Detecting Harmful Race Conditions in SystemC Models Using Formal Techniques

Sven Beyer, Dominik Strasser
OneSpin Solutions GmbH
Nymphenburger Str. 20a
80335 Muenchen, Germany

Abstract- Due to the nature of some Hardware Description Languages (HDLs), it is possible to inadvertently insert race conditions into design code. The SystemC class library standard suffers from this issue, by allowing race conditions to be coded that may lead to complex, hard-to-detect bug conditions. Automated applications, based on Formal methods, may now be used to verify many hardware design scenarios more effectively and easily than simulation-based methods. We will present a new, unique technique that leverages formal methods to exhaustively test a block of SystemC code that uses any number of threads for potential race issues. This mechanism may be applied to Transaction Level Model (TLM) code as well as RTL code in SystemC, and could be leveraged by any Formal Verification tool that allows Property Checking to be applied to SystemC based designs.

I. Introduction

Race conditions in HDL-based design have caused significant problems in integrated circuit development, particularly when using the Verilog HDL. Verilog is notorious for allowing a designer to create code that contains sections open to alternative interpretations by different tools, and that also behave differently than expected in a production device. Race conditions are a significant subset of these coding anomalies, and later versions of IEEE 1364 Verilog [1] as well as the IEEE 1800 SystemVerilog standard [2] have new capabilities in an attempt to provide a more deterministic coding paradigm for engineers. SystemC shares many of the same problems as observed in Verilog.

Nevertheless, race conditions are still inserted into many designs and it is important that these are eliminated from the Register Transfer Level (RTL) code prior to synthesis and regression verification. Although simulation with the correct test vectors may be used to find some race conditions types, a simulator will interpret temporal code operation in a specific order, making it hard to test for all possibilities. Some simulators have incorporated race condition analyzers to highlight potential issues, although often these analyzers are inadequate to track all possibilities.

Formal Verification has proven a useful platform to provide exhaustive analysis for many complex verification scenarios. Providing the technology is applied in the correct manner, formal techniques may explore the entire state space of a code segment, inherently allowing the testing of every possible scenario that may lead to an issue. This would appear to be ideal when tracking race conditions as all possible inputs to a storage element under test may be analyzed without the risk of forgetting a specific test input pattern. However, the key to providing such a rigorous test is the setting up of an automated model that reads the code and provides the correct properties to drive the formal engines. This paper will describe such a model that avoids problems with the previous methods employed.

II. A Review Of Race Conditions In HDL-Based Design

Because of the parallel nature of SystemC and other hardware description languages, the semantics of a segment of code is based on execution schedules, i.e. the order in which the different processes that are active at a certain execution step are executed sequentially. In an ideal design implementation, the choice of a particular execution schedule should not influence the behavior of the code, that is, at the end of any entire execution step, all outputs will have the same value regardless of the particular execution schedule that was chosen. Hence, simulation and synthesis tools are free to select an execution schedule that is best suited for their operation.

In the presence of several processes accessing the same variable concurrently, the choice of execution schedule may very well have an effect on the value of the registers after the execution step, potentially leading to mismatches between the interpretation of the code operation used by the synthesis and simulation tools. Hence, it is crucial to ensure that such concurrent accesses are "thread-safe," i.e., they are not affected by the chosen execution schedule.

In general there are two forms of race conditions that commonly occur in HDL based designs, and electronics design in general.

The write-write race condition is one where a variable that retains its state may be written by two or more input signals within a single write transition (usually a clock period), and depending on the interpretation of the order of code execution, either one of the inputs may provide the final stored value. A simple example of such a condition in Verilog is shown in Figure 1. Although reading the code would suggest that the flipflop value would end up as B, one simulator optimizing the code in one particular manner could easily chose the statement that writes the value of A to flipflop to be executed after the statement that writes the value of B to flipflop.

```
module writewrite (input bit clock, A, B, output bit flipflop);
always @(posedge clock) flipflop = A;
always @(posedge clock) flipflop = B;
endmodule
```

Figure 1: An example write-write race condition in Verilog

The second form of race condition is the read-write condition, where it is non-deterministic as to whether a state variable value will be read before or after it is written. Figure 2 shows a simple example of this scenario. It may be seen that when flipflop2 reads flipflop1, it may or may not hold the latest value of A depending on the event ordering employed.

```
module readwrite (input bit clock, A, output bit flipflop2);
bit flipflop1;
always @(posedge clock) flipflop1 = A;
always @(posedge clock) flipflop2 = flipflop1;
endmodule
```

Figure 2: An example read-write race condition in Verilog

Although these two examples serve to illustrate the point, realistic race conditions can occur due to reads and writes separated by 100's of lines of code, in designs where the logic that contain these conditions is extremely complex. As such it may be extremely difficult to track these problems, and an exhaustive check becomes very valuable.

The first version of the Verilog language suffered significantly from this issue. As there was only one simulator and synthesis tool for practical purposes leveraging the language, and the engineers were expert enough to eliminate synthesis, simulation mismatches in the relatively small designs being created at the time, race conditions were not a significant issue. However, as the language was opened into the public domain and alternative simulators became available, it became clear that a design that ran in one simulator would often execute differently in another, and the full impact of these race conditions became clear.

As a result the "non-blocking" assignment, represented by "<=", was introduced into the Verilog language. This assignment statement used a master slave relationship, where for a specific series of events occurring at the same time, all the right hand side of statements would be processed and their result placed into temporary variables, and then all the temporary variables would be used to write to the left hand sides of the statements. Although this solved the problem partially, it did not eliminate all race condition scenarios and was unpopular as it had a negative effect on performance and memory consumption. In the SystemVerilog standard "event regions" were added [3], which further improves the possible determinism available in the language. However, to this day, Verilog designs are still susceptible to race conditions.

The IEEE 1076 VHDL language standard [4] was designed from the ground up to avoid this issue. This is due to the fact that VHDL leverages a language paradigm that reminds the author of George Orwell's "newspeak" in his novel 1984 [5]. Dangerous constructions (the thought crimes) are disallowed in the language so that you cannot express anything that might not work (or is forbidden, as in 1984). In VHDL order dependent behavior is eliminated through the use of signals to communicate between processes, and the concept of "deltatime". All VHDL processes running in a specific time slot are fully executed before the inter-process signals are written and read, during deltatime (an additional processing period between coded time slots), thus eliminating races. It should be noted that subsequent language revisions allow multiple processes to write to variables, increasing the risk of a race. Additionally, IEEE 1076-2002 adds the notion of "shared variables" that can introduce race conditions, although these must be marked explicitly and are rarely used.

The IEEE 1666 [6] SystemC classes and macros provide a discrete event simulation class library for the parallel execution of C++ blocks, thus enabling a C++ hardware design paradigm. SystemC was designed to provide an efficient modeling style and enable fast simulation. As such, it bears more resemblance to Verilog than VHDL, in terms of both the semantics of parallelism and the method of sharing data between processes. Therefore, SystemC is susceptible to race conditions in a similar manner to Verilog. In SystemC, given the potential use scenarios of the "wait()" statement and thread behavior, these race conditions can be particularly hard to detect and may lead to significant design issues. Additionally, as SystemC can be written in a way that the hardware nature of the design is almost invisible, SystemC designers might even not be aware of the parallel semantics of the language.

III. ALTERNATIVE METHODS FOR HDL RACE CONDITION ANALYSIS

Kroening [7] shows that due to the possibility of writing to the same variable in several threads and the fact that the scheduling between threads is non-deterministic, different results can occur. Kroening proposes to build execution graphs and to detect races by performing model checking on those graphs. While this is a valid approach to solve the problem, it requires additional, specialized tools. In addition, understanding a race condition found using this solution requires additional debugging skills.

Le and Drechsler [8] propose an approach named input-output determinism where the output of a design for a given input is calculated and then compares this with all other possible outputs for different execution schedules. The SystemC model in this case is converted to C and then a software model checker used to compare the different execution schedules. This methodology is limited to the static inputs provided remaining constant, and must rely on C based software model checking, a departure from the norms of hardware design, for which SystemC was created in the first place.

For hardware language such as Verilog, on the other hand, the problem of data-races has been solved using commercial property-checking tools, such as OneSpin 360 DV™. As a by-product of compiling the design for formal analysis, all variables with potential data races are detected and for each of those variables, an assertion is created to check that a race condition cannot occur under any circumstances. This assertion can then be formally verified in the property checking tool with several results: by providing a full proof, i.e. the assurance that the race condition is indeed impossible, by ensuring that there can be no synthesis-simulation mismatch for the particular race condition, or by detecting a race condition. Such a race condition is then presented to the user in a standard hardware debugging environment, showing a simulation trace from reset where concurrent writes to the same variable occur.

IV. A PROPOSED MODEL FOR SYSTEMC RACE CONDITION ANALYSIS

This paper describes a different solution to the problem of data-races in synthesizable SystemC by leveraging a translation scheme of SystemC into Verilog, allowing the use of standard formal techniques to detect data races. By translating into Verilog, the event semantics of the HDL language may be fully leveraged, along with the proven commercial solutions to this problem.

A Verilog model is created from the SystemC model using a translation table. An abbreviated table is shown in Figure 3.

```
SystemC                 Verilog
SC_MODULE               -> module
SC_IN                   -> input
SC_OUT                  -> output
SC_METHOD               -> process
sensitive               -> processes' sensitivity list
async_reset_signal_is   -> processes' sensitivity list + reset condition
constant                -> localparam
```

Figure 3: Abbreviated Translation Table

Reset declarations are translated into sensitivity lists as well as direct sensitivity declarations.

The list above is the translation table for the example in Figure 4. Of course the full translation table is much longer. The entire SystemC synthesizable subset can be translated in this manner.

Once the translated model has been derived, relatively standard hardware model checking may be applied.

The basic idea for detecting write-write race conditions is simple: instead of actually comparing execution schedules as in [6] [7], we identify statements which variable *v* is updated with a value *value(v*). These statements may correspond to 2 different execution schedules leading to different results, one where statement i is executed before statement j, the other with statement j before statement i. For read-write races, the values don't play a role, they occur whenever the read and write conditions are not mutually exclusive.

An *accumulated condition* of a statement is the sum of all conditions that need to be true to activate this statement. For any signal s, we define the *readcondition* as the accumulated condition under which the signal is read and symmetrically the *writecondition* as the accumulated condition under which the signal is written, ordering the conditions from all processes together.

Formula for read-write races:

$$readwriterace(v) = \bigvee_i writecondition(v)_i \wedge \bigvee_i readcondition(v)_i$$

Formula for write-write-races:

$$writewriterace(v) = \bigvee_{i,j}(writecondition(v)_i \wedge writecondition(v)_j \wedge value(v)_i \neq value(v)_j)$$

If any such race occurs, the result is undefined. Note that similar to the above precise version of the write-write race formula that requires two different values being written, the simplified read-write formula above can also be refined to require that the "new" value of the variable actually matters in the context of the read location. This is illustrated later on in the example.

Both simulators and synthesis tools "choose" one order of conflicting statements. So we encounter a classical simulation-synthesis mismatch here. Simulation-synthesis mismatches are a frequent source for misbehavior of designs. Even standard equivalence checking techniques don't help here, as equivalence checkers are designed as tools that detect problems in the synthesis flow. Since the synthesis tools implements a single execution schedule, the equivalence checker will prove the correctness of the synthesis result even if there are other execution schedules that would lead to a different design behavior.

V. EXAMPLE

Figure 4 shows the example from [7].

```
const int PMAX = 49;
SC_MODULE(m) {
      sc_in<bool> clk;
      sc_in<bool> reset;
      sc_out<int> pressure;
```

```
      void guard() {
            if (pressure == PMAX)
                  pressure = PMAX - 1;
      }
      void increment() {
            pressure = pressure + 1;
      }
      SC_CTOR(m) {
            pressure = 0;

            SC_METHOD(guard);     sensitive << clk.pos();
            SC_METHOD(increment); sensitive << clk.pos();
            async_reset_signal_is(reset,true);
      }
};
```

Figure 4: Example SystemC Module

The example module in Figure 4 translates into the Verilog code shown in Figure 5, using the translation table in Figure 3.

```
localparam int PMAX=49;
module m(input clk, input reset, output int pressure);

always @(posedge clk or posedge reset)
begin: guard
      if (reset)
            pressure = 0;
      else if(pressure == PMAX)
            pressure = PMAX - 1;
end: guard

always @(posedge clk or posedge reset)
begin: increment
      if (!reset)
            pressure = pressure + 1;
end: increment
endmodule
```

Figure 5: Verilog Translation for Example SystemC Module

A potential race occurs if the value written in one process differs from the value written in a different process.

So the write-write race condition in this case would be
(from process guard)

```
      !reset && pressure == PMAX
      value: PMAX - 1
```

(from process increment)

```
      !reset
      value: pressure + 1
```

In combination:

```
      !reset && pressure == PMAX && !reset && (pressure + 1 != PMAX - 1)
```

Reduced:

```
      !reset && pressure == PMAX && (pressure != PMAX -2)
      !reset && pressure == PMAX
```

So whenever there's no reset and pressure equals `PMAX`, there's a write-write race.

As the first process reads pressure in order to decide whether to modify it, there is an additional read-write race evaluating `pressure==PMAX`. For a simplified view of a read-write race as given by formula in this paper, the process reads `pressure` whenever there is no reset, and the other process increments `pressure` whenever there is no reset. So there is a read-write race whenever there is no reset.
However, since the actual reading of pressure only computes `pressure==PMAX`, many of these read-write races are not relevant since the difference of the value of `pressure` before and after the increment is not relevant for evaluating `pressure==PMAX`. For a refined read-write race, we therefore get:

```
!reset && ((pressure == PMAX) != (pressure+1 == PMAX))
```

Reduced:

```
!reset && (pressure == PMAX || pressure == PMAX-1)
```

For a hardware language like Verilog, the write-write race can be refined to bits, so that a race condition is only flagged when a bit value is different, allowing for concurrent accesses to individual bits of a signal.

The only safe way out of this is to guard conflicting read and write statements and make them mutually exclusive.

Figure 6 shows the race condition displayed in the OneSpin 360 DV Verify debug user interface. As can be seen in the figure, `pressure` is being incremented until up to the value "49" or `PMAX` where the race is occurs. The light red background coloring of the statements depicts their concurrent activity, which leads to the observed race.

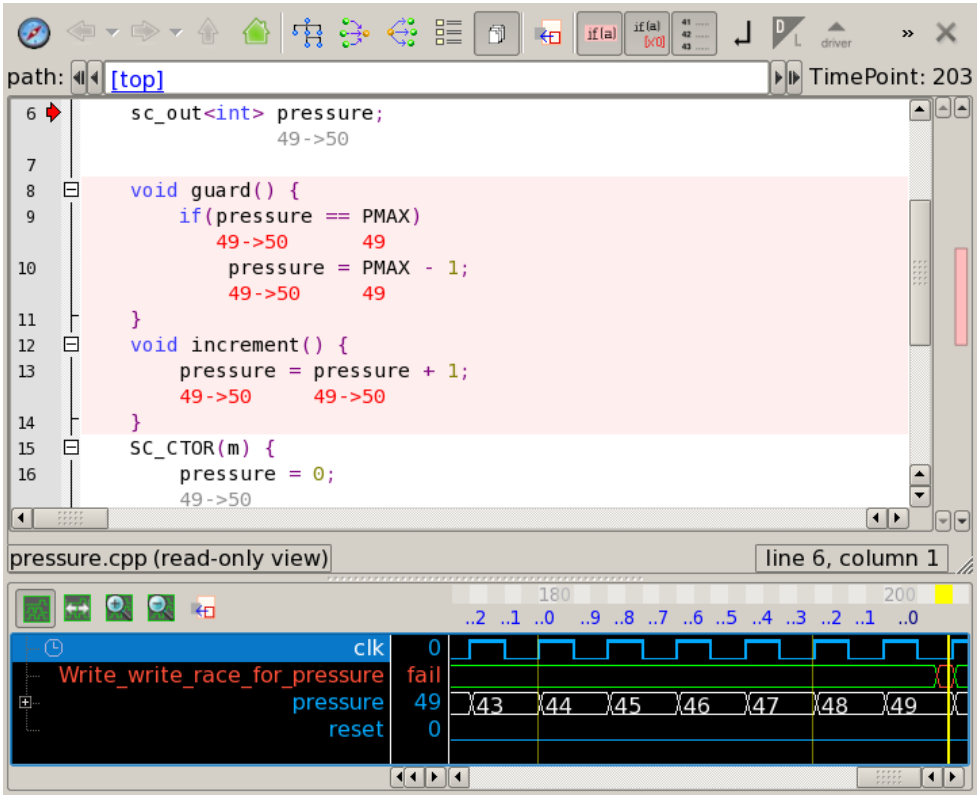

Figure 6: Race Condition Displayed in OneSpin 360 DV Debug Window

Let us now try to make the SystemC code robust against a write-write race. We first introduce a further input to guard the increment assignment. This is illustrated in Figure 7.

```
const int PMAX = 49;
```

```
SC_MODULE(m) {
      sc_in<bool> clk;
      sc_in<bool> reset;
      sc_in<bool> guard_inc;
      sc_out<int> pressure;

      void guard() {
            if (pressure >= PMAX)
                  pressure = PMAX-1;
      }
      void increment() {
            if (guard_inc)
                  pressure = pressure + 1;
      }
      SC_CTOR(m) {
            pressure = 0;

            SC_METHOD(guard);     sensitive << clk.pos();
            SC_METHOD(increment); sensitive << clk.pos();
            async_reset_signal_is(reset,true);
      }
};
```

Figure 7: SystemC Example Re-coded with additional guard

Of course, the two assignments can still happen simultaneously, so the environment must make sure that the guard is activated properly. This leads to another strong advantage of using a SystemC-to-Verilog translation scheme and a formal ABV tool capable of handling assertion languages like SVA; the SystemC environment can be modeled in SVA:

```
exclusive: assume property
      (@(posedge m.clk) m.pressure>=m.PMAX-1 |-> ~m.guard_inc);
```

This constraint ensures that the guard can only be activated if pressure is below the threshold. If this constraint is used for the SystemC code in OneSpin 360DV Verify, the absence of race conditions can be proven.
Of course, there is a plethora of applications for an assertion language like SVA on top of SystemC – in addition to modeling the environment, assertions about the SystemC code can be written and proven or disproven with formal analysis. Note that this covers normal state expressions in SystemVerilog – very similar to C-style assert statements, but is not restricted to state expressions. In other words, the full capability of SVA can be used to express timed assertions.

VI. CONCLUSION

Race conditions have dogged electronic design since the invention of digital circuits, and the advent of hardware description languages simply compounded their impact on design. Although different approaches, including language capabilities, have been employed to reduce or eliminate their impact, they still persist in driving bug conditions in IC designs. SystemC is no different to other HDLs in this regard, and the possible complexity of some SystemC designs can make detecting races difficult. This paper proposes a method whereby a SystemC design might be analyzed for race conditions through the use of a formal tool. This provides an assured method of eliminating all possible races through the technique detailed in the paper.

More generally, the paper demonstrates how a standard technique leveraged for SystemVerilog designs can be extended to SystemC, exploiting the full power of formal tools including, for example constraining, which would need to be implemented additionally in any other method. The detection of race conditions in SystemC is an

important capability, and by leveraging this approach, the method is applicable to any kind of races/simulation-synthesis mismatches. This approach could be used to apply other formal based techniques on SystemC designs.

REFERENCES

[1] IEEE 1364 Verilog Standard
[2] IEEE 1800 SystemVerilog Standard
[3] Cummings and Salz, SystemVerilog Event Regions, Race Avoidance and Guidelines, SNUIG Boston, 2006
[4] IEEE 1076 VHDL Language Standard
[5] George Orwell, Nineteen Eighty Four, Secker and Warburg, 1949
[6] IEEE 1666 SystemC Standard
[7] Daniel Kroening and Nicolas Blanc, Race Analysis for SystemC using Model Checking, ACM, 2010
[8] Hoang Le and Rolf Drechsler, Toward Verifying Determinism of SystemC Designs, DATE, 2014