P2VSIM: A SIMULATION AND VISUALIZATION TOOL FOR THE P2V COMPILER

A Thesis

by

OSCAR MICHAEL ALMEIDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2009

Major Subject: Computer Engineering

P2VSIM: A SIMULATION AND VISUALIZATION TOOL FOR THE P2V COMPILER

A Thesis

by

OSCAR MICHAEL ALMEIDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jyh-Charn (Steve) Liu
Committee Members,	Rabi Mahapatra
	Jiang Hu
Head of Department,	Valerie Taylor

May 2009

Major Subject: Computer Engineering

ABSTRACT

P2VSim: A Simulation and Visualization Tool for the P2V Compiler. (May 2009)

Oscar Michael Almeida, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Jyh-Charn (Steve) Liu

The Property Specification Language (PSL) is an IEEE standard which allows developers to specify precise behavioral properties of hardware designs. PSL assertions can be embedded within code written in hardware description languages (HDL) such as Verilog to monitor signals of interest. Debugging simulations at the register transfer level (RTL) is often required to verify the functionality of a design before synthesis. Traditional methods of RTL debugging can help locate failures, but do not necessarily immediately help in discovering the reasons for the failures. The P2VSim tool presents the ability to combine multiple Verilog signals not only instantaneously, but also across multiple clock cycles, producing a graphical display of the state of active PSL assertions in a given RTL simulation.

When using the P2VSim tool, users will write PSL assertions directly into their Verilog source files. After the tool searches for and loads the embedded assertions, execution trace monitors for the relevant Verilog signals are dynamically generated and written back into the Verilog source code. P2VSim then invokes an RTL simulator, Modelsim, to generate a simulation execution trace, requiring that the designer has some hardware or software testbench already in place. Next, the input PSL assertions are

parsed into time intervals that have logical and temporal properties. These intervals are to be displayed graphically when PSL property checking is performed. Finally, the user is allowed to step through simulation one cycle at a time, while the tool applies the simulation execution trace to the instantiated time intervals, performing PSL property checking at each clock cycle. From this, the user can witness the exact clock cycles when PSL assertions are satisfied or violated, along with the causes of such results.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

The Property Specification Language (PSL), developed by Accellera, is an IEEE standard intended to allow developers to specify precise behavioral properties of hardware designs [1]. PSL assertions can be embedded within code written in hardware description languages (HDL) such as Verilog and VHDL to monitor signals of interest. Such assertion-based verification can be performed not only in simulation, at the register transfer level (RTL), but also during execution on actual hardware after synthesis. Simulation provides not only the opportunity for design exploration and verification, but also a means for more rapid design iterations [2].

Debugging simulations at the RTL level is often required to verify the functionality of a design before synthesis. Traditional methods of RTL debugging, in which simulation results are superimposed on structural connectivity, can greatly help the designer locate failures, but do not necessarily immediately help in discovering the reasons for the failures [3]. A common practice is viewing simulation waveforms, in which any signal in the system can be observed.

The PSL-to-Verilog (P2V) compiler is a tool being developed that accepts PSL assertions as input and generates synthesizable Verilog modules capable of monitoring the desired RTL signals. The P2V compiler, being developed using Python, will support a subset of PSL operators which allow a user to precisely and succinctly specify behavioral temporal assertions.

This thesis follows the style of *Association for Computing Machinery (ACM) journals.*

Although this tool was originally intended for zero-overhead non-intrusive software program monitoring [4], this work builds upon P2V in that it can be expanded to include monitoring any Verilog signals.

This work introduces a graphical user interface (GUI) written in the C# programming language that accepts PSL assertions as input and transforms them into time intervals that can be viewed graphically in a cycle-accurate manner. After loading the input PSL assertions from the user's Verilog source code, Verilog execution trace monitors are dynamically created and placed back into the developer's Verilog source code by the P2VSim tool. The Verilog source code and execution trace monitors are simulated together using Modelsim, resulting in a simulation execution trace for all signals referenced in any of the PSL assertions. At this point, the tool statically parses the original PSL assertions into time intervals, and then simulation of the PSL assertions within the GUI can begin. At each simulation clock cycle, the PSL properties of each interval are checked and updated, and the results are displayed graphically for the user to visualize.

The remaining sections of this paper are as follows. First, background information is provided regarding the subset of PSL used in this work. Following this is a description of how PSL assertions can be represented graphically. Next, specific detail is provided regarding the implementation of the P2VSim tool. Finally, two tool usage examples are presented in a step by step fashion.

2. THE GRAPHICAL REPRESENTATION OF PSL ASSERTIONS

This work introduces the notion that it is useful to represent PSL assertions graphically. However, comprehending how the PSL assertions can be represented graphically first requires an understanding of the Property Specification Language. PSL contains several operators, both temporal and logical, that provide a means of representing behavioral requirements. Only a subset of the several PSL operators available is used in this work. When using the P2VSim tool, the syntax of a PSL assertion has the general form shown in Equation 1.

always (trigger → condition)	(1)

2.1 A Subset of the Property Specification Language

Let *trigger* and *condition* represent PSL properties, consisting of zero or more PSL operators that apply to at least one monitored Verilog signal. The PSL operator a*lways*, as shown in Equation 1, implies that every time the signal *trigger* is asserted, *condition* must be asserted in order to satisfy the overall PSL assertion. In this context, the term "asserted" represents a logic value of '1', HIGH, or TRUE. The → symbol means "implies". For example if *A implies B*, then if A is true, B must be true in order to satisfy the property. Shown on the next page in Table 1 are the PSL operators currently supported by P2VSim, along with their respective definitions. Table 2 shows the truth tables that apply to the logical PSL operators.

Table 1. Subset of PSL Operators Used in P2VSim

PSL Operator	Meaning
always	Every cycle for which *A* is true, *B* must also be true
implies	If *A* implies *B*, then if *A* is true, *B* must be true (denoted by →)
and	Logical *and*
or	Logical *or*
next_e [m:n] S	Condition *S* must be asserted at least once from cycles *m* to *n* (inclusive)
next_a [m:n] S	Condition *S* must be always be asserted from cycles *m* to *n* (inclusive)

Table 2. Truth Table for the Logical PSL Operators

A	B	A and B	A or B	A implies B
F	F	F	F	T
F	T	F	T	F
T	F	F	T	F
T	T	T	T	T

The PSL operator *next_e*, shown below in Equation 2, is defined such that the input signal *S* must be asserted at least once between the inclusive clock cycles *x* and *y* (in relation to the current clock cycle). The PSL operator *next_a*, shown in Equation 5, is similar to *next_e*, with the exception that the input signal *S* must *always* be asserted between the inclusive clock cycles *x* and *y*.

$$next\_e\ [x : y]\ S \mid x,y >= 0 \quad (2)$$

$$\text{next\_a } [x : y]\ S \mid x,y >= 0 \quad (3)$$

2.2 Graphical Representation of Temporal PSL Operators

Just as the logical PSL operators can be described using a truth table, this work introduces the concept that temporal PSL operators can be depicted using graphs. Timing diagrams can be composed of one or more graphed time intervals. Each time interval corresponds to either the *next_e* or *next_a* temporal logic PSL operators. To begin with a straight-forward example, consider the following PSL property.

next_e [10:15] S	(4)

Graphical characterization of the above property requires knowledge of four of its attributes in order to generate its corresponding timing diagram. The first attribute required is the *PSL operator*. This field must be set to *next_e* or *next_a*. Note that these are both temporal logic operators, and can inherently be represented on a timeline. The next attribute is the time interval's *start time*. This is an integer value corresponding to a simulation clock cycle used to determine when the temporal logic operator will become active, relative to some time zero. Third is the interval's *finish time*. Similar to the interval's *start time*, this integer value will determine at which clock cycle the temporal logic operator will become inactive. Lastly, the *condition*, denoted by *S* in Equation 4, is the combination of zero or more PSL operators that apply to at least one monitored Verilog signal, necessary to determine whether the PSL property is satisfied or violated.

In the example given in Equation 4, the *PSL operator* is *next_e*, the *start time* is 10, the *finish time* is 15, and the *condition* is some condition "S". Only the start time and finish time are needed to generate the graph, while the *PSL operator* and *condition* fields are necessary in order to determine how to apply the PSL property *next_e* to the condition *S*. Shown below in Figure 1 is a graph for *next_e [10:15] S*. The *interval number* is set to zero, as labeled, since it is the first an only interval in this PSL property timing diagram.

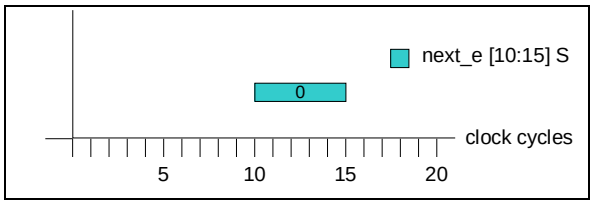


Figure 1. Timing diagram for the PSL property *next_e[10:15] S*.

In addition to the above PSL property, consider its counterpart, *next_a [10:15] S*. Figure 2 demonstrates the difference between *satisfied* and *violated* temporal PSL operators. The vertical dashed lines indicate clock cycles when the input signal *S* is true. In Figure 2(a), the *next_e* operator is satisfied because *S* is asserted at cycle 13. In Figure 2(b) however, the operator is violated because *S* is never asserted between inclusive cycles 10 and 15. In Figure 2(c), the *next_a* operator is satisfied because *S* is asserted during every cycle between 10 and 15 inclusive. However, in Figure 2(d), the operator is violated because *S* is not asserted at cycles 10, 12, and 15.

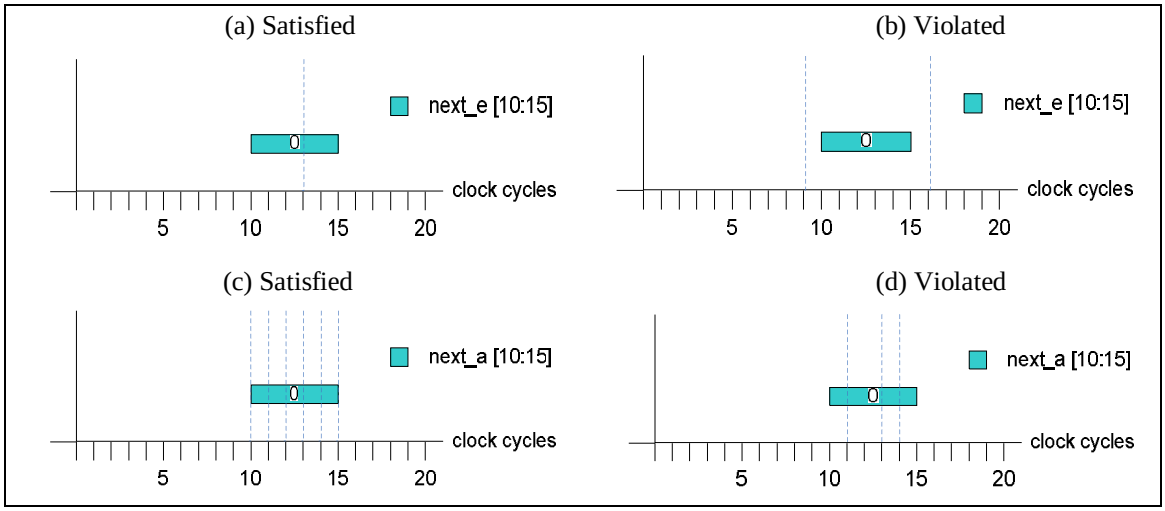


Figure 2. Differences between satisfied and violated PSL properties.

2.3 Nested Temporal PSL Operators

In introducing a more complex example, recall that *condition* was previously defined as a combination of zero or more PSL operators that apply to at least one monitored signal. Consider the following example in Equation 5, containing two *nested* PSL operators. Here, the *next_e* operator is defined to be the *temporal parent* of the *next_a* operator. A temporal logic operator is a temporal parent if its *condition* is another *next_e* or *next_a* operator. The presence of temporal parents requires the introduction of *absolute* and *local* start times and finish times. Absolute time in this sense is referenced against time zero (or the *enable*, from Equation 1), and local time is simply the values in the brackets of a temporal logic operator.

next_e [10:15] (next_a [4:6] T)	(5)

First, we examine the *next_e* operator. Its local start time is 10 and its local finish time is 15. Because this operator has no temporal parent interval, the absolute start and finish times are the same as the respective local start and finish times. The *next_a* operator, however, does have a temporal parent interval. Therefore, one instantiation of the *next_a* interval will be generated for each clock cycle that its parent, the *next_e* interval, is active. These times are from cycle 10 to cycle 15. The local start and finish times of each generated *next_a* interval are 4 and 6, respectively. To find the absolute start and finish time of each generated *next_a* interval, add the clock cycle number of interest of the parent interval to the local start and finish times of the child, as shown in Table 3. Effectively, PSL property *next_e[10:15] (next_a[4:6] T)* is logically equivalent to that given in Equation 6.

Table 3. Time Intervals for Nested Temporal Logic

Interval Number	Interval Contents
0	next_e[10:15]
1	next_a[14:16] T
2	next_a[15:17] T
3	next_a[16:18] T
4	next_a[17:19] T
5	next_a[18:20] T
6	next_a[19:21] T

(next_a[14:16]T) OR (next_a[15:17]T) OR (next_a[16:18]T) OR…OR (next_a[19:21]T) (6)

At cycle 16 the result of Interval 1 is known. Also, the partial result of Interval 0 is known, for cycle 10. In other words, if Interval 1 is satisfied, then its respective input signal to Interval 0 is satisfied. Similarly at cycle 17, Interval 2's result is known, which is the partial result of Interval 0 at cycle 11. Finally, at cycle 21, the result for Interval 6 is known, corresponding to Interval 0 at cycle 15. At this time, it is known whether Interval 0 has resulted in satisfaction or violation. It's possible for the overall result to be realized before cycle 15. For example, if signal *T* were asserted at cycles 14 through 16, then interval 1 would be satisfied, and at this point so would interval 0. Consider the static timing diagram in Figure 3 for the PSL property in Equation 5. Notice that the start and finish times match those given in Table 3.

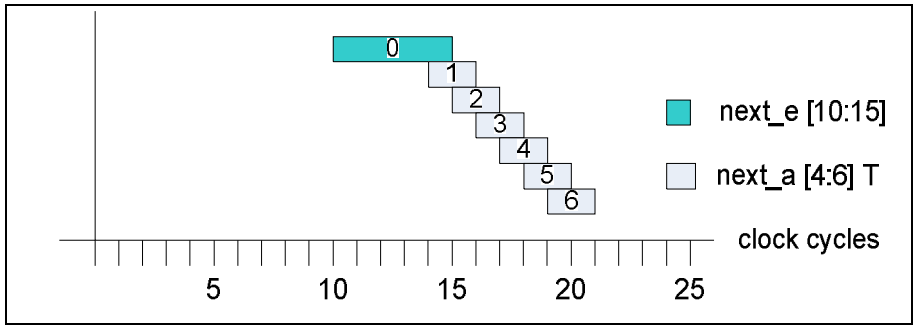


Figure 3. Timing diagram for PSL property *next_e[10:15] (next_a[4:6])*.

Shown in Figures 4(a) and 4(b) are examples of the same input PSL property being satisfied and violated, respectively. In each diagram, the vertical dashed lines indicate clock cycles when the signal *T* is true, or a logical '1'. In the first example, notice that Interval 4 is satisfied because T is asserted at each clock cycle within the interval. As a

result, the PSL property as a whole is satisfied because it requires that at least one of the *next_a* intervals is satisfied. In the second example, however, note that all of the *next_a* intervals are violated. As a result, the overall PSL property is violated.

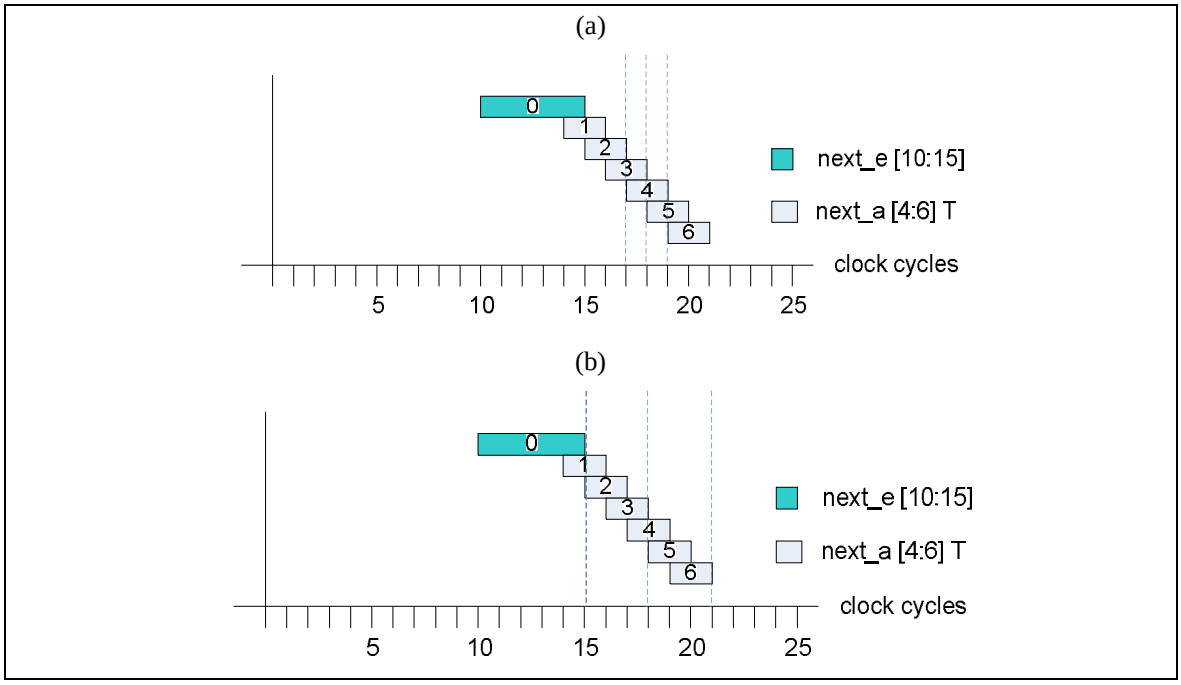


Figure 4. Satisfying and violating nested temporal PSL properties.

2.4 Logical Sibling Operators

Two or more temporal PSL operators are logical siblings if they are joined with each other via a logical *and* or *or* operator. First, consider the example in Equation 7, in which neither temporal operator has a temporal parent. As shown in the accompanying timing diagram in Figure 5, the result of Interval 0 is known at cycle 9, while the result of Interval 1 is known at cycle 12. Therefore, at cycle 12, we can apply the *and* operator to the results of the two intervals to achieve the overall result.

next_e [6:9] S AND next_a [8:12] T (7)

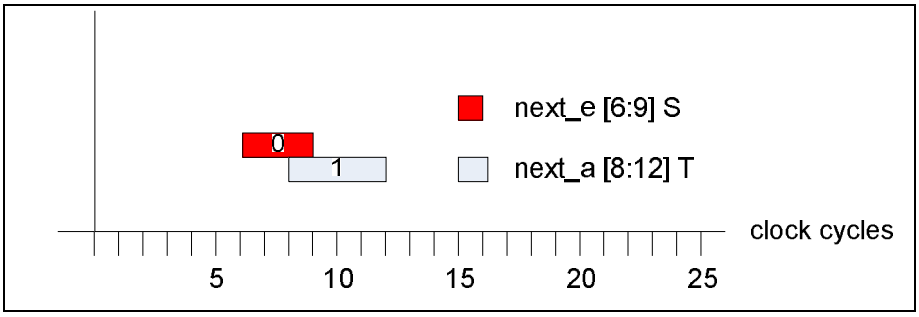


Figure 5. Timing diagram for the PSL property *next_e [6:9]S and next_a [8:12]T*.

Now consider a more complex example by adding a temporal parent, as shown in Equation 8. This is an extension of the example in the "Nested Temporal PSL Operators" section earlier, with the difference that there are now two temporal children of the PSL operator *next_e [12:17]*. As shown in Figure 6, there are six intervals (Intervals 1 – 6) that correspond to *next_a [2:4] T*, and six interval (Intervals 7 – 12) that correspond to *next_a[3:6] W*.

next_e [12:17] (next_a [2:4] T AND next_a [3:6] W) (8)

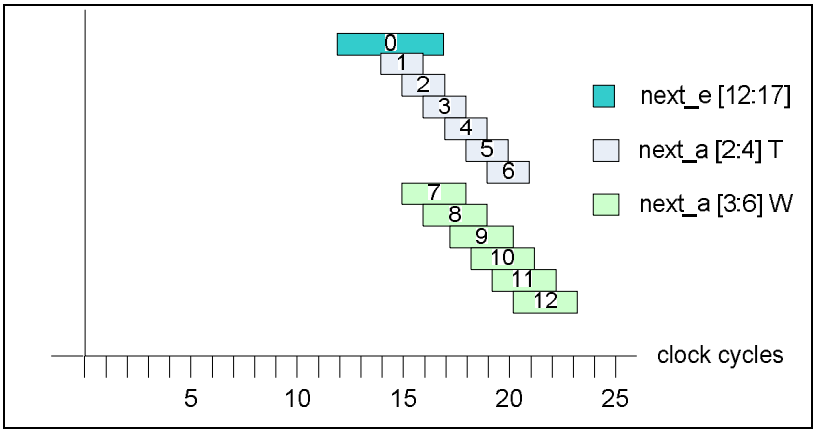


Figure 6. Timing diagram for the PSL property *next_e [12:17] (next_a [2:4] T and next_a [3:6] W)*.

Recall from the nested temporal logic example in Equation 5, that the input signal to Interval 0 at cycle 10 was the result of Interval 1. Now *next_a [3:6] W* must be incorporated into the input signal for Interval 0 at cycle 12. At clock cycle 18, the result (satisfied or violated) of Interval 7 is known. At this time, the *and* operator can be applied to the results of Intervals 1 and 7. This output becomes the input signal to Interval 0 at cycle 12. Therefore, Intervals 1 and 7 are logical siblings, as are Intervals 2 and 8, Intervals 3 and 9, and so on. This relationship is shown below in Table 4.

Table 4. Time Intervals for Nested Temporal Logic with Logical Siblings

Input signals to next_e [12:17]	Clock cycle when result is available
Result (Interval 0)	23
Result (Interval 1) AND Result (Interval 7)	18
Result (Interval 2) AND Result (Interval 8)	19
Result (Interval 3) AND Result (Interval 9)	20
Result (Interval 4) AND Result (Interval 10)	21
Result (Interval 5) AND Result (Interval 11)	22
Result (Interval 6) AND Result (Interval 12)	23

Figure 7 contains a graphical representation of the sibling relationships in the previous example. The arrows are to indicate the clock cycles at which Interval 0 may use the combined results of its child intervals. Thus, at cycle 23, the result of the PSL property *next_e [12:17] (next_a [2:4] T AND next_a [3:6] W)* is known. The solid lines connecting the intervals are to graphically display the existing logical sibling relationships. The complexity of the PSL properties can increase depending upon the number of nested temporal logic PSL operators and logical sibling operators.

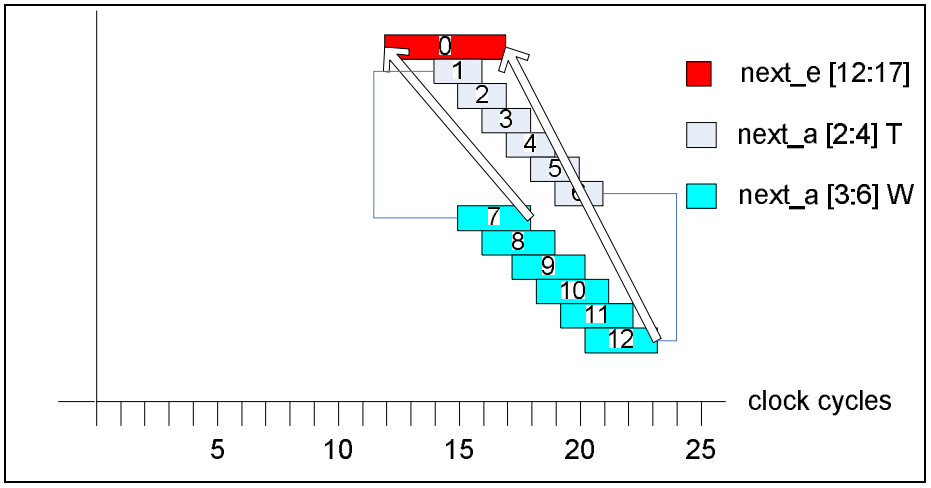


Figure 7. Example timing diagram for PSL properties with logical siblings.

3. IMPLEMENTATION

The flow for using this tool is as follows, and is shown graphically in Figure 8 on the next page. First, the user must provide PSL assertions that specify some behavior of the hardware design, and place them directly in the Verilog source code. Subsequently, the P2VSim tool will detect these embedded PSL assertions, and generate corresponding Verilog execution trace monitors to be placed back into the original Verilog code. P2VSim will then execute a script to run Modelsim non-interactively (in command line mode), resulting in the generation of an execution trace of the desired Verilog signals. Once this process has completed, the tool will statically parse the previously found embedded PSL assertions into time intervals, while storing the temporal and logical properties to be checked at runtime. Finally, the tool will allow the user to step through simulation, applying the execution trace generated from the Modelsim simulation as input against the PSL time intervals as they are instantiated. It is during this process that the user can visualize PSL assertions at exactly the clock cycle they are initialized, and also at which cycles they are satisfied or violated.

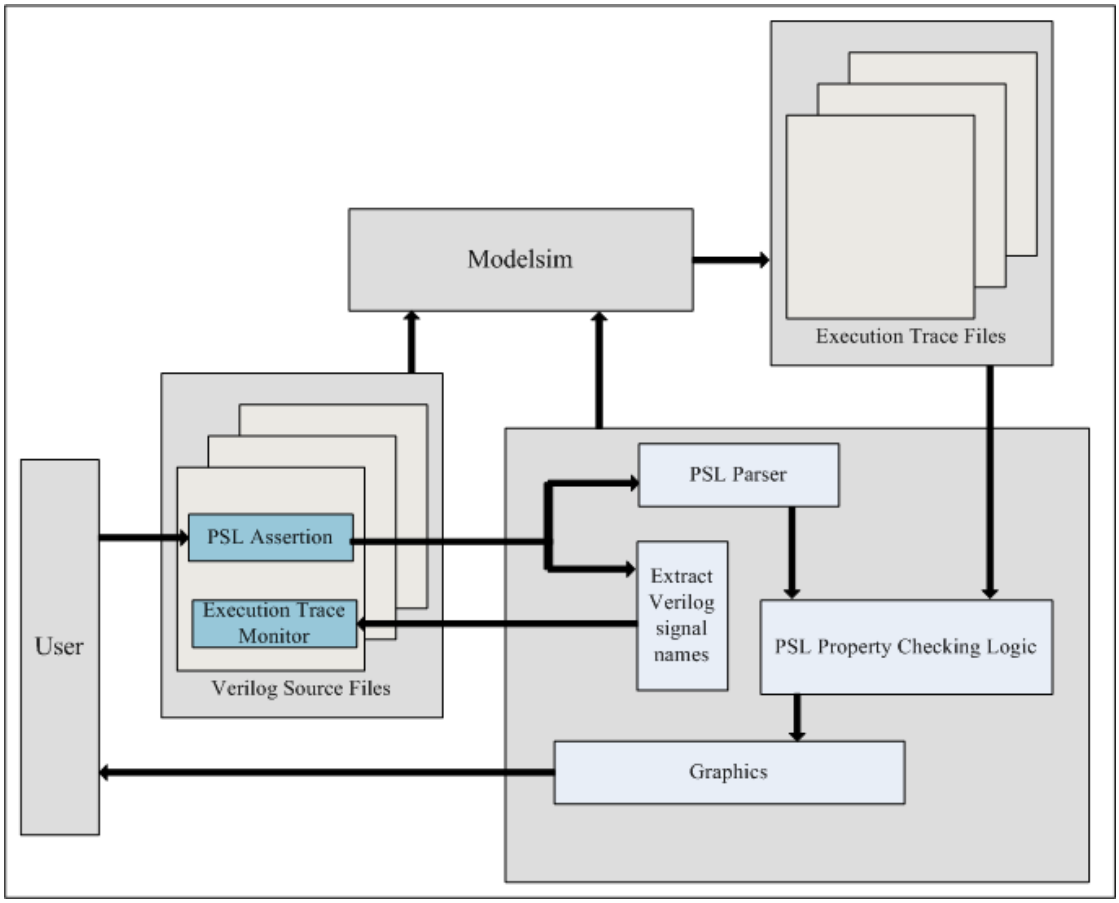


Figure 8. System block diagram.

3.1 User Specification of Embedded PSL Assertions

Throughout the developer's Verilog code, there may be signals which when logically or temporally combined, enable more efficient and effective debugging or verification. Additionally, some signals can be better evaluated over a time period rather than instantaneously. Suppose it is desirable to monitor the state of some signal *B* every time that some signal *A* is asserted for five consecutive cycles. Or perhaps when a request signal is asserted, an acknowledgement signal must be asserted within so many cycles. PSL assertions can be carefully written to define such relationships or behaviors, and then

placed inside the relevant Verilog source files. However, they must only exist in the form of Verilog comments, to ensure that they are not detected by the Verilog compiler. Figure 9 below displays the syntax to be used for the embedded PSL assertions.

```
//PSL_begin
//[Insert PSL assertion here]
//PSL_clock=[Insert clock name here]
//PSL_end
```

Figure 9. Syntax for embedded PSL assertions.

Verilog signals referenced by a given PSL assertion must be defined within the Verilog module containing the assertion. An example of a Verilog module containing an embedded PSL assertion is shown in Figure 10. The below PSL assertion specifies that if at some given clock cycle, *request* is asserted, then *ack* must be asserted at least once within the next four clock cycles. for zero-overhead non-intrusive software program monitoring [4], this work builds upon P2V in that it can be expanded to include monitoring any Verilog signals.

```
always (request → next_e [0:4] ack)                    (8)
```

```
module request_interface(clk, ack, request);
input clk;
input ack;
output request;

test u_test(clk, ack, request);

//PSL_begin
//always(request -> next_e[0:4] ack)
//PSL_clock=clk
//PSL_end

endmodule
```

Figure 10. Example Verilog module containing an embedded PSL assertion.

3.2 Obtaining a Simulation Trace of the Monitored Signals

Once the user has inserted all embedded PSL assertions, the P2VSim tool must detect them. But in order to do this, the tool must know in which directories to search for Verilog code, and also the name of the top level Verilog module. The paths specified should not only include all Verilog files containing embedded PSL assertions, but also any files to be compiled by the Modelsim compiler before running simulation. Note that the Verilog source code must be functional, and free of compiler errors, before adding embedded assertions and using this tool. Otherwise, an execution trace cannot be obtained. Figure 11 on the next page is a screenshot of the P2VSim tool allowing the user to search for and select directories containing Verilog files to be compiled in Modelsim. The paths of all directories selected, and that of the top level Verilog module, are written to a text file by the tool. This is to prevent requiring the user to enter the

paths upon each execution of the tool, although the user is certainly able to modify the current paths upon each execution of the tool if necessary.

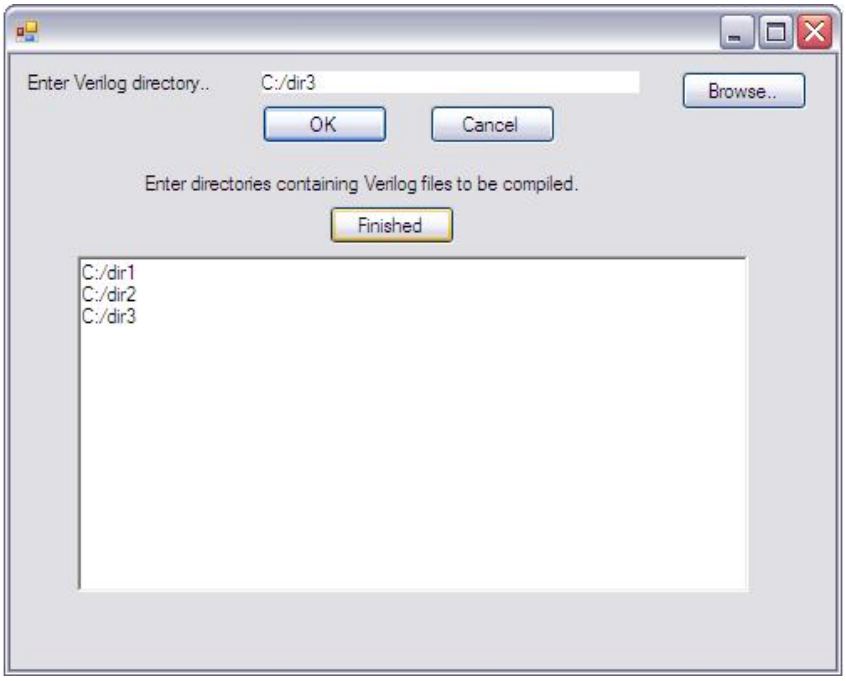


Figure 11. P2VSim allowing user to select Verilog source directories.

After all relevant paths having been set, the tool is ready to search for all embedded PSL assertions that may exist. First, the tool loads the Verilog directories from the previously mentioned text file into a C# *ArrayList* data structure. Each directory in the ArrayList is probed for Verilog files. All Verilog files are searched for the string "PSL_begin". For each instance in which this string is found, the following information is recorded: file name and path, the PSL assertion, the related clock signal, and the line number in the file in which the embedded assertion is found.

Once all embedded PSL assertions have been located, it may not be desirable for the user to examine all of the found assertions during a given simulation run. Therefore, the user is presented with the option to select any or all PSL assertions that have been found. Thus, it will not be required that PSL assertions need to be removed entirely from the Verilog files when not being examined. Below in Figure 12 is a screenshot of how the user is presented with the PSL assertions to choose from.

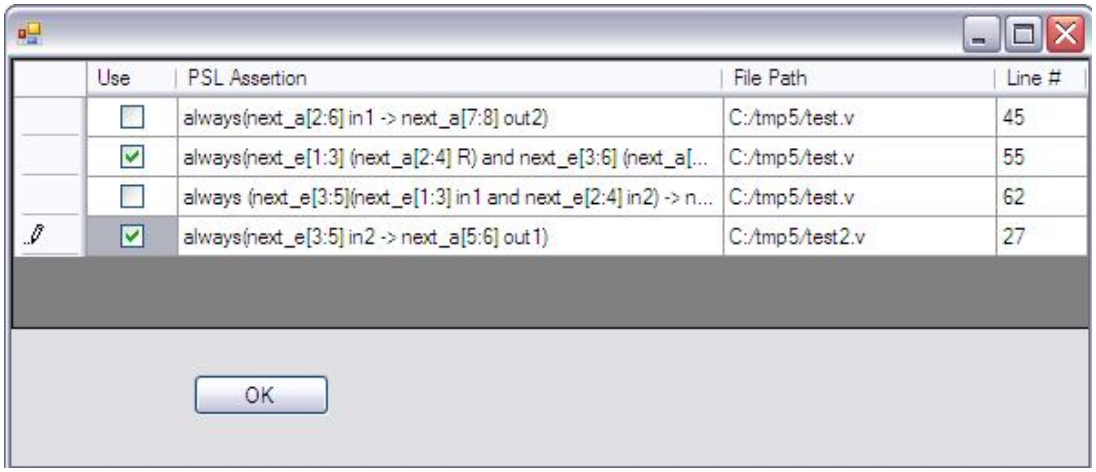


Figure 12. P2VSim allowing user to select which assertions to simulate.

At this point, the list of chosen PSL assertions has been stored within the tool. The next task is to obtain a simulation execution trace of the hardware design. This requires that there exists some testbench, implemented in either hardware or software (i.e. either a Verilog testbench or a hex file to be loaded into the hardware design), to allow meaningful events to occur during simulation of the developer's Verilog source code. The Verilog signals of interest are those included in any of the selected PSL assertions.

For example, if an embedded PSL assertion is written as shown in the “request-acknowledge” example in Figure 8, then the Verilog sigals ‘request’ and ‘ack’ require execution trace monitors to be inserted into the relevant Verilog source code. To provide for the generation of a simulation execution trace, the P2VSim tool writes Verilog code which it inserts directly into the Verilog source code. This code is programmed to dump every clock cycle number, for which the desired signals are asserted on the rising edge of the specified clock, to unique text files specific to those signals. The added Verilog code must be able to be compiled, and must be placed within the appropriate scope in the HDL code to ensure that all signals to be dumped have been previously declared. Below is an example of the inserted Verilog monitor. A unique text file is created for each monitored signal or condition. The contents of each file include the number of every clock cycle that the signal or condition is asserted. Note that while this code must be compiled, it is considered to be passive, as it monitors previously existing signals without functionally altering their behavior in any way. The inserted Verilog is generated automatically and dynamically with no additional input from the user, and is placed just below the embedded PSL assertion in the Verilog source code. An example of such Verilog code is shown in Figure 13.

```
//PSL_Trace_Begin
integer file_1_in1;
integer file_1_out2;
reg [31:0] PSL_counter;
initial begin
  PSL_counter = 0;
  file_1_in1 = $fopen("tracefile_1_in1.txt");
  file_1_out2 = $fopen("tracefile_1_out2.txt");
end
always @(posedge clk)
begin
  PSL_counter = PSL_counter + 1;
  if (in1)
    $fdisplay(file_1_in1, "%0d ", PSL_counter);
  if (out2)
    $fdisplay(file_1_out2, "%0d ", PSL_counter);
end
//PSL_Trace_End
```

Figure 13. Example execution trace monitor Verilog code.

Once the execution trace monitors have been established, the tool will invoke a non-interactive instance of Modelsim. Non-interactive in this case means that a command prompt is enabled to automatically execute the desired commands, saving the user the trouble of manually simulating his or her design. Having stored the name of the top level Verilog module, the P2VSim tool is able to dynamically generate a script to be executed that invokes Modelsim. But before this script can be executed, P2VSim first must create a working directory for the Modelsim simulation environment.

```
[modelsim executables path]/vlib work

[modelsim executables path]/vlog –work work –error –v [top module] –y [Verilog directories] +libext+.v

[modelsim executables path]/vsim –c –do run_file
```

Figure 14. Commands used to invoke Modelsim.

The command to create this directory is shown above in Figure 14. This figure also contains the Modelsim command necessary for compiling all of the necessary Verilog directories and files. The third command shown is used to initialize the command line mode in Modelsim. Again, recall that the P2VSim user need not interact in Modelsim in any way during this process, as this is all automated by the tool. Figure 15 displays the commands in the newly generated script, which contains the Modelsim commands to be executed by the newly initialized Modelsim *vsim* command prompt. The *vsim* and *run* commands are specific to Modelsim. The *vsim -c* command re-compiles the Verilog source code, including the newly added execution trace monitors. The *run -all* command simulates the Verilog RTL. During simulation, the execution trace files are created and written to as previously described.

```
vsim -c work.[name of top-level module]
run -all
exit
```

Figure 15. Commands used to simulate Verilog in Modelsim.

A naming convention exists such that a unique execution trace file is generated for each monitored signal, per PSL assertion. The name of the execution trace file takes the general form shown in Figure 16, where each selected embedded PSL assertion has been assigned a unique property number.

trace_file_<property_number>_<signal_name>.txt

Figure 16. Format of unique execution trace filenames.

3.3 Static Parsing of Embedded PSL Assertions into Time Intervals

As has been previously described, this work presents that PSL assertions can be converted into time intervals that contain temporal and logical properties, which allow PSL assertions to easily be viewed graphically. String parsing techniques are required for realizing the transition from PSL assertions to time intervals. At the first parsing layer is a transition of a PSL assertion into what is introduced as *elements*. Elements help simplify PSL assertions by removing all parentheses from a PSL assertion while allowing the ordering properties of the parentheses to be maintained. In the next layer of parsing, elements are transformed into *intervals*, which consist of several properties. These include a start time, finish time, type (*next*_e, *next_a*), temporal logic parent interval, logical sibling intervals, logical sibling operator (*and*, *or*) and a condition which leads to the eventual satisfied or violated status. Software methods are written in this tool to handle parsing when encountering the *next_e*, *next_a*, *and*, and *or* PSL operators, in addition to the *element* operator. The following subsections describe how PSL assertions

are parsed into elements, followed by detailed descriptions of the *next_e*, *next_a*, *and*, *or*, and *element handlers* used to translate elements into intervals.

3.3.1 Generating Elements from PSL Assertions

The software method *parse_into_elements()* examines an input PSL assertion string. Element 0 is the first element by default. String characters from the PSL assertion are added to element 0 one by one until an open parenthesis is encountered. When this occurs, a new element is created and the characters are copied from the PSL assertion to the new element. Further, when a new element is created due to an open parenthesis, the previous element is labeled as the new one's *element parent*. When a close parenthesis is encountered, the current element's parent element becomes the new current element, and the character copying continues. This process continues until all characters in the input PSL assertion have been copied to the elements

As there can possibly be several input PSL assertions, the first element of the second PSL assertion is the next element after the last element generated from the first PSL assertion, and so on. The parent element of any first element of any PSL assertion is set to null, to indicate that it has to element parent. Below are some examples of elements created from various input PSL assertions. One thing to note in Figure 17 is the significance of the *corresponding PSL property* column. Because each of the two examples is comprised of one PSL assertion, its property number is 1. However, to differentiate between *trigger* properties and *condition* properties, the trigger properties are assigned the value of the condition multiplied by -1.

(a)

Example 1: *always(next_e[3:5](next_e[1:3] in1 and next_e[2:4] in2) → next_a[7:8] out2)*			
Element #	Element string	Element parent	Corresponding PSL property
0	next_e[3:5] Element_1	-1	-1
1	next_e[1:3] in1 and next_e[2:4] in2	0	-1
2	next_e[7:8] out2	-1	1

(b)

Example 2: *always(next_e[1:3](next_a[2:4] R) and next_e[3:6](next_a[4:5] S) → next_a[7:8] T)*			
Element #	Element string	Element parent	Corresponding PSL property
0	next_e[1:3] Element_1 and next_e[3:6] Element_2	-1	-1
1	next_a[2:4] R	0	-1
2	next_a[4:5] S	0	-1
3	next_e[7:8] T	-1	1

Figure 17. Example for parsing PSL assertion into element strings.

3.3.2 The 'element_handler' Method

The *element handler* is executed when parsing elements into intervals and the *Element_x* operator is found. Generally, the element handler has the following purposes when checking a particular index of a given element string: Check for *next_e*, *next_a*, *or*, *and*, or another *Element_x*. If the index is not pointing to any of these operators, then the end of the element string is checked for. If not at the end of the string, then it is determined

that the string index is pointing to a *condition*, or Verilog signal(s) for which an execution trace has been generated. The element handler has the general form shown below in Figure 18.

```
Element_Handler ()
if (next_item == "next_e")  then next_e_handler()
else if (next_item == "next_a")  then next_a_handler()
else if (next_item == "and")  then and_handler()
else if (next_item == "or")  then or_handler()
else if (next_item == "element")  then element_handler()
else if (end_of_string)  then element = parent_element[element]; index = string_index[element]
else  //must be a condition (i.e. input signal)
```

Figure 18. Pseudocode for the element handler.

3.3.3 The 'next_e_handler' and 'next_a_handler' Methods

The *next_e handler* is executed when parsing elements into intervals and the *next_e* operator is found. Unlike the element handler, the next_e handler has the ability to create new intervals. Although each next_e PSL property has a start time and finish time, it is possible in some cases that these times do not indicate the actual time the interval should be examined. If an interval has a temporal logic parent interval, then its local start and finish times must be added relative to their parent interval's start and finish times. Such behavior was previously shown in Table 3. Another duty of the next_e handler is to detect logical sibling intervals and correctly associate them with each other using the *and* or *or* operator for PSL property checking later. There are four possible scenarios for a given interval, based on whether or not it has a temporal parent interval and whether or

not it has a logical sibling interval. Following are descriptions of two scenarios: when the interval does not have a temporal parent interval or a logical sibling interval, and then when the interval has both a temporal parent interval and a logical sibling interval.

First, consider the case where a given interval does not have either a temporal parent interval or a logical sibling interval. Initially, the local start and finish times of the interval are obtained, and whether or not the interval has a temporal parent interval is checked. Because this interval does not have a temporal parent interval, much of the complex code in this handler method is ignored. The next step is to instantiate the interval. The local start and finish times are applied directly to the interval's start and finish times. The *type* of interval is declared to be *next_e*. The interval parent field is listed as null, and the interval property is the unique property number assigned to the input embedded PSL assertion. The condition field remains unset at this point. Because this interval does not have a logical sibling interval, the logical sibling field is set to null, as is the logical sibling operator. This interval number is, however, added to a *potential_sibling* field in case it can possibly become a sibling to an interval created later. It is important to remember that while two intervals *A* and *B* are in fact logical sibling operators, if *A* is created before *B*, then *A* stores a null value as its logical sibling interval but *B* stores *A* as its logical sibling operator. Next, the string "next_e" is stored in the *most_recent_op* field for the current element string, and the interval number is stored in the *most_recent_element* field for the current element string. At this point, it is checked whether an *element* is the next substring following the next_e substring. If so, then the *element_handler* is executed. Otherwise, the substring following the next_e substring must be a condition. Once the

condition is added to the *condition* field for the interval, the interval is complete. It is possible that either this is the end of the string, or that the condition has been followed by an *and* or *or* operator. If the latter is true, then the corresponding handler method is invoked.

Next, consider the case where an interval has both a temporal parent and a logical sibling. As described in the previous paragraph, the local start and finish times of the interval are obtained, and whether or not the interval has a temporal parent interval is checked. Because this interval does have a temporal parent interval, there are more scenarios that must be checked in order to ensure the correct start and finish times, along with the correct logical sibling relationships with other intervals. In each iteration of a *for-loop* from this interval's parent's start time to finish time, one new instantiation of this interval is created with a start time of the sum of the for-loop index and this interval's local start time, and a finish time of the sum of the for-loop index and this interval's local finish time. Also, for each iteration of the for-loop, before entering the next iteration, it must be checked whether or not this interval has a logical sibling interval. In the first iteration(s) of a next_e interval that has a temporal parent interval, it is not possible that the intervals generated inside the for-loop will have logical sibling intervals. However, if these intervals are logically joined to other intervals via the *and* or *or* operator, then these other (later set of) intervals will be said to have logical sibling operators pointing to this first set of intervals. For this second set of intervals, the code executed is more complex than with respect to the first set of intervals. While the first set of intervals does not have

logical sibling intervals, they must be stored as *potential logical sibling intervals* such that the second set of intervals is able to link to them as logical siblings.

For the above-mentioned second set of intervals to be instantiated, whose *logical sibling* fields will point to the first set of intervals, the operations are as follows. At the time of checking whether a temporal logic operator exists, the first two assignments of logical sibling intervals are paid special attention. On the first pass, or instantiation of the first interval in the for-loop, the interval number of the first logical sibling is stored. Similarly, on the second pass, the interval number of the second logical sibling is stored. The purpose of storing these interval numbers in dedicated local variables is that at this point the interval number distance (difference in interval numbers) is known, and can be extrapolated to all future logical sibling intervals in this series. The below example better describes what occurs during the first pass, second pass, and thereafter. In this example, the PSL property being parsed is:

next_e[3:5] (next_e[1:3] in1 and next_e[2:4] in2)	(9)

The first interval encountered when parsing the above string is *next_e[3:5]*. This interval has no temporal parent interval and no logical sibling operator, and is instantiated as previously described for such a scenario. At this point, the following interval, shown in Figure 19(a), has been instantiated.

(a)

Interval #	String	Has parent?	Has sibling?	Start time	Finish time	Condition
0	next_e[3:5]	No	No	3	5	NULL

(b)

Interval #	String	Has parent?	Has sibling?	Start time	Finish time	Condition
0	next_e[3:5]	No	No	3	5	NULL
1	next_e[1:3]	Interval 0	No	4	6	in1
2	next_e[1:3]	Interval 0	No	5	7	in1
3	next_e[1:3]	Interval 0	No	6	8	in1

Figure 19. Interval parsing example displaying interval attributes – Part I.

Next under consideration is *next_e[1:3] in1*. From the PSL property string, interval 0 is the temporal parent interval of the intervals to be generated with respect to *next_e[1:3] in1*. Because these intervals make up the first set of child intervals of interval 0, these intervals do not have logical siblings. The resulting intervals instantiated from this first set of children intervals is as shown in Figure 19(b).

Note that while intervals 1 – 3 do not have logical sibling intervals, they have been appropriately stored as *potential logical sibling intervals* for possible future use by other intervals. Additionally, the *interval_unit_width*, or difference between the interval numbers between intervals 2 – 4 has been stored. In this case, the interval_unit_width is

1. In this example, intervals 4 – 6 will use this information to establish a logical sibling link. Revisit Figure 17(a), which displays the element string breakdown for this example. The next substring in element 1 after instantiating interval 3 is “and”. Therefore, after executing the *and_handler*, the most recent PSL operator for element 1 will be listed as “and”, and the potential logic sibling for element 1 will be listed as interval 1. Upon entering the next_e handler for the second child set of intervals, labeled *next_e[2:4]*, after interval 4 is created, the most recent PSL operator being “and” will notify the interval that it has a logical sibling. Interval 1, the logical sibling of interval 4, is currently stored in the potential logic sibling variable. This process represents the “first pass” of finding the logical sibling intervals of intervals 4 – 6. Next of importance of this point is the storage of interval 4’s logical sibling in a two-item FIFO queue. The tail of the queue, where logical sibling numbers are pushed, holds the current sibling number just added in the most recent interval instantiation. The data at the head of the queue holds the interval sibling number stored in the previous interval instantiation before the current. At this point (after the first pass), the FIFO has the status shown in Figure 20(a).

(a)

FIFO index	Logical sibling interval number stored
Head of FIFO	Uninitialized
Tail of FIFO	Interval 1

Figure 20. FIFO for maintaining interval number distance between logical siblings.

(b)

FIFO index	Logical sibling interval number stored
Head of FIFO	Interval 1
Tail of FIFO	Interval 2

Figure 20 continued.

Following is the instantiation of interval 5. Again, it is detected that interval 5 does have a logical sibling interval. In this second pass, it is determined that the logical sibling interval of interval 5 is interval 2. This calculation has two inputs: the logical sibling of interval 4, and the *interval_unit_width* stored previously with respect to intervals 1 – 3. Because the logical sibling of interval 4 is interval 1, and the interval_unit_width of intervals 1 – 3 is 1, it is calculated that the logical sibling interval of interval 5 is interval 2. Interval 2 is then pushed onto the FIFO described in Figure 20(b).

It can be extrapolated that the logical sibling of interval 6 is interval 3. The FIFO is no longer updated, as only the difference between intervals 1 and 2 is needed. Once interval 6 is instantiated and associated with its logical sibling interval, the parsing for this example has completed. Shown in Figure 21 is the final description of intervals 0 – 6 for this example.

Interval #	String	Has parent?	Has sibling?	Start time	Finish time	Condition
0	next_e[3:5]	No	No	3	5	NULL
1	next_e[1:3]	Interval 0	No	4	6	in1
2	next_e[1:3]	Interval 0	No	5	7	in1
3	next_e[1:3]	Interval 0	No	6	8	in1
4	next_e[2:4]	Interval 0	Yes	5	7	in2
5	next_e[2:4]	Interval 0	Yes	6	8	in2
6	next_e[2:4]	Interval 0	Yes	7	9	in2

Figure 21. Interval parsing example displaying interval attributes – Part II.

The next_a_handler method holds logic nearly identical to the next_e_handler. The only difference is in the *interval type* field when instantiating an interval. Either *next_e* or *next_a* must be chosen. Correctly tagging an interval with its interval type is necessary as this affects the PSL property checking to be described in a later section.

3.3.4 The 'and_handler' and 'or_handler' Methods

Interval instantiation does not occur in the *and_handler* and *or_handler* methods. These methods are present for two main reasons: to advance the string index in an element string by the appropriate amount, and more importantly, to set the most recent PSL operator of an element to *and* or *or*. When the most recent PSL operator of an element is set to one of these values, it implies that some logical sibling relationship exists, waiting

to be realized. Below in Figure 22 is the C# code for the and_handler, similar to that of the or_handler.

```
   private void and_handler()  {
      //set this element's 'most_recent_op' to AND
      most_recent_op[elem] = (string)"and";
      //advance string index past "and"
      ind += 3;
      ind = eatwhitespace(elements[elem].ToString(), ind);
}
```

Figure 22. C# code for and handler.

3.4 PSL Property Checking and Simulating PSL Assertions

Once the input PSL assertions have been parsed into intervals, the P2VSim tool is ready to simulate the PSL assertions. Recall that intervals collectively represent a PSL assertion, and also that each interval requires one of two sources of input. This can either come from the satisfied or violated state of a child interval or from a Verilog signal or group of signals. By previously running Verilog simulation using Modelsim, an execution trace of all signals referenced by any of the intervals has already been created. Thus, all required inputs are available. The software method which simulates the PSL assertions, one cycle at a time, is called *one_step_sim*. By clicking the one_step_sim button, the simulation shown in the graph will advance by one cycle, updating the satisfied or violated property of any active intervals currently shown on the graph.

3.4.1 The Possible Interval States

There are seven unique possible states an interval that can be in during simulation, as shown below in Table 5. An interval is in state 0 if the current clock cycle number is less than the interval's start time. Once the current clock cycle is the interval's start time, then the interval becomes active. At any time that an interval is active, it can become satisfied or violated, depending on the input conditions. Once an interval is satisfied or violated, it remains in that state and is not considered for further property checking. States 4 and 5 relate to intervals that are temporal parents. Before such an interval's start time is reached, it is simply in state 0. Once the start time has been reached, if the state of the interval's first child has not been determined, then this parent interval is in state 4. Once the first input from once of its children is received, the interval is in state 5. From state 5, this interval will eventually become satisfied or violated. The latest cycle that this will occur is when the last of its children have a status of satisfied or violated, although the result can potentially arrive sooner. Lastly, state 6 is solely for trigger intervals. There must be a difference between a satisfied trigger interval and a satisfied non-trigger interval. This is because upon each cycle that a trigger interval is satisfied, a condition interval must be instantiated. To avoid the incorrect instantiation of condition intervals, the status of a satisfied trigger interval is set to 6, and its status is never again checked.

Table 5. The Possible States of the Intervals During PSL Property Checking

0	Before start: Interval start time not yet reached.
1	Active: Interval start time has been reached, but a final status has not been determined.
2	Satisfied: Interval status is satisfied.
3	Violated: Interval status is violated.
4	Waiting parent: Start time has been reached, but waiting on first child status as input.
5	Active parent: Start time has been reached, but waiting on remaining children's status as inputs.
6	Satisfied trigger interval that has been satisfied. State 2 must not apply.

3.4.2 The Property Checking of Intervals

Upon each click of the *one_step_sim* button by the user, the state of each instantiated interval must be updated. This process occurs one interval at a time. The first check is whether or not an interval is a temporal parent. If it is not a temporal parent, it is checked whether the interval is a trigger or a condition interval. This is done so that the correct execution trace file can be opened for review of the relevant inputs. Once the execution trace file is open, the file is searched for the current clock cycle number. If the number is found, then it means that the interval's condition is asserted. If not, the interval's condition is deasserted. The remaining code directly updates the interval's state appropriately.

The first check is whether or not the current clock cycle is this interval's start time. If so, then the interval enters either state 1 or 4, depending on whether or not it is a

temporal parent waiting for its children's states. If the interval is not a temporal parent, then the following steps shown in Table 6 are taken in updating the interval's state. The property checking must be in the same order depicted in the table. First, if the input condition is false and the interval type is next_a, then the interval is violated. If the input condition is satisfied and the condition is next_e, then the interval is satisfied. If the interval type is next_a and it is the last cycle of the interval, then the interval is satisfied. A value of *don't care* is listed as the input condition because it is known that the input condition is true if the interval type is next_a. Otherwise, the interval state would have been changed to violated already during the first entry of the table below. Similarly, if it is the last cycle of the interval and the interval type is next_e, then the interval is violated. Otherwise, the interval would have been satisfied in the second entry in the below table. If none of the first four checks apply to the current interval, then the interval remains in the same state.

Table 6. The PSL Property Checking Steps Taken When not a Temporal Parent Interval

Current State	Input Condition	Interval Type	Last Cycle of Interval?	Updated State
1: Active	False	next_a	Don't Care	3: Violated
1: Active	True	next_e	Don't Care	2: Satisfied
1: Active	Don't Care	next_a	Yes	2: Satisfied
1: Active	Don't Care	Don't Care	Yes	3: Violated
1: Active	Don't Care	Don't Care	No	1: Active

Now consider the property checking if the interval is in fact a temporal parent interval. The time that a parent must wait until knowing the result of its first child is previously calculated and is known by the parent. If the timing the parent must continue to wait for its child is nonzero, then the wait time is simply decremented by one. If the wait time counter is zero, then the parent is updated to state 5 as an active parent. When an interval is in state 5, there are several checks to be made. Before proceeding, the status and interval number of the child that the parent is waiting for must be stored locally. If the child has a logical sibling interval, then its result cannot be directly sent to its parent. First, the child's result must be logically combined with all of its siblings using the appropriate *and* and *or* operations. Once a combined result has been obtained, the result becomes available for the parent interval. If the child had no logical siblings, then its results can be sent directly to its parent. At this point, the property checking is similar to that shown in Table 6 above. After each cycle, the interval will either have a state of active parent, satisfied, or violated.

3.4.3 The Instantiation of Trigger Intervals

After static parsing is performed on the input PSL assertion strings, the properties of each generated interval such as start time, finish time, and type are stored. However, when simulation begins for PSL property checking, there are initially no intervals instantiated. The intervals generated from the static parsing are stored as *relative time intervals*, as it is unknown, for example when a *condition interval* will be activated. Stated differently, the start times of condition intervals are unknown initially. They are only instantiated

once its corresponding *trigger interval* is satisfied. Consider the PSL property *next_e [0:4] S*. The parsing mechanism would calculate a start time of 0 and a finish time of 4. However, suppose that this property is a condition property dependent of some trigger *T*. At the time *T* is satisfied, the condition property will be instantiated with a start time of the current clock cycle and a finish time of the current clock cycle plus four.

While it is more obvious that the interval start and finish times of condition intervals are unknown initially, this also holds true for trigger intervals. Referring to Equation 1 displaying the syntax of the input PSL assertions, it can be assumed that every trigger interval is instantiated at each clock cycle, to respect the *always* PSL operator. Obviously this introduces potential for extensive memory usage during program execution, but a watchdog mechanism should be implemented in the P2VSim tool to free satisfied or violated intervals after some time period. This functionality currently does not exist in the P2VSim tool.

3.4.4 The Instantiation of Condition Intervals

After the PSL property checking on all intervals has completed for a given clock cycle, each trigger interval must be checked for whether or not it is in the satisfied state. If a trigger interval is in the satisfied state, then the corresponding condition interval must be instantiated and added to the graph. Trigger intervals are identified by their PSL property number. As mentioned before, each embedded PSL assertion selected by the user has a unique property number. Further, each PSL property is divided into a trigger and condition interval. Because of this, when a PSL assertion is assigned a property number

of 1, then its trigger intervals have a property number of -1, and its condition intervals have a property number of 1. Similarly, if a PSL assertion is assigned the number 2, then its trigger intervals' property numbers are -2, while its condition intervals' property numbers are 2. Thus, when searching all instantiated intervals, those with a negative property number must be checked for whether or not they are in the satisfied state. After all trigger intervals have been checked, the one_step_sim method continues. The final task at each clock cycle is to generate a new additional set of trigger intervals. This is because a trigger is checked at every clock cycle and most triggers will span across multiple cycles. Therefore, the only way to ensure that all asserted triggers are successfully realized, an additional trigger must be instantiated for each cycle.

4. USING THE P2VSIM TOOL

This section is intended to demonstrate how the P2VSim tool is used by examining two use cases. The first example relates to real time monitoring. Real time monitors are useful for guaranteeing with cycle accuracy that specified deadlines are met. This example is a request-acknowledge bus handshake, in which every time a bus request is made, an acknowledgement must be received within a certain number of cycles. The second use case is an example of pattern detection. Sometimes when debugging RTL simulations, it would be useful to have the ability to monitor multiple signals collectively over a period of time. Consider debugging a hardware system that uses JTAG. The single-bit *TMS* signal dictates the state of the JTAG state machine. That is, a unique pattern can be seen on the TMS signal over multiple clock cycles indicating that a certain JTAG operation has commenced.

4.1 Real Time Monitoring Example

This example exhibits how to use the P2VSim tool to monitor the real time specification of a hardware system. The time being monitored is the number of cycles between a bus request and acknowledgement. Suppose that in the developer's Verilog source code, the request signal is *request* and the acknowledgement signal is *ack*. Further, assume that every time *req* is asserted, *ack* must be asserted within five clock cycles, inclusive of the cycle when *request* is asserted. The relevant PSL assertion is shown below.

always (request → next_e[0:4] ack)	(10)

As noted previously, the *request* interval will be instantiated upon every clock cycle, and the *ack* interval will be generated only when *request* is asserted. The *request* intervals will not be visible graphically, due to the number of trigger intervals that will be generated over time. Only the instantiated *ack* intervals will be visible graphically, with a start time of the current cycle and a finish time of the current cycle plus four. Assume for this example that *request* is asserted at clock cycles 2, 9, and 16. The corresponding *next_e* intervals shown on the graph will start at these same respective cycle times. Also assume for this example that *ack* is asserted at clock cycles 5, 13, and 21. Thus, it is expected that the first two instantiations will be satisfied and the third will be violated. Shown below in Figure 23 is a sequence of the significant clock cycles while simulating the PSL assertions using the P2VSim tool.

(a)

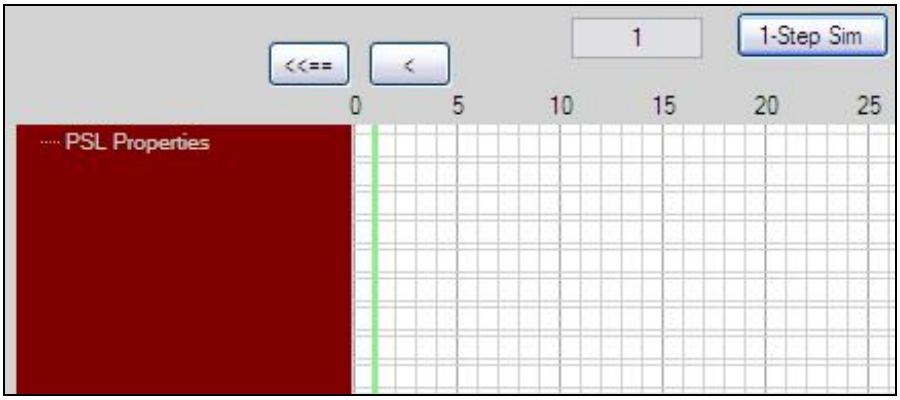
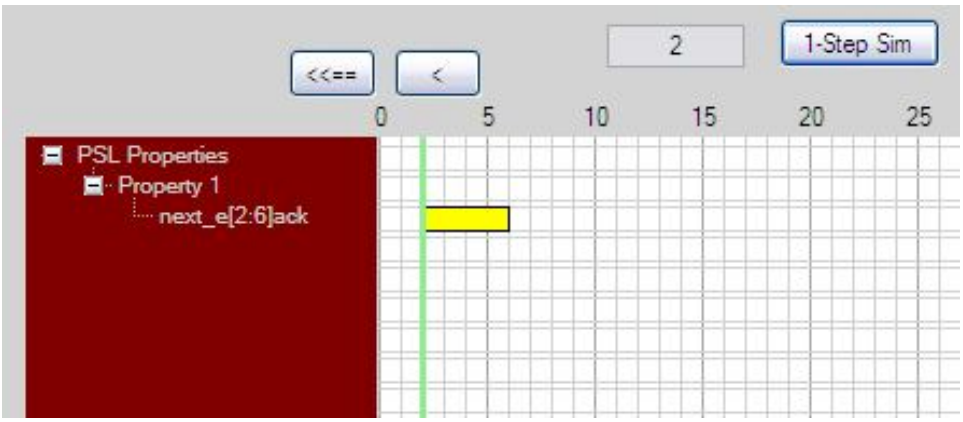
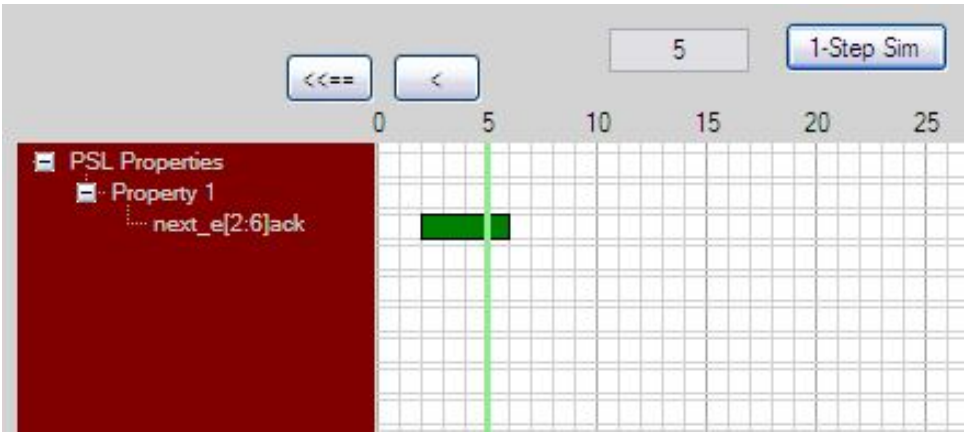


Figure 23. Real time monitor example screenshots.

(b)



(c)



(d)

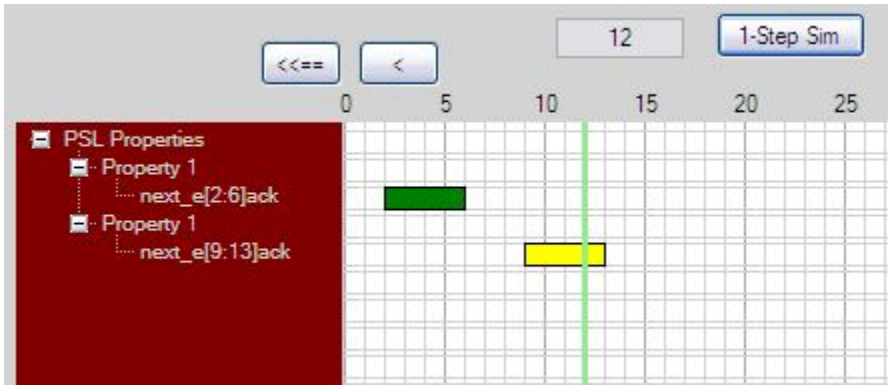
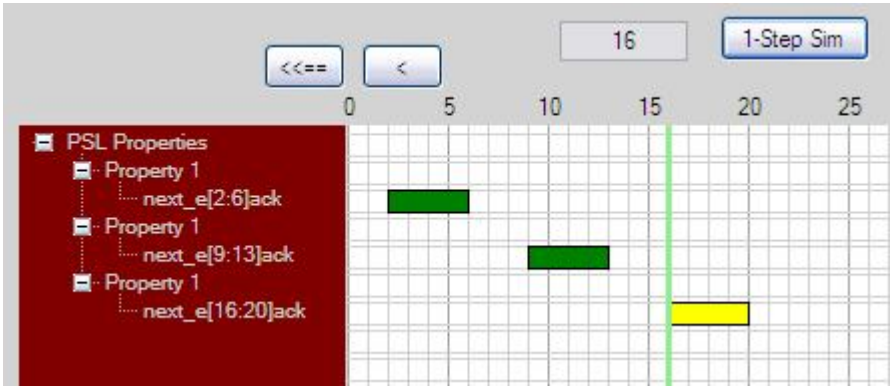


Figure 23 continued.

(e)



(f)

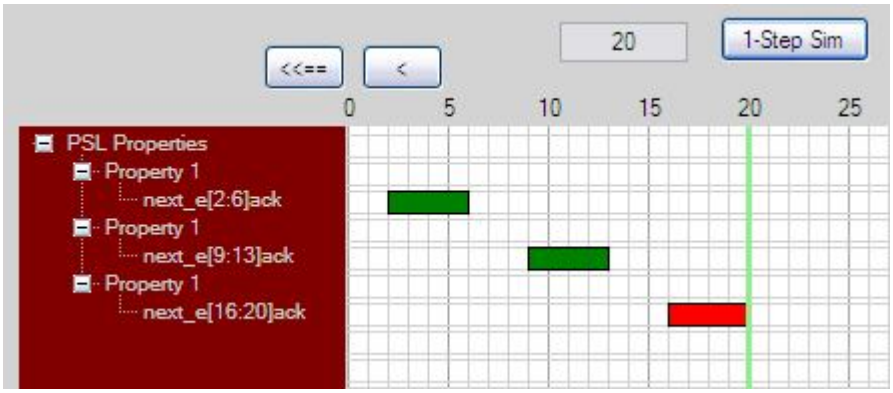


Figure 23 continued.

Figure 23(a) shows the state of the graph in cycle 1, before any trigger intervals have been satisfied. Correspondingly, no condition intervals have been instantiated and placed on the graph. In Figure 23(b), the state of the graph is displayed at cycle 2. As noted before, the trigger signal *request* is asserted at cycle 2, resulting in the instantiation of the corresponding interval *next_e[2:6] ack*. This interval is shaded yellow to indicate that the interval is active but a final satisfied or violated decision has not yet been made.

Figure 23(c) reflects the state of the graph at cycle 5. Recall that it is at this cycle that *ack* is asserted, resulting in the satisfied state of this interval. Note that a green shading indicates a satisfied interval. Figure 23(d) below jumps to cycle 12. Note that *request* was asserted at cycle 9, resulting in the corresponding next_e interval. However, at this cycle, the interval has not yet become satisfied. The result of *ack* at cycle 13 will determine the result of this interval.

In Figure 23(e), it can be seen that at cycle 13, *ack* was in fact asserted and the second next_e interval was satisfied. This figure shows the state of the graph at cycle 16, when *request* is asserted and the corresponding condition interval is instantiated. Figure 23(f) shows that at cycle 20, the *request* signal still has not been asserted and the third next_e interval is violated. Overall from this example we can see that in the first two cases an acknowledgement to the request was seen within 4 and 5 cycles, respectively, resulting in satisfaction of the real time specifications. For the third interval, it can be seen that because no acknowledgement comes by cycle 20 for the request in cycle 16, the hardware design has violated the real time requirements.

4.2 Pattern Detection Example

This example demonstrates how the P2VSim tool can be used to detect patterns among collective Verilog signals across multiple clock cycles. For this scenario, let signal *A* be the trigger, while signals *B* and *C* are checked collectively for the pattern *BBBCC*, where *B* is asserted for three consecutive cycles and *C* is asserted in the following two cycles.

Further, this pattern is to occur within ten cycles of signal *A* being asserted. The relevant PSL assertion is shown below, followed by the equivalent logic for this assertion.

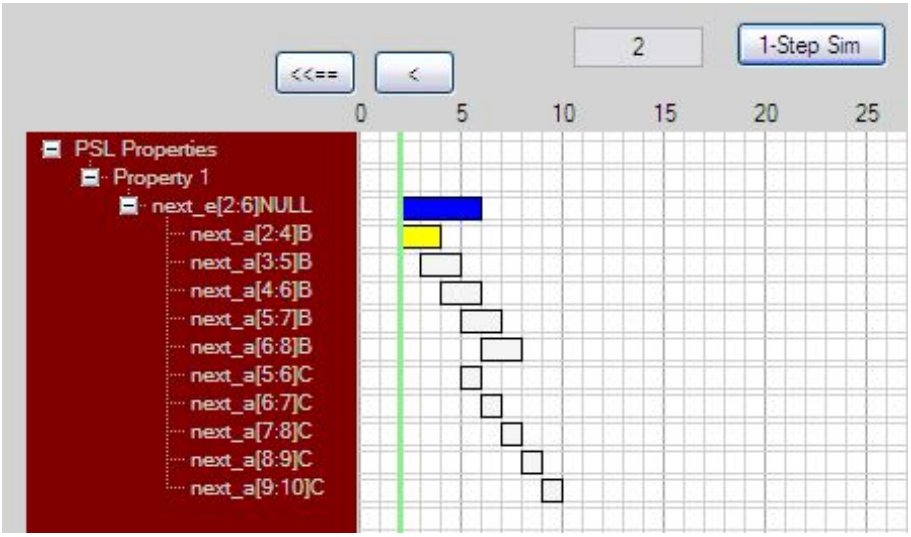
always (A → next_e[0:4] (next_a[0:2] B and next_a[3:4] C)) (11)

(next_a[0:2] B AND next_a[3:4] C) OR…OR (next_a[4:6] B AND next_a[7:8] C) (12)

Assume for this example that the trigger signal *A* is asserted at cycle 2. Also assume that signal *B* will be asserted in cycles 2 – 4 and 6 – 8. Lastly, assume that signal *C* will be asserted in cycles 5 – 6 and 9 – 10. Shown below is a sequence of the significant clock cycles while simulating the PSL assertions using the P2VSim tool. The purpose of this example is to demonstrate the PSL property checking logic when a temporal logic parent interval is present and logical siblings exist as well.

Figure 24(a) shows the state of the graph at cycle 2. It is during this cycle that signal *A* is asserted, resulting in the instantiation of the corresponding condition intervals. The *next_e* interval is shaded blue, indicating that this is a temporal logic parent interval waiting for the results of its child intervals. It can also be concluded from the graph at cycle 2 that the signal *B* is asserted, because its corresponding next_a interval has not yet been violated.

(a)



(b)

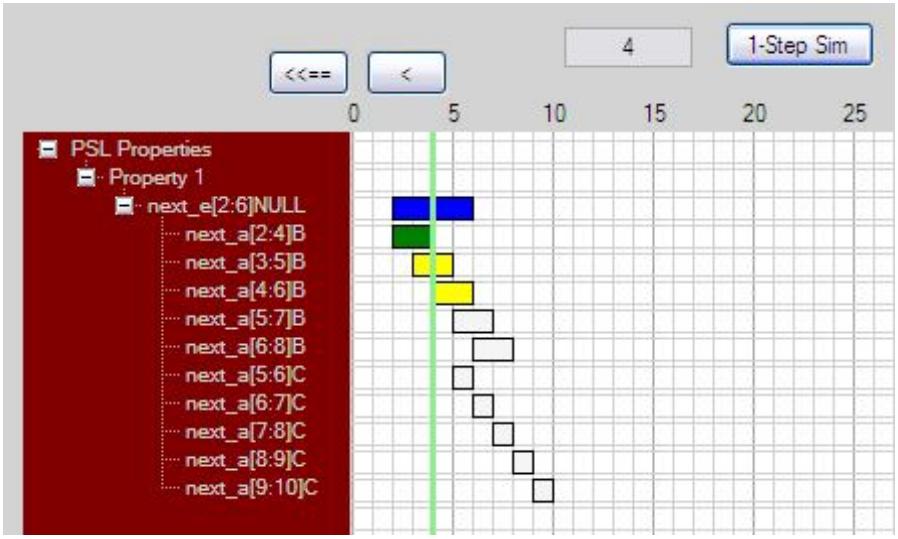
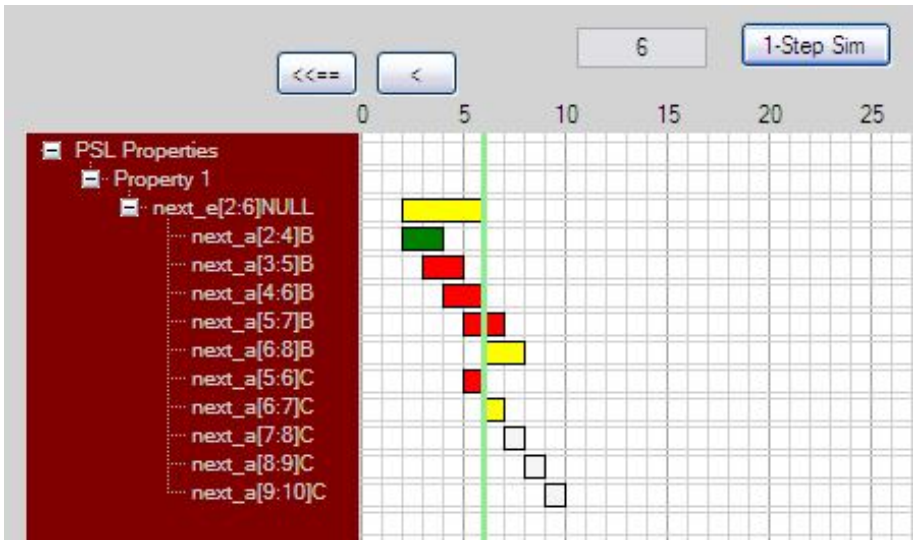


Figure 24. Pattern detection example screenshots.

(c)



(d)

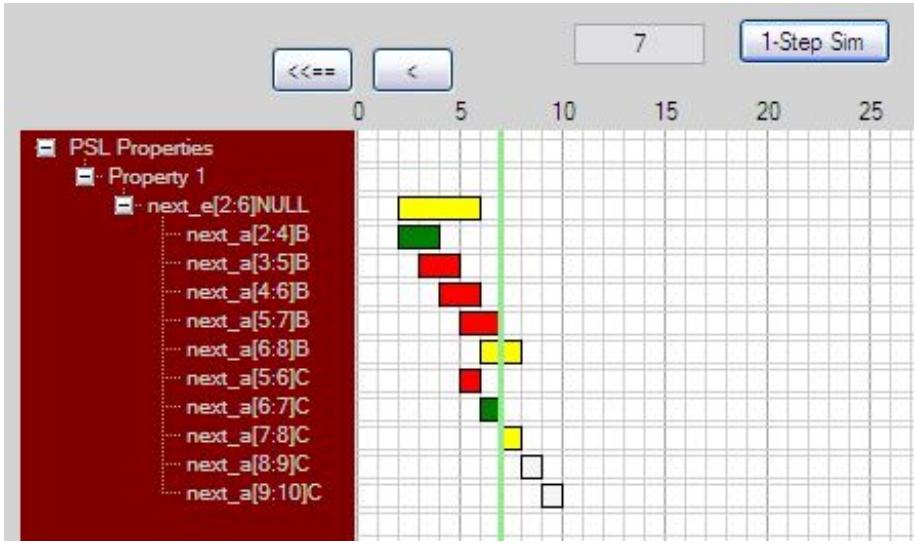


Figure 24 continued.

(e)

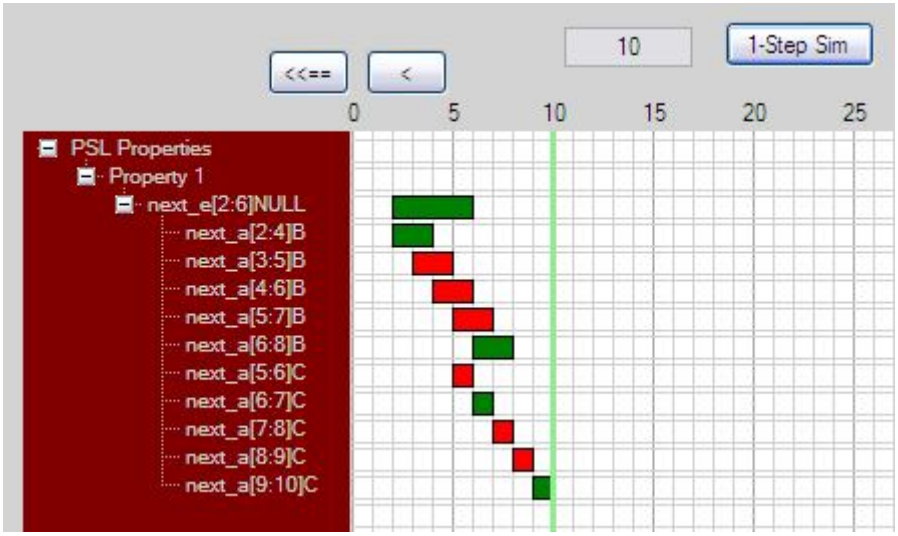


Figure 24 continued.

Figure 24(b) shows the state of the graph at cycle 4. It is at this cycle that the first *B* next_a interval has been satisfied. This is because *B* was asserted at cycles 2, 3, and 4. The two following intervals are active but not yet determined, as *B* would have to continue to be asserted in order to satisfy those next_a intervals. Also note that the next_e interval is still shaded blue. This interval will not have its first input until the result of the first *B* next_a interval is ANDed with the result of the first *C* next_a interval. This will not take place until cycle 6.

Figure 24(c) above displays the state of the graph at cycle 6. Here it is seen that the first *C* next_a interval is violated. This result is ANDed with the satisfied result of the first *B* next_a interval. Because this intermediate result is violated, the input to the next_e parent interval corresponding to cycle 2 is deasserted or violated. However, because it is a next_e interval, it is only required that a satisfied or asserted input is returned at least

once within the following four cycles. Figure 24(d) shows similar behavior at cycle 7. The second *C* next_a interval is satisfied, but its logical sibling interval, the second *B* next_a interval has already been declared as violated. Thus, another violated result is returned to their parent interval.

Figure 24(e) shows the final results of this overall PSL assertion. It can be seen that the final *B* next_a interval was satisfied. Also at cycle 10, the final *C* next_a interval is satisfied. The logical AND of these two intervals is satisfied, and this result is passed back to their parent interval. As a result, the next_e interval becomes satisfied, which is considered to be the overall result of the entire PSL assertion. Recall that signal *A* was asserted at cycle 2. From cycles 6 – 8, signal *B* was asserted and from cycles 9 – 10, signal *C* was asserted. This confirms that the pattern *BBBCC* was seen within the required number of clock cycles after the signal *A* was initially asserted.

CONCLUSION

The Property Specification Language allows hardware developers to specify clock cycle accurate temporal and logical behavioral properties of hardware designs. When using the P2VSim tool, such PSL assertions are embedded directly within the Verilog source code to monitor the signals of interest. Ultimately, the user is allowed the opportunity to combine Verilog signals not only logically, but across multiple clock cycles to perform RTL verification or debugging.

While PSL is already widely used, the representation of PSL as graphical time intervals is presented with the P2VSim tool. Here it is argued that viewing multiple Verilog signals as one temporal group can be easier and more useful than individually tracking the signals individually. Further, by parsing the input PSL assertions into elements and then into intervals, more is visible to the developer than a simple satisfied or violated result. The use of intervals provides finer granularity in examining which signals and at which clock cycles cause a PSL assertion to be satisfied or violated.

As described, the flow of the P2VSim tool begins with hardware property specification. Once the assertions have been placed inside the Verilog source code, the tool locates all assertions and allows the user to select which PSL assertions to continue with on a particular property checking simulation run. Verilog monitors are dynamically created and placed back into the Verilog source code to generate a simulation execution trace. Once the monitors are in place, P2VSim invokes Modelsim to simulate the Verilog source code and generate execution trace files for the relevant signals. The tool then parses the PSL assertions into intervals, and performs PSL property checking on the

intervals using the execution trace as input. Finally, the user is able to see the step by step simulation and property checking of the PSL assertions.

The examples provided demonstrate how the P2VSim tool can be used for monitoring real time requirements of a hardware system and also for pattern detection. In the real time monitor example, three cases are analyzed: an acknowledgement to the corresponding request is seen prior to the deadline, at the deadline, and after the deadline. The first two cases result in satisfied assertions, while the third is violated. The pattern detection example is provided to demonstrate the versatility of the P2VSim tool. Here, property checking for temporal logic parent intervals is shown, along with property checking of logical sibling intervals. Because PSL allows a user to specify hardware behavior very precisely with respect to cycle accuracy and to combine several signals into one useful assertion, the pattern detection example can be reduced or expanded upon to suit the particular needs of a verification or debugging task.

REFERENCES

[1] Accellera, *IEEE P1850 PSL.* Accellera Organization, Napa, CA.

[2] Weste, N. and Harris, D., 2005, *CMOS VLSI Design: A Circuits and Systems Perspective* (3rd Edition). Addison-Wesley.

[3] Hsu, Y., Tabbara, B., Chen, Y., and Tsai, F., 2003, Advanced Techniques for RTL Debugging. In *Proceedings of the Design Automation Conference*, pp. 362-367.

[4] Lu, H. and Forin, A., 2007, P2V: An Architecture for Zero-Overhead Online Verification of Software Programs. In *Workshop on Application Specific Processors.* pp. 1-8.

[5] Pittman, R., Lynch, N., and Forin, A., 2006, eMIPS, A Dynamically Extensible Processor. *Technical Report MSRTR- 2006-143*, Microsoft Research, Redmond, WA.

VITA

Name: Oscar Michael Almeida

Address: Department of Computer Science and Engineering

Texas A&M University

TAMU 3112

College Station, TX 77843-3112

Email Address: oscar10@tamu.edu

Education: B.S., Computer Engineering, Texas A&M University, 2006

M.S., Computer Engineering, Texas A&M University, 2009