consists of four parts: entity declaration of the inverter, type declaration, port declaration, and actual behavior description using a select statement.

The timed assignment. statement L1 has the conventional meaning of anticipatory semantics. The guarded statement L2 is interpreted as follows. If the input to "inport" remains at high for a time interval of 4 ns starting at the current time, then low is assigned to the output at the time t = (current-time t 14). As a result, an assignment to the output, i.e. an event, is generated only when the model has received adequate input information and determined that the input must yield an output. The reader will notice that the low level programming details required by the use of "waitfor" in Figure 5-1 are not presented in Figure 5-2.

Commentary on Figures 4-1 and 5-2:
The model in Figure 5-2 and the VHDL model in Figure 4-1 are similar in syntactic complexity. However, the most meaningful comparison is the complexity of checking if the models are correct. We will try this informally for the one input case that has plagued us throughout: the input to **"inport"** goes from low to high at t = 0, and from high to low at t = 3.

Consider Figure 5-2. The first input change will result in the guarded assignment at L2 being evaluated for 4 time units. The second input change results in this guard evaluating to false. Consequently, the device does not change its output in response to the first input. Consider Figure 4-1. The first input change results in an assignment of low to "outport" at t = 14. This, we know, is wrong. Therefore, we must check that this will be preempted. At t = 3, the second input change results in an assignment of high to the outport at t = 13, which will preempt the wrong assignment. In other cases, where the input change goes from high to low, the first assignment to "outport" is correct; but we must continue to check that it cannot be preempted by an incorrect one.

5.5 Specification of a Traffic Controller .

The example of a traffic controller [8] may be used to illustrate some other important points concerning the introduction of timing 'qualifiers into a hardware design language. In addition to the **select** statement, we introduce the following features to our language fragment.

The first point is to provide an **activation** statement to declare the list of elements which cause activation of an entity specification. This list usually consists of input ports and/or the entity state. The semantics of activation is that whenever the values of at least one of the elements in the activation list is changed, the entity specification will be **executed.**

Second, we provide a language facility to denote the abstract state of an entity, which is one of the fundamental issues in specifying a hardware entity having history-dependent sequential behavior. The abstract state of a hardware entity is denoted by the attribute of an entity, *entity_name*'STATE, which is provided by the language. This attribute is used to denote the current state of the entity at any time. At the entity specification level, this name may be used to define behavior that depends on the internal state of the entity without revealing the structure of the state. The type of entity states is denoted by *entity_name*'**STATE_TYPE.**

Third, we provide another predefined attribute that denotes the simulated time. The current simulated time of an entity is denoted by the attribute, *entity_name*'**Time.** Also, the simulated time at which an input port was previously excited, is denoted by *input_name*'**LastChange.** These predefined attributes are useful in constructing time intervals for qualifiers.

Design Requirement:

The traffic controller in [8] (pages 85-88) has the following design requirements:

- The highway light should stay green for at least **"LongTimeOut".**
- The highway light stays green if there is no car on the farmroad.
- The highway light stays yellow for **"ShortTimeOut".** .
- The **farmroad** light does not stay green if there is no car on the farmroad.
- The **farmroad** light stays green for at most **"LongTimeOut".**
- The **farmroad** light stays yellow for **"ShortTimeOut".**
- Highway and **farmroad** lights may not be green at the same time.

The top level specification of the traffic controller is presented in Figure 5-3. This specification is described in the overall framework of VHDL, but using the additional language features discussed above.

Commentary on **Figure 5-3:**

In the top-level entity specification of the traffic controller, constants for the time-out period,. **"LongTimeOut"** and **"ShortTimeOut",** are declared as generic parameters. Thus, their actual values are assigned when the specification is instantiated. Declaration of the entity name **"Traffic_Controller"** and the enumerated type “traffic-light-color” is followed by declaration of ports where input and output ports are declared as typed objects with the directionality of **a** signal flow.

The abstract entity state of the traffic controller is the predefined attribute of an entity, **"Traffic_Controller'STATE".** The entity state type is declared as an enumeration of four distinguishable states, **"Hw_Green", "Hw_Yellow", "Fm_Green",** and “Fm-Yellow”. The activation

```
generic
    LongTimeOut, ShortTimeOut: INTEGER;
entity Traffic-Controller is

    type traffic-light-color is-(red, green' yellow);

    port
        car-on-fmrd, reset: in BOOLEAN;
        hw-light, fm-light: out traffic-light-color:
    end port;

    Traffic-ControllerSTATE-TYPE is
        (Hw-Green, Hw-Yellow, Fm-Green, Fm-Yellow):

    activation car-on-fmrd, reset, Traffic_Controller'STATE;

    rename
        STATE is Traffic-ControllerSTATE;
        TIME is Traffic_Controller'TIME;
    end rename:

    select
        when reset =>
            STATE := Hw-Green;
            hw-light := green;
            fm-light := red:
        when car_on_fmrd and
             (STATE=Hw_Green during LongTimeOut-(TIME-STATE'LastChange)) =>
            STATE := Hw-Yellow;
            hw-light := yellow;
        when STATE=Hw_Yellow during ShortTimeOut =>
            STATE := Fm_Green;
            hw-light := red;
            fm-light := green:
        when (nbt car-on-fmrd and (STATE=Fm_Green)) or
              (STATE=Fm_Green during LongTimeOut-(TIME-STATE'LastChange)) =>
            STATE := Fm-Yellow:
            fm-light := yellow:
        when STATE=Fm_Yellow during ShortTimeOut =>
            STATE := Hw-Green;'
            fm-light := red:
            hw-light := green:
    end select;

end Traffic-Controller;
```

Figure 5-3: Entity specification of the traffic controller

statement declares that any change occurred at “STATE” and ports "car_on_fmrd" and “reset” will activate the traffic controller. The **rename** statement is used to represent a long name by a simplified name. In this example, "STATE" represents "Traffic_Controller'STATE" and "TIME" represents "Traffic_Controller'TIME".

The **select** statement following the activation list actually defines the behavior. There are five guards. In the specification, "STATE'LastChange" denotes the predefined attribute **"LastChange"** of the entity state, and it keeps the simulated time at which the most recent change of the entity state occurred.

Proof of **Correctness of Figure 5-3:**
In this section, we will show how the correctness of the model in Figure 5-3 can be proved. Although an informal method is used for proving the correctness, it **can be** developed as a formal verification method. As an example, we will prove that the model satisfies the first design requirement on Page 19:

"The highway light should stay green for at least **LongtimeOut".**

It follows from the description of the traffic controller that **hw_light =** green if and only if STATE = Hw-Green. Therefore, we will only prove that the state of the mode! stays STATE = Hw-Green, for at least LongTimeOut. To prove this, we will consider an arbitrary state transition from Hw_Green to any other state. We wilt show that the given transition can occur at a simulated time which is at least LongTimeOut (in simulated time unit) after the time indicated by STATE'LastChange.

Let us consider an activation, which results in change of state from Hw-Green to any other state. Let t_a be the simulated time at which the activation starts and t_c be the simulated time at which the state changes its value from Hw_Green to another value, during the given activation.

It follows from the description **of the** traffic controller that the state cannot change value from Hw_Green, unless the second guard evaluates to true. This is because all the other guards can only be true in other states.

Let us consider the timing qualifier of the second guard. Let W_a be the width of its time interval evaluated at time t,.

$$W_a = \text{LongTimeOut} - (t_a - \text{STATE'LastChange}) \quad (5.1)$$

From the definition of the timing qualifier, if $W_a < 0$ then $t_c = t_a$, and if $W_a \geq 0$ then $t_c = t_a + W_a$. In **both cases,**

$$t_c \geq t_a + W_a. \quad (5.2)$$

By replacing W_a in Equation (5.2) with Equation **(5.1),**

$$t_c \geq t_a + (\text{LongTimeOut} - (t_a - \text{STATE'LastChange})),$$

which is equivalent to $t_c - \text{STATE'LastChange} \geq \text{LongTimeOut}$.

The rules of proof embodied in this informal argument involve the normal rules of quantified logic and rules defining the semantics of **during,** guarded assignments, and **select** statements. These rules can be formally defined, providing a basis for automated analysis of models.

The reader should now be able to check other design requirements of the traffic controller. For user's convenience in proving the correctness of the model, we will provide the following corollaries which are derived from the definition of the timing qualifier.

Corollary 1: If a qualified expression evaluates to true, it is the case that $t_c - t_a \geq W_a$, where the notations are same as the above.

Proof: Proof of Corollary 1 was already given in the above proof (see the derivation to Equation (5.2)).

Corollary 2: If the expression W, representing the width of the time interval of the qualified expression in a guard, has the form W = K - T, where K is a constant during evaluation of the qualified expression and T denotes the current simulated time, and if the qualified expression evaluates to true at time t_c, then $t_c \geq$ K.

Proof: Using the same notations as in Corollary 1, W_a = K - t_a. From corollary 1, it follows that $t_c - t_a \geq K - t_a$, which leads to $t_c \geq$ K.

The reader can see now how Corollary 2 can be used to prove the correctness of the traffic controller model for given design requirements.

6. Concluding Remarks

Timing constructs in modern-hardware description languages are inadequate for various reasons. Anticipatory timed assignment, as in ADLIB, cannot be used to express common kinds of timing behavior. Preemptive timed assignment, as in VHDL, is also inadequate in some cases. Moreover, it is a highly misleading design construct because its effect (or outcome) depends on other preemptive timed assignments, and is not known when it is executed. Use of preemptive assignment may itself be a frequent source of modeling errors. It is unlikely that formal axiomatic semantics can be defined for this construct. This precludes the application of automated proof techniques to correctness of hardware descriptions.

The time qualified expression has been suggested as an alternative design language concept for expressing timing behavior. It is argued that expressive power of languages with this construct will be at least equal to previous languages, that designs will be more clearly expressed, and their correctness can be analyzed by formal proof methods (in addition to simulation). Language design issues, and the methodology of using timing qualifiers remain to be investigated.

Acknowledgment

We wish to thank Sumit Ghosh for his involvement and collaboration at the beginning of this study, Also, our thanks are due to Benoit Gennart for discussions during later stages.

References

1. D. D. Hill, *ADLIB User's Manual,* Technical Report # 177, Computer Systems Laboratory, Stanford University, Aug. 1977.

2. *DAB L Reference Manual,* Daisy System Corp., 1984.

3. *NHDL reference Manual,* SILVAR-LISCO, 1983.

4. *TEGAS Reference Manual,* G.E. CALMA.

5. *VHDL Language Reference Manual* (Version 5.0), Intermetrics, Aug, 1984.

6. *VHDL Language Reference Manual* (Version 7.2), Intermetrics, Aug, 1985.

7. D. C. Luckham, Y. Huh, S. Ghosh, and A. Stanculescu, *Analysis of the VHSIC Hardware Description language,* Technical Report, Computer Systems Laboratory, Stanford University, (In preparation).

8. C. Mead and L. Conway, *Introduction to VLSI Design,* Addison-Wesley, 1980.