A Simulation-Based Temporal Assertion Checker for PSL

Kai-Hui Chang, Wei-Ting Tu, Yi-Jong Yeh, and Sy-Yen Kuo

Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
sykuo@cc.ee.ntu.edu.tw

Abstract

A simulation-based temporal assertion verification engine for PSL (Property Specification Language), called Tempral Wizard, is proposed in this paper. It is very efficient because its time and space complexity are both O(n). A new concept, tag, is introduced in Tempral Wizard and it handles the forall operator elegantly.

1. Introduction

Many circuit designs exhibit temporal behaviors. In the past, there was no good solution to express rules for these temporal behaviors and it was difficult to verify them. As the complexity of circuits increases, it becomes more and more important to find a way to express and verify these temporal rules [1]. There are several vendors providing various solutions, such as OpenVera [2] from Synopsis and E from Verisity [3]. Recently, the IBM Sugar has been adopted as a formal specification language called PSL(Property Specification Language) [4]. PSL can be used for both formal and simulation-based verification tools, and a simple subset is defined for simulation-based verification. In this paper, a simulation-based verification engine, called Temporal Wizard, is proposed for the simple subset. It provides a set of Verilog user-defined system tasks/functions (USTF) and is compatible with all Verilog simulators which support PLI. A new concept, tag, is introduced in Temporal Wizard and it handles the forall operator elegantly. Compared with some state-machine based temporal logic checkers [5][6], Temporal Wizard enhanced the power of temporal assertion checkers significantly. Its time and space complexity is O(n) and is very efficient.

2. Thread, Tag and Sequence

2.1 Terminology

Assertion: A statement that a given property is required to hold and a directive to verification tools to verify that it does hold.

Check: The execution of the auxiliary process that monitors simulation of a design and reports success or failure when asserted properties do or do not hold.

Event: An event can be produced by a variable or a Sequence. For variable, if clock is specified, an event is said to occur when the value of the variable is sampled one at the clock edge. If no clock is specified, then an event is said to occur when the value of the variable becomes one. For Sequence, an event occurs when the thread corresponding to the Sequence terminates, which may either succeed or fail.

Sequence: A textual representation of temporal assertions. Each Sequence corresponds to a Verilog USTF. It is the basic element in Temporal Wizard. All temporal assertions are written in Sequence USTF in Temporal Wizard.

Tag: Data that is associated to a variable and carried by a thread.

Trigger: The start of a Sequence checking.

2.2 Thread

Thread is the data structure created dynamically during simulation for a particular Sequence. When an event occurs, the threads waiting for the event will be evaluated, and corresponding actions will be taken. When the first event of a Sequence occurs, a new thread is generated for further checking of this temporal logic assertion. This process is called "spawn a thread."

Once a Sequence checker is triggered, there may be several streams of events being checked at the same time. For example, if events "a b c d e" are expected to occur in sequence, and the events occur in the order "a b c d a b", then there will be two possible event streams that satisfy the rule, each has its own state. One has events "a b c d" checked, and the other has events "a b" checked. If "e" occurs, the first Sequence stream will finish successfully, but the partially checked "a b" will be left intact. Later if "c d e" occur, the Sequence stream will generate a successful event when it finishes. See Figure 1 for detailed description about the example.

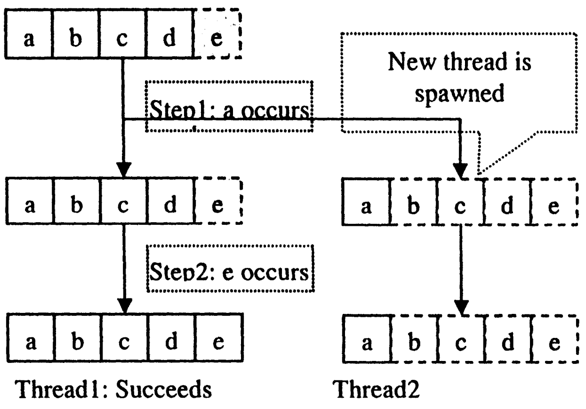

Figure 1. Example Event Stream.

2.3 Tag

Since there may be several threads spawned from the same Sequence, it will be very useful if some data can be carried by the thread and be reused later. For example, if we want to express the following temporal rule, we have to pick some value and carry it with the temporal flow for later comparison or use:

Event2 should occur after event1, and variable V should have the same value when either event occurs.

In this case, we should pick the value of V when event1 occurs and compare it with variable V when event2 occurs.

This is why Sequence tag is necessary. A tag is a data handler in Sequence that can be used to attach a data with a Sequence thread. It is always associated with a variable. It can be used to save data to a Sequence or load data from a Sequence. It can also be used to qualify an event.

2.4 Use of Tag for Forall

One important usage of tag is to reduce the time and space complexity that the PSL forall operator brings.

In ordinary checkers which do not have tags, the forall operator is expanded. For example, in the following assertion:

Forall i in 1..3: request && (data_in == i) -> next (data_out == i)

Ordinary checkers will expand it to three rules for i= 1..3. But with tag, only one rule is necessary. The value of *data_in* will be saved in the tag when *request* occurs, and it will be compared with *data_out* at the next clock cycle.

2.5 Temporal Wizard Usage

A complete Sequence definition has three blocks: Assertion definition, Sequence trigger, and result handling. Assertions are written in the Verilog tasks provided by Temporal Wizard. Sequence trigger is done by the *$tb_seq_trigger* task. It has two arguments: The first one is the Sequence handle returned in assertion block, and the second one is the result variable. After a Sequence is triggered, the check starts immediately. If the assertion fails, bit 1 of the result variable toggles. If the assertion succeeds, bit 0 toggles. Result handling should be based on the result variable. A typical usage of Sequence is given in Figure 2. In the example, $tb_seq_range is used for the assertion that event2 should occur between 1 to 2 clocks, and $tb_seq means event1 and the assertion represented by handle1 should occur in sequence. Hence the assertion says that event2 should occur within 1 to 2 clocks after event1 occurs.

```
// Sequence Definition Block
handle1= $tb_seq_range(clock, $tb_range(1, 2),
event2);
handle2= $tb_seq(clock, event1,handle2);
// Sequence Trigger Block
$tb_seq_trigger(handle2, result_variable);
// Result Handling Block
always @(result_variable[1])
  $display("Assertion failure");
always @(result_variable[0])
  $display("Assertion Success");
```

Figure 2. Typical Sequence Usage.

2.6 PSL to Sequence Converter

A PSL to Sequence converter is provided as the bridge between PSL and Temporal Wizard. With the converter, PSL assertions can be converted to a Verilog module that can be included in the design directly. The inputs of the module are the signals we are monitoring. Clock and reset ports are also included to control the behavior of the verification module.

An example of the verification module is given below:

PSL:
*always ({signal_a} |-> {{signal_b;signal_b}[*3..5];signal_c}) @(posedge clock);*

Verfication module:
module Verification_unit(clk, rst, signal_a, signal_b, signal_c);
Sequence definition
...
endmodule

Design module:
`include Verification_unit
module Design(...);
...
Verification_unit Assertions(clk, rst, signal_a, signal_b, signal_c);
endmodule

The flow to generate the verification module is given in Figure 3.

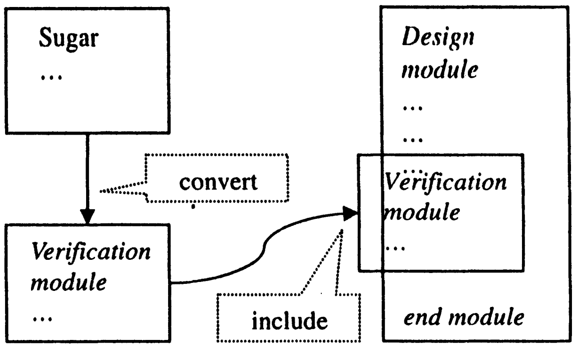

Figure 3. Flow to Generate Verification Module.

3. Temporal Wizard Implementation

3.1 Data Structures

Temporal Wizard has two main data structures: Sequence and Thread. An example of Sequence structure is given in Figure 4. The Sequence it represents is:
h1= $tb_seq_range($tb_posedge(clk), $tb_range(2, 4), e4);
h2= $tb_seq($tb_posedge(clk), e1, e2, h1, e3);

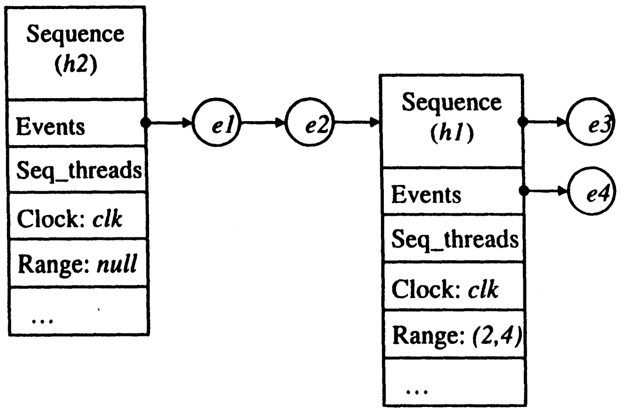

Figure 4. Example Sequence.

3.2 Algorithms

When the first event of a Sequence occurs, a thread for it will be created. That thread will monitor some events. When those events occur, the thread will be evaluated, and Sequence-specific actions will be taken.

There are several types of Sequences defined in Temporal Wizard, including serial, temporal constraint, order constraint, flow control, logic arithmetic, thread control, and tag control. In general, a thread has four stages in its life cycle: Setup, evaluate, execute and finalize. In setup, the thread is created. In evaluate, the thread waits for events to evaluate itself. In execute, the program counter is advanced to the next event. In finalize, some clean-up is done and the thread is destroyed.

A Sequence thread has two states: Active and inactive. In active state, the thread is waiting to be evaluated. In inactive state, the thread is not waiting for any event and is waiting for its child thread to resume itself. The life cycle of thread is given in Figure 5. In the figure, PC means Program Counter, which is the execution state of the thread.

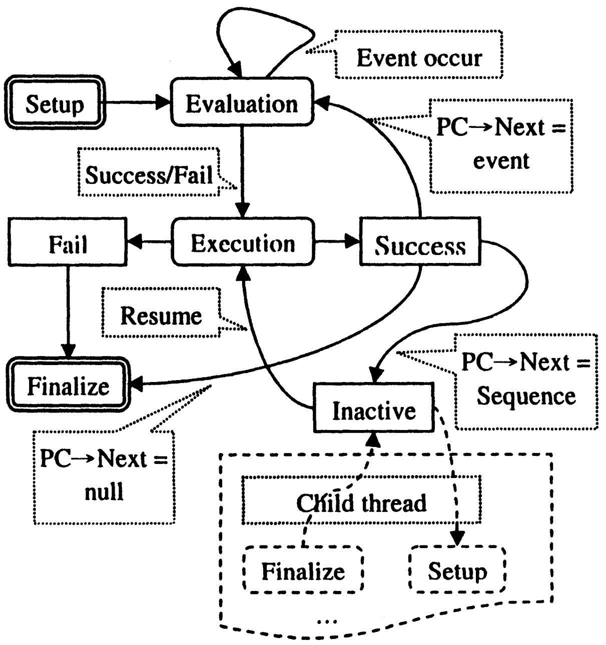

Figure 5. Life Cycle of Thread.

Detailed algorithm for serial, temporal constraint, and order constraint are illustrated in the following subsections. Note that a Sequence can also be an event that a thread monitors.

3.2.1 Serial Sequence

The USTF provided for serial constraint sequence is:
seq_handle = $tb_seq(clock, list of events)

It specifies a list of events that should occur in sequence. It contains a linked list of events. For example, if event "A B C" should occur in a series, then it contains a linked list of these events.

The thread for it contains a Program Counter (PC) which points to the event it is waiting for and monitors that event. When that event occurs, the callback mechanism will evaluate the thread, and the PC will point to the next event. If it is the last event in the Sequence, then the thread terminates successfully. See Figure 6 for example checking status of the sequence "A B C D E". In the example, there are two partially checked threads. One is waiting for event B, and the other one is waiting for event E.

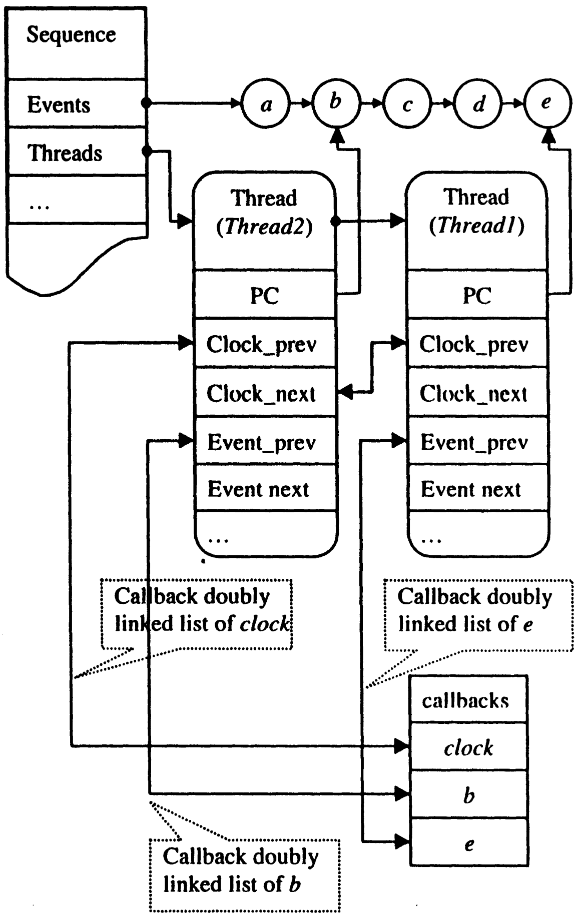

Figure 6. Serial Sequence Execution.

3.2.2 Temporal Constraint Sequence

The USTF provided for temporal constraint sequence is:

seq_handle = $tb_seq_range(clock, range, event)

It specifies the constraint that an event should occur within in a clock range. For example, event "A" should occur between 3 and 5 clocks from now. The Sequence contains the event and the clock to monitor. The range of the clock is also saved in it.

The thread for it contains a counter for the number of clocks. It monitors the event and the clock. When the clock event occurs, the counter increases. If the event occurs within the clock range, the thread terminates successfully. Otherwise it fails.

3.2.3 Order Constraint Sequence

The USTF provided for order constraint sequence is:
seq_handle = $tb_seq_before(event1, event2)

It specifies that an event should occur before the other one. For example, that event "A" should occur before event "B." Two events and the clock to sample them are saved in this Sequence.

The thread for the Sequence monitors the two events. If the first event occurs before the second one, then the thread terminates successfully; otherwise it fails.

3.3 Time and Space Complexity Analysis

For a temporal assertion, if there are *n* events, then at most *n* threads will be created. When an event occur, there will be at most *n* threads to be evaluated, each evaluation takes constant time. So the time and space complexity for an assertion are both O(*n*).

3.4 Benchmarks and Comparisons

Besides Sequence, Cadence NC-Verilog also offers ABV (Assertion-Based Verification) [7] that supports PSL language. It can simulate design with PSL assertions internally or externally. For assertions with a small number of events, the running time remains small and is similar between Temporal Wizard and NC-ABV. Therefore the repeat operator is used to create an assertion with a large number of events.

The benchmark is done on Redhat 7.2. The version of NC-Verilog is 5.00-b006, and the CPU is Pentium 4 1.6G. In the benchmark, the Sequence code is produced by the PSL-to-Sequence converter.

PSL:
*always({signal_b}|->{signal_a[*REPEAT_TIMES]})*
@(posedge clk);
Note: REPEAT_TIMES is the number of times that signal_a repeats.

Sequence:
h1 = $tb_seq_range_start($tb_posedge(clk), 1,signal_a);
h2 = $tb_seq_repeat(REPEAT_TIMES - 1,h1);
h3 = $tb_seq_now(signal_a);
h4 = $tb_seq($tb_posedge(clk), h3, h4);
h5 = $tb_seq($tb_posedge(clk), signal_b);
h6 = $tb_seq_imply0(h5, h4);

Stimulus:
Signal_a is always 1.
Signal_b period is 80ns, initilized to 0.
Clock period is 10ns, initialized to 0.

	Sequence	NC-ABV
REPEAT_TIMES	Running Time (sec)	Running Time (sec)
1	0.280	0.270
10	0.610	0.480
100	3.260	5.730
1000	35.780	399.580
10000	276.610	34479.180

From the experimental results, it can be observed that the running time of Temporal Wizard is O(*n*), while NC-ABC seems to be O(n^2). Compared with NC ABV, Sequence is more powerful. First, Sequence reports assertion success as well as failure, while NC only reports failure. Second, the performance of Temporal Wizard is better than NC ABV.

4. Conclusion

A temporal assertion checker for the simple set of PSL, called Temporal Wizard, is proposed in this paper. Its algorithm and data structures are also described. A new concept, tag, is introduced in this checker to handle the forall operator more elegantly. With tag, the time and space complexity of Temporal Wizard remains O(*n*) and is vert efficient. The approach to convert PSL to Verilog USTF makes Temporal Wizard compatible with all Verilog simulators which support PLI and it makes Temporal Wizard easy to use.

5. References

[1] Don MacMillen, Michael Butts, Raul Composano, Dwight Hill, and Thomas W. Williams, An Industrial View of Electronic Design Automation, IEEE Tranc. on Computer-Aided Design of Integrated Circuits and Systems, Volume: 19 Issue: 12 , Dec 2000

[2] Synopsis, OpenVera 1.0 Language Reference Manual Version 1.0, Mar. 2001

[3] Verisity, e Language Reference Manual Version 3.2.1, 1999

[4] Accellera, Property Specification Language Reference Manual, Version 1.0, Jan. 2003

[5] Koji Ara and Kei Suzuki, A proposal for transaction-level verification with component wrapper language, Design, Automation and Test in Europe Conference and Exhibition, 2003

[6] Ajay J. Daga and William P. Birmingham, A symbolic-simulation approach to the timing verification of interacting FSMs, Computer Design: VLSI in Computers and Processors, Proceedings on 1995 IEEE International Conference, 1995

[7] Cadence, Writing and Using Assertions in Cadence's Dynamic Assertion-Based Verification for NC-Verilog Version 4.1, May. 2002