A Tag-Augmented Temporal Logic Checker

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

Many circuit designs need to follow some temporal logic assertions. However, it was hard to express and verify them in the past. Therefore a temporal logic checker, called Sequence, is proposed in this paper. It provides some Verilog system tasks for the users to write assertions in their designs directly. A PSL-to-Sequence converter is also provided so that assertions written in the standard PSL (Property Specification Language) can be verified by the Sequence checker. In this paper, two new concepts, tag and thread, are introduced to help users attach data to their temporal assertions and provide more functionalities than previous temporal logic checkers. The importance of tags and threads to temporal logic checker is like enhancing a regular expression parser to context-free. A benchmark comparison with a commercial product is also given.

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 logic rules. There are several vendors providing various solutions, such as OpenVera from Synopsis and E from Verisity. Recently, the IBM Sugar has been adopted as a formal specification language called PSL [1]. However, no connection with the underlying simulator is proposed.

In this paper, a new way to express temporal logic rules in Verilog HDL is proposed and is called “Sequence.” Sequence can be used as a bridge between PSL and simulators with a PSL-to-Sequence converter, but it can also be used directly to write temporal assertions in Verilog HDL. Two new concepts in temporal logic, thread and tag, are also introduced in Sequence. Compared with some state-machine based temporal logic checkers [2][3], thread and tag enhance the power of temporal logic checkers significantly. Their importance is like enhancing a parser from regular expression to context-free [4]. This paper describes the syntax of Sequence and how it is implemented. A comparison between Sequence and a commercial PSL assertion checker is also given.

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.

Clock: It is used in synchronous Sequence checking to sample signals. It can be “$tb_posedge(clock)” for the positive edge of the clock, “$tb_negedge(clock)” for the negative edge of the clock, or “clock” for both edges. If “0” is used, then it means asynchronous Sequence checking. For synchronous checking, an event variable is always sampled at the clock edge. For asynchronous checking, if the event variable is 1 when the Sequence is triggered, or becomes 1 after the Sequence is triggered, then it is considered a successful event.

Event: An event has no time duration, and is either successful or failed. There are two ways that an event can be generated: From a Verilog variable or from a Sequence. For a Verilog variable, an event occurs if its value is 1, becomes 1, or is sampled 1 at the specified clock edge. An event from Verilog variable is always a successful event. For a Sequence, an event is generated after that Sequence is finished. If no rule specified in that Sequence is violated, a successful event will be generated. If any rule is violated, a failed event will be generated.

Handle: Every Sequence task returns a handle. The handle is always 32-bits in width and can be used as an argument in other Sequence tasks. In this way, complicated temporal logic assertions can be described.

Sequence: An assertion written in Verilog system tasks purposed in this paper. A Sequence is a series of events with timing, order and/or tag constraints. A Sequence will generate a failed or successful event when it finishes, and can be used as an event in other Sequences.

Spawn a thread: 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.”

Thread: A partially checked Sequence. It has its own status and represents a temporal logic stream.

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

Trigger: The start of a Sequence checking.

2.2 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.

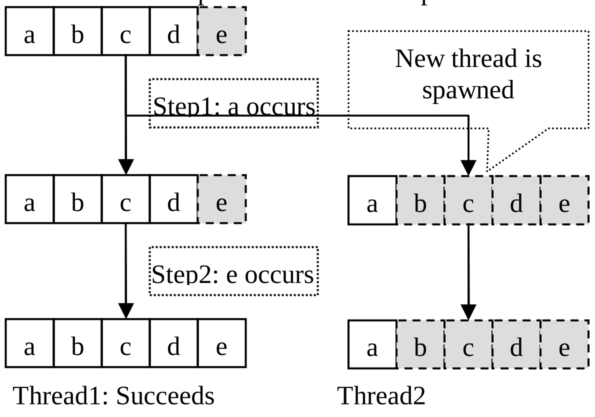

Figure 1. Example Event Stream.

Since there may be multiple streams of the same assertion flowing concurrently, there should be a way to represent these streams so that they can be handled. In Sequence, these streams are represented by threads. Every thread has its unique ID, and the thread can be manipulated by it. This concept is similar to the threads in an operating system.

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.

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 Analysis of Thread and Tag

With thread and tag, the power of temporal logic checker is enhanced significantly. Take an example from the language theory [5] first. In a regular expression parser, the following sentences are allowed, where superscript denotes number of times the alphabet should repeat:

$$A^1B^2, A^*B, AB^*$$

However, the following sentence is not allowed:

$$A^nB^n$$

It is because a regular expression parser does not have the ability to "remember" how many times an alphabet has repeated. To parse A^nB^n, a context-free parser must be used.

The same situation exists in the temporal logic checker. If no information is allowed to be carried with the logic flow, then the following assertion can be checked, where superscript denotes number of times the event should occur:

$$A^1B^2, A^*B, AB^*$$

However, the following assertion cannot be checked:

$$A^nB^n$$

It is because if "n" is not saved with the temporal stream, the number of times that B should repeat will not be known. With tag, such an assertion is possible.

Another usage of tag is to qualify an event with the value of another variable. For example, we may want to express the following assertion: (Subscript means the value of variable V. For example, A_1 means only if V is 1 when A occurs should this event be considered legal.)

$$A_nB_n$$

It means that when A occurs, if V's value is "n", then only when B occurs and V's value is also n should this assertion be valid.

Such an assertion is common in real designs. For example, assume there is a bus bridge with two sides. If one side, called bus A, writes something to an address on the other side, called bus B. Then only the transaction on bus B that matches the address of the transaction on bus A is the correct transaction we are looking for. For example, if on bus A data is written to address 0x10, then the write transaction to address 0x10 on bus B is correct, while the transaction to address 0x20 is not. It is one of the A_nB_n examples. Split-completion on a PCI-X or PCI-Express bus is another example. See Figure 2 for a more detailed description about how A_nB_n is checked..

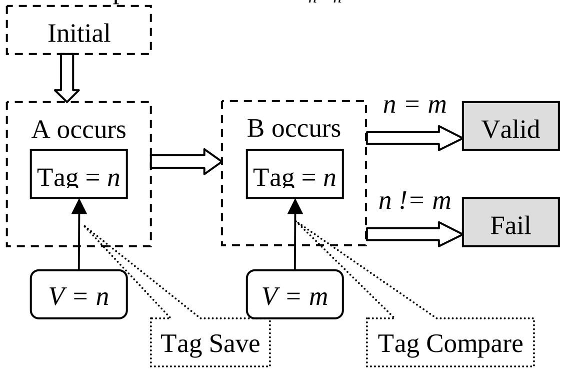

Figure 2. A_nB_n Example.

Thread and tag also provide the bridge for Sequence to interact with auxiliary Verilog code. With these augmented code, some complicated assertions that cannot be expressed by Sequence alone can be written and checked.

2.5 Sequence Usage

A complete Sequence definition has three blocks: Assertion definition, Sequence trigger, and result handling. Assertions are written in the Verilog tasks provided by Sequence. 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 3.

```
// Sequence Definition Block
handle1= $tb_seq_range(clock, $tb_range(1, 2), event1);
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 3. Sequence Usage.

CTL (branching-temporal logic) [6], defined as OBE (Optional Branching Extension) in PSL, are supported in Sequence by reporting both failure and success of the assertion. The “all path” operator is supported by detecting failures of the assertion. If any thread fails in the assertion, then the all path operator fails. The “some path” operator is supported by detecting successes of the assertion. If any thread succeeds in the assertion, then the some path operator holds.

3. Sequence Implementation

3.1 Data Structures

Sequence has two main data structures: Sequence and Thread. Sequence represents the assertion that the user writes, and thread is created dynamically during run-time. The data saved in a tag is also carried by thread. An example of Sequence structure is given in Figure 4.

```
h1= $tb_seq_range($tb_posedge(clk), $tb_range(2, 4), e4);
h2= $tb_seq($tb_posedge(clk), e1, e2, h1, e3);
```

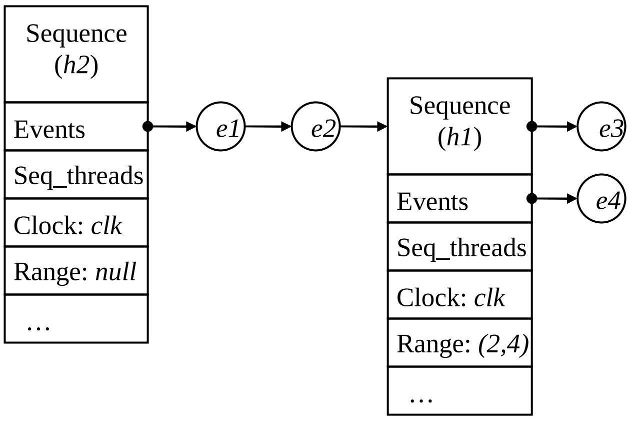

Figure 4. Example Sequence.

3.2 Algorithms

A Sequence 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.

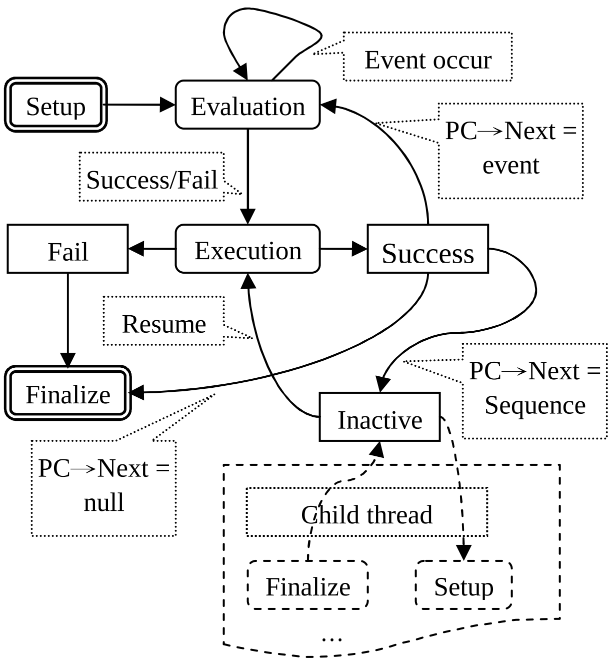

Figure 5. Thread Execution Algorithm.

4. PSL to Sequence Converter

4.1 Converter Usage

A PSL to Sequence converter is provided as the bridge between PSL and Sequence. 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 and the flow to generate it is given in Figure 6.

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

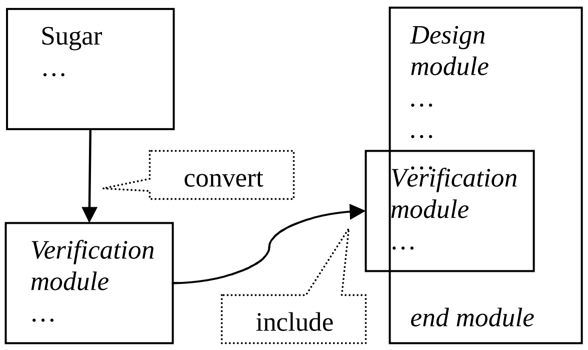

Figure 6. PSL-to-Sequence Converter.

4.2 Benchmarks and Comparisons

Besides Sequence, Cadence NC-Verilog also offers ABV (Assertion-Based Verification) that supports PSL language [7]. It can simulate design with PSL assertions internally or externally. 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 Sequence is better than NC ABV. A benchmark is given below. The Sequence code is produced by the converter.

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.

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

4.3 Discussion

Repeat is chosen as the benchmarks because it is the most time consuming property. In the benchmark, new threads are spawned at every positive clock edge. It is the reason the benchmark takes so long to simulate. From the benchmark, it can be seen that the time complexity of Sequence remains linear, while NC is exponential. Even with large numbers of threads, the performance of Sequence is still extraordinary.

5. Conclusion

A temporal logic checker, called Sequence, is proposed in this paper. Its syntax, usage and implementation are also described. Compared with commercial products, its performance is outstanding. Two innovative concepts, tag and thread, are also introduced in this paper. These two features enrich the power of temporal logic checkers and enable the user to write more complex temporal assertions than before. Its importance is like enhancing a regular expression parser to context-free.

6. References

[1] Accellera, Property Specification Language Reference Manual, Version 1.0, Jan. 2003
[2] 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
[3] 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
[4] Charles N. Fisher and Richard J. LeBlanc, Jr., Crafting a Compiler with C, The Benjamin/Cummings Publishing Company, Inc., 1991
[5] Noam Chomsky, The Logical Structure pf Linguistic Theory, The University of Chicago Press, 1985
[6] L. Lamport, Sometimes is sometimes “not never” – on the Temporal Logic of Programs, Proc. 7th ACM Symposium on Principles of Programming Languages, Jan. 1980
[7] Cadence, Writing and Using Assertions in Cadence’s Dynamic Assertion-Based Verification for NC-Verilog Version 4.1, May. 2002