SystemVerilog-2009 Enhancements: Priority/Unique/Unique0

Clifford E. Cummings

Sunburst Design, Inc.
14355 SW Allen Blvd., Suite #100
Beaverton, OR 97005
1-503-641-8446

cliffc@sunburst-design.com

ABSTRACT

SystemVerilog-2009 enhanced the **`case`** / **`if`** modifiers, **`priority`** & **`unique`**, with new scheduling semantics to help correctly identify more bugs during simulation. SystemVerilog-2009 also added a new **`unique0`** keyword to emulate the **`parallel_case`** capability of synthesis tools. This paper will detail how these two enhancements will help to accurately identify more bugs in a design and provide better synthesis results for a certain class of designs.

General Terms

Verilog, SystemVerilog, full_case, parallel_case, priority, unique, unique0, simulation, synthesis.

Keywords

SystemVerilog, full_case, parallel_case, priority, unique, unique0.

1. INTRODUCTION

The SystemVerilog **`priority`** and **`unique`** keywords, described in this paper, actually pre-date the SystemVerilog language and were part of the Superlog language, most of which was donated to Accellera and became the foundation for the early Accellera SystemVerilog 3.0 Standard[7]. These keywords offered a simulation-aware replacement for the existing comment-style synthesis directives **`full_case`** and **`parallel_case`**.

Although the new SystemVerilog keywords made it possible to detect illegal simulation conditions that would cause a mismatch between pre- and port-synthesis simulations, over the past eight years it was discovered through usage of these constructs in SystemVerilog designs that their defined semantics still lacked minor features and important error-trapping capabilities.

The SystemVerilog-2009 Standard[6] has taken important steps to close the shortfalls in both features and error detection. This paper describes existing priority and unique semantics and then introduces the new **`unique0`** keyword semantics and the potential for enhanced error trapping capabilities. The error trapping is still insufficient for most design teams but recommendations are included at the end of this paper to describe how EDA tool vendors can remedy the insufficiencies that still exist through tool command lines options (see section 9).

2. FULL_CASE BASICS

When an engineer declares that a **`case`** statement is a **`full_case case`** statement, the engineer is asserting that all possible matching conditions have been listed as case items and that any unlisted possibility is not reachable by the actual logic; therefore, the other conditions are don't-care conditions.

An easy way to understand **`full_case`** is to compare a case statement to a Karnaugh map (K-map) as taught in undergraduate engineering classes.

In the K-map for any given output variable, the engineer notes when the output is a **`1`** and when it is a **`0`**. If there are input conditions that cannot be reached, those K-map squares are filled with **`X`**'s to indicate don't-care conditions. The synthesis tool can then optionally incorporate the **`X`**'s to build simpler product terms for the K-map. The simpler product terms translate into smaller and faster logic.

It should be noted that the **`full_case`** directive is disabled if the **`case`** statement includes a **`case-default`** statement.

It should also be noted that one typically will get the same synthesis results if the **`full_case`** directive is replaced with a **`case-default`** where every signal assigned in the body of the **`case`** statement is assigned the value of **`X`** in the **`case-default`**.

The biggest problem with the **`full_case`** directive is that it is a potential command to the synthesis tool but it is a comment to the simulator, so if the **`full_case`** directive causes the synthesis tool to take actions to optimize the design, those same optimizations will not be executed by the simulator, which makes it possible to have a mismatch between pre- and post-synthesis simulations. This can be the source of design problems in the final synthesized design[1].

3. PARALLEL_CASE BASICS

When an engineer declares that a case statement is a **`parallel_case case`** statement, the engineer is asserting that it is only possible to match the case expression to one and only one (or none) of the case items. The engineer has declared that the

case expression shall only uniquely match up to only one of the case items.

This declaration is intended to inform the synthesis tool that no priority encoders are required to build the logic in the **case** statement and that each tested case item in the **case** statement can be treated as if it were a unique **if** statement. The resultant logic would be both smaller and faster than if the case items had been assembled into a set of priority logic.

Like the **full_case** directive, the biggest problem with the **parallel_case** directive is that it is a potential command to the synthesis tool but it is a comment to the simulator, so if the **parallel_case** directive causes the synthesis tool to take actions to optimize the design, those same optimizations will not be executed by the simulator, which again makes it possible to have a mismatch between pre- and post-synthesis simulations. This again can be the source of design problems in the final synthesized design.

4. PRIORITY KEYWORD

The **priority** keyword was added to the SystemVerilog language to be a simulation-aware replacement for the **full_case** comment-style directive[5].

Apology - on behalf of most of the SystemVerilog Standards Group, I apologize for the choice of the **priority** keyword. Most on the SV Standards Group believe it is a terrible keyword that does not describe the actual behavior imposed by this keyword. A **case** statement is already a priority statement. Within both Verilog and SystemVerilog, if the **case** expression matches a case item, the case item expression is executed and there is an implied break that causes the **case** statement to skip testing all of the trailing case items, which raises the question, how does a **priority** keyword change the implementation of the design if a **case** statement already behaves like a priority expression? The **priority** keyword does not add priority semantics to a **case** statement because, by its nature, a **case** statement already has priority semantics. So what does the **priority** keyword really do?

From a synthesis perspective, the **priority** keyword really has the same semantics as the old **full_case** directive. Most of the SystemVerilog Standards Group believes that it would have been better to replace the **priority** keyword with either a ***full_case*** or ***all_cases*** keyword. The latter two would have provided a better description of the intended behavior. Unfortunately, we discovered this confusion and potential solution too late.

Perhaps the SystemVerilog Standards Group should consider adding one of the keywords, ***full_case*** or ***all_cases*** as a synonym for the **priority** keyword.

4.1 Priority Case

As mentioned above, a **priority case** construct informs the synthesis tool that all possible cases have been defined and that any unlisted case items can be used as don't-cares during synthesis optimization.

The advantages that the **priority case** statement has over the **full_case** statement are: (1) the **full_case** is just a comment to the simulator, while **priority case** is an assertion that is tested during simulation, and (2) if during simulation a **priority case** command is executed but the case expression does not match any of the case items, the simulator will report a warning or violation (see sections 7 and 8).

When an engineer adds the **priority** keyword to a **case** statement, the engineer has assumed that it is only possible to match the defined case items in the **case** statement and therefore it is safe to treat all other potential matching patterns as don't-cares. If during simulation, none of the listed case items matches the case expression, then the engineer's initial assumption was wrong and it should be flagged as some form of violation for the engineer to fix, because the synthesis tool has been directed to treat the unspecified pattern as a don't-care. From a designer's perspective, the best simulation behavior under these circumstances would be to abort the simulation with a violation message (see sections 8 and 9). This would help eliminate incorrect design assumptions that could cause the design to have a fatal design flaw.

If the **priority case** statement contains a **case-default** statement, the **priority** testing will be disabled because every execution of the **case** statement must match *something*, even if it is just the **default** statement.

4.2 Priority If

The ability to add the **priority** keyword to an **if** statement is new to SystemVerilog-2005 and allows an engineer to specify **if-else-if** statements where all conditions that would be covered by an **else** statement are treated as don't-care conditions by a synthesis tool.

Just as a **case-default** disables the effects of a **priority case**, an **else**-statement attached to the end of a **priority-if-else-if** statement will disable the effects of the **priority** keyword.

As with the **priority case** statement, the **priority if** statement requires that one of the tested conditions be matched during simulation, otherwise the simulator is required to report a violation message.

5. UNIQUE KEYWORD

The **unique** keyword was added to the SystemVerilog language to be a simulation-aware replacement for the combined **full_case parallel_case** comment-style directives.

On behalf of most of the SystemVerilog Standards Group, unlike the **priority** keyword, we are quite proud of the **unique** keyword. Most of us believe that the **unique** keyword is a far superior description of the intended check over the older and equivalent **full_case** and **parallel_case** comment-style synthesis commands.

The **unique** keyword directs the compiler and/or simulator to ensure that whenever the corresponding **case**-statement or **if**-statement is executed, that one of the tested items is matched and that it is possible to match one and only one of the tested alternatives, that the testing uniquely matches one of the tested items and can uniquely match only one of the alternatives.

Uniqueness testing can also be treated like a onehot test, meaning that only one of the tested alternatives can match, but also that

one of the tested alternatives MUST match. So the **unique** testing also encompasses the **priority** testing requirement, which is why an engineer will never use both **unique** and **priority** on the same **if** or **case** statement (unlike the common grouping of **full_case parallel_case**).

5.1 Unique Case

As mentioned above, a **unique case** construct informs the synthesis tool that all possible cases have been defined, that any unlisted case items can be used as don't-cares during synthesis optimization and that the case expression shall only match one of the case items so it is not necessary for the synthesis tool to build any priority logic dependencies between the case items.

The advantages that the **unique case** statement has over the **full_case parallel_case** directive are: (1) the **full_case parallel_case** directive is just a comment to the simulator, while **unique case** is an assertion that is tested during simulation, (2) if during simulation a **unique case** command is executed but the case expression does not match any of the case items, the simulator will report a warning or violation and (3) if during simulation it is determined that more than one of the case items of a **unique case** could be executed, the simulator will report a uniqueness violation.

5.2 Unique If

The ability to add the **unique** keyword to an **if** statement is new to SystemVerilog-2005 and allows an engineer to specify **if**-**else**-**if** statements where all conditions that would be covered by an **else** statement are treated as don't-care conditions by a synthesis tool. It also asserts that it is not possible during simulation to match more than one of the **if**-tested conditions, which informs a synthesis tool that no large and slow priority encoders are required to build the **if**-specified logic equations.

Just as a **case**-**default** disables the effects of the **full_case** testing, an **else**-statement attached to the end of a **unique**-**if**-**else**-**if** statement will disable the **priority**-**case** effects (fullness testing) of the **unique** keyword but still retain the uniqueness testing.

6. UNIQUE0 KEYWORD (*NEW*)

Since the introduction of the **priority** and **unique** keywords, it has been observed that there are some designs that would benefit from the uniqueness testing without the requirement to match one of the **case** items or **if**-**else**-**if** tested items. For this reason, the **unique0** keyword was added to the SystemVerilog-2009 Standard[6].

The **unique0** keyword is a simulation equivalent to the older **parallel_case** comment-style synthesis directive.

6.1 Unique0 Case (*New*)

To understand the motivation for the **unique0** keyword, consider the example of an efficient 2-to-4 decoder in Example 1. At the top of the **always_comb** procedure, the **y**-output is initialized to **0**, and then only if the enable is set to **1** will one of the four outputs be reset to **0** inside of the **case** statement.

```
module dec2_4a (
  output logic [3:0] y,
  input  logic [1:0] a,
  input  logic       en);

  always_comb begin
    y = '0;
    case ({en,a})
      3'b100 : y[a]='1;
      3'b101 : y[a]='1;
      3'b110 : y[a]='1;
      3'b111 : y[a]='1;
    endcase
  end
endmodule
```

Example 1 - Efficient 2-to-4 decoder model (no priority or unique keywords used)

This **case** statement is not "full" because it does not list any of the cases when enable is **0**. This style is a recommended coding style because all of the default cases have been covered by the initial assignment at the top of the procedure (guarantees removal of latches) and then all of the exception conditions are noted in the **case** statement.

Sample synthesized logic for example Example 1 is shown in Figure 1.

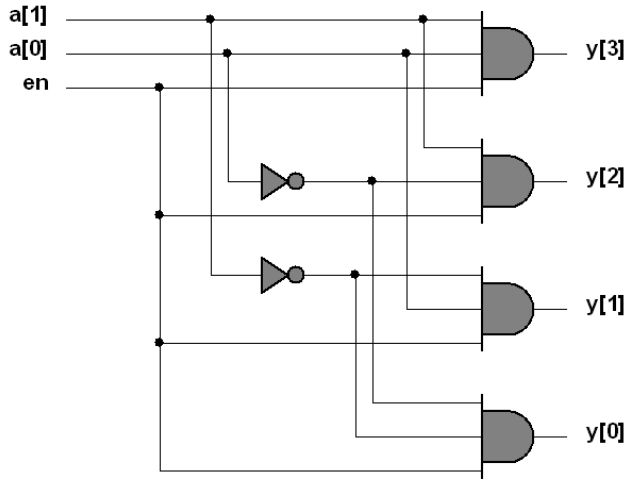


Figure 1 - 2-to-4 Decoder with enable - correct synthesis result

SystemVerilog added the **unique** keyword, which allowed the simulator to give run-time warnings if (a) the case expression could match more than one of the case items, or (b) if the case expression did not match any of the case items.

The **unique** keyword informs the synthesis tool that (a) the case items are unique so do not build priority logic base on case-item order, and (b) all possible case conditions are listed (this is a full case) so the output for any unspecified case item combination can be treated as a don't-care. The latter condition overrides any pre-default assignment that might have been specified at the top of the procedure.

Now consider the 2-to-4 decoder of example Example 2. The code is identical to example Example 1 except the **unique** keyword has been added to the **case** statement.

```
module dec2_4b (
  output logic [3:0] y,
  input  logic [1:0] a,
  input  logic       en);

  always_comb begin
    y = '0;
    unique case ({en,a})
      3'b100 : y[a]='1;
      3'b101 : y[a]='1;
      3'b110 : y[a]='1;
      3'b111 : y[a]='1;
    endcase
  end
endmodule
```

Example 2 - Flawed 2-to-4 decoder model with unique case statement

Upon examination, it can be seen that the **case** statement only defines four out of eight possible case conditions. The four conditions defined are the four conditions when the **en** input is set to **1**. The **unique** keyword informs the synthesis tool that the four conditions when **en** is **0** are don't-care conditions (overriding the pre-default at the top of the **always_comb** procedure). The synthesis tool therefore concludes that since the output is a don't-care whenever the **en** input is low, that the **en** input is a don't-care and the **en** input is optimized out of the design as shown in Figure 2.

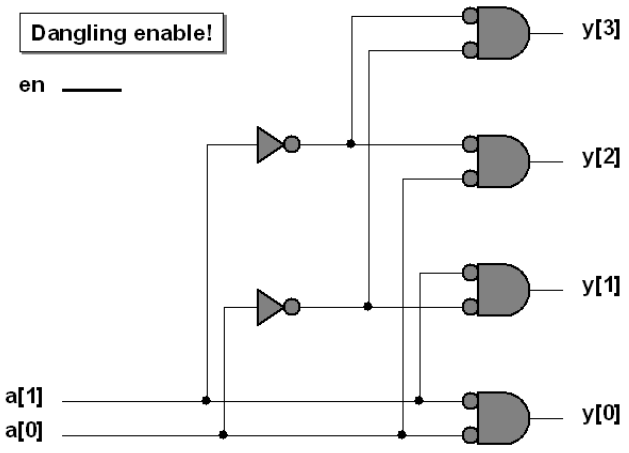


Figure 2 - 2-to-4 Decoder with dangling enable - WRONG synthesis result

Since **unique** is a recognized simulation keyword with simulation semantics, the simulation should report a violation whenever the **always_comb** procedure is executed when **en** = **0**.

For the 2-to-4 decoder model, it is desirable to add uniqueness testing without adding the **full_case** testing and optimization. Prior the to addition of the **unique0** keyword, in order to cancel the **full_case** simulation testing and synthesis optimization, an engineer had to add an empty **case-default** (remember, the **case-default** cancels the **priority case** testing).

```
module dec2_4c (
  output logic [3:0] y,
  input  logic [1:0] a,
  input  logic       en);

  always_comb begin
    y = '0;
    unique case ({en,a})
      3'b100 : y[a]='1;
      3'b101 : y[a]='1;
      3'b110 : y[a]='1;
      3'b111 : y[a]='1;
      default: ; // empty default
    endcase
  end
endmodule
```

Example 3 - SystemVerilog-2005 empty-default work-around to replicate parallel_case functionality

Although this is a reasonable work-around, it is a rather awkward looking piece of code that often requires explanation.

To avoid the awkward, empty-**default** of Example 3, SystemVerilog-2009 added the ability to use a **unique0** directive that would allow uniqueness testing while avoiding the **full_case** testing without the addition of the awkward empty **default** statement.

Example 4 shows the preferred use of the **unique0 case** statement. This example code synthesizes to the correct 2-to-4 decoder implementation, just like the implementation shown in Figure 1.

```
module dec2_4d (
  output logic [3:0] y,
  input  logic [1:0] a,
  input  logic       en);

  always_comb begin
    y = '0;
    unique0 case ({en,a})
      3'b100 : y[a]='1;
      3'b101 : y[a]='1;
      3'b110 : y[a]='1;
      3'b111 : y[a]='1;
    endcase
  end
endmodule
```

Example 4 - SystemVerilog-2009 unique0 case decoder to replicate parallel_case functionality

One of the nice features of the **unique0** keyword is that it forces the simulator to ensure that either none or only one of the case items can be reached during execution of any **case** statement.

When one compares making a pre-default assignment prior to a **case** statement, to adding a **case-default**, I have found that making the pre-default assignment is both more effective at removing latches and typically yields equal or better synthesis

results. Why does a pre-default remove latches better than a **case-default**?

Consider the 2-to-4 decoder with case-default in Example 5.

```
module dec2_4e (
  output logic [3:0] y,
  input  logic [1:0] a,
  input  logic       en);

  always_comb begin
    unique case ({en,a})
      3'b100 : y[a]='1;
      3'b101 : y[a]='1;
      3'b110 : y[a]='1;
      3'b111 : y[a]='1;
      default: y = '0;
    endcase
  end
endmodule
```

Example 5 - ERROR - 2-to-4 decoder with case-default - infers latches

In this decoder example, the combination of **unique case** with **case-default** appears to have all possible cases covered, but the common mistake is that the explicit case items only set one of the four outputs, which means that the other three outputs must remain unchanged (they must be latched).

It is very easy to make a latch-inference coding mistake using a **case-default** when multiple outputs are assigned in the same **case** statement. If the same **case-default** is repositioned to be a pre-default assignment at the top of the **always_comb** block, then all latches will be removed. As long as all outputs are assigned to anything (even **X**'s) at the top of the procedure, no latch inference will occur. This technique is simple, effective and synthesis efficient.

6.2 Unique0 If (*New*)

The SystemVerilog-2009 Standard also allows the **unique0** keyword to be added to an **if-else-if** statement.

Just like a **unique0 case** , the **unique0 if** imposes uniqueness testing but does not require any of the tested if conditions to be matched during simulation. In synthesis, no priority encoder will be built from the conditions tested in the **if-else-if** statement.

7. SYSTEMVERILOG-2005 WARNINGS

SystemVerilog-2005 requires a simulator to report warnings if a **priority** or **unique** test-expression does not match any of the listed **case** items or **if**-tests.

SystemVerilog-2005 similarly requires a simulator to report warnings if a **unique case** test-expression would match more than one of the listed case items during simulation, or if more than one of the **unique if**-test conditions would match during simulation.

The **priority** testing is essentially a strict must-match test while the **unique** testing is essentially a strict onehot test.

Unfortunately, because these are warnings and not fatal errors, they are easy to miss, and many engineers have told me that **priority**/**unique** cannot be reliably used because it is too easy for design errors to go unnoticed. Engineers have requested a more strongly tested version of the same constructs; hence, the SystemVerilog-2009 violation enhancement of the next section.

8. SYSTEMVERILOG-2009 VIOLATIONS (*NEW*)

In SystemVerilog-2009, a new type of violation checking replaces the SystemVerilog-2005 warning checks for **unique** and **priority** constructs, as well as the new **unique0** construct. Strictly speaking, the violation checking is not a stronger check; it is just a different check, but with user encouragement, we might convince EDA vendors to turn the violation into a stronger check (see section 9).

Clause 12.4.2.1 of the new SystemVerilog-2009 standard states:

> A **unique**, **unique0**, or **priority** violation check is evaluated at the time the statement is executed, but violation reporting is deferred until the Observed region of the current time step.

Since all module RTL code is executed in the Active Region set that includes the Active events region where all of the properly coded[2] always_comb procedures are executed, the combinational logic will iterate and settle out within the Active region before any violations can be reported in the Observed region (see Figure 3).

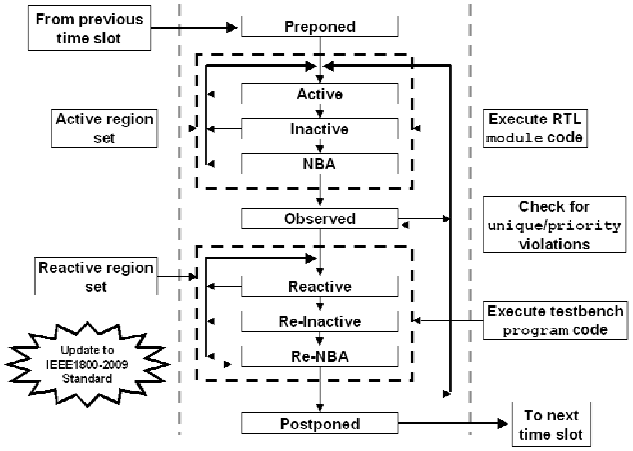


Figure 3 - SystemVerilog-2009 Event Scheduling - Violation checks in Observed Region

Consider the fully coded 2-to-1 mux example based on the example code in clause 12.5.3.1 of the IEEE SystemVerilog-2009 Standard.

The descriptions in 12.5.3 mentions several cases in which a violation report shall be generated by **unique-case**, **unique0-case**, or **priority-case** statements. These violation checks shall be immune to false violation reports due to zero-delay glitches in the active region set (see 4.4.1).

The mechanics of handling zero-delay glitches shall be identical to those used when processing zero-delay glitches for **unique-if**, **unique0-if**, and **priority-if** constructs (see 12.4.2.1).

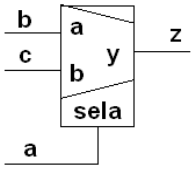
The following is an example of a **unique-case** that is immune to zero-delay glitches in the active region set:

```
module sv_logic1 (
  output logic z,
  input  logic a, b, c);

  logic not_a;

  always_comb begin: a1
    not_a = ~a;
  end

  always_comb begin: b1
    unique case (1'b1)
      a     : z = b;
      not_a : z = c;
    endcase
  end
endmodule
```



Example 6 - 2-to-1 mux implemented with two concurrent always_comb blocks

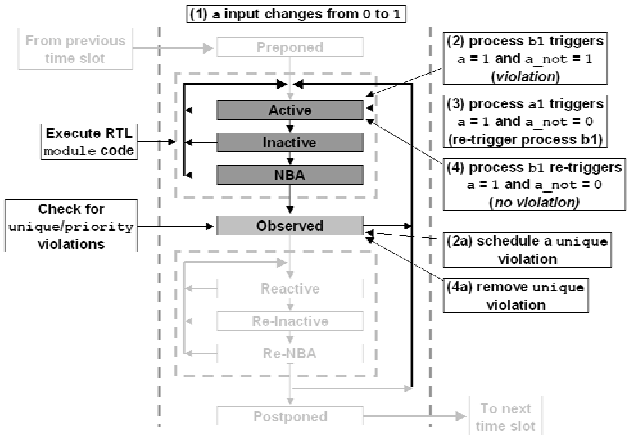


Figure 4 - Violation scheduling and removal from concurrent always_comb blocks

In this example the **unique case** is checking for overlap in the two case_item selects. When **a** and **not_a** are in states **0** and **1** respectively and **a** transitions to **1**, the following sequence of events can happen (see Figure 4):

(1) **a** changes from **0** to **1**, which would cause process **a1** and **b1** to trigger with indeterminate order. For this example, assume that process **b1** triggers first.

(2) Process **b1** triggers and both **a** and **not_a** both momentarily have the values of **1**.

(2a) In process **b1**, the **unique case** could be executed while **a** and **not_a** are both true, so the violation check for uniqueness will fail, and the scheduled failure will be reported in the Observed region.

(3) When process **b1** completes, process **a1** would then trigger and invert the value of **not_a**.

(4) When **not_a** is inverted, it will again trigger process **b1** while still in the Active events region.

(4a) In process **b1**, the **unique case** will determine that there is no overlap in the case items, so the scheduled violation in the Observed region will be flushed. From this example, it can be seen that although there was a momentary glitch where the **unique case** would have reported a violation, the logic settled to a valid state to satisfy the **unique case** assumption and no violation will be reported.

A note about violations and combinational logic. The violation enhancement works with 0-delay combinational RTL code, which is sufficient for most RTL coders. If the separate assignments had unit delays between updates, there could be multiple combinational settling violations during simulation. One the design is synthesized and rendered into a gate-level representation, the **unique**/**unique0**/**priority** keywords will have been removed so even gate-level models with delays will not be subject to these RTL-related violations.

9. FATAL ON VIOLATION

Although the introduction of violations is a good first step, most engineers want a simulation to abort on an error. Although not required by the IEEE Std 1800-2009, it would be most useful if EDA vendors would provide a command line switch to enable aborting on a violation of the **unique**/**unique0**/**priority** constructs. The command line switch would be optional and could be easily turned off to disable aborting on violations.

The only reason to add the **unique**/**unique0**/**priority** keywords to an RTL design is to inform the synthesis tool of a design assumption that would allow optimization of the RTL design during synthesis. If the assumption is incorrect, the engineer would like to be notified by having a simulation abort, forcing the engineer to examine the circumstances that caused the design to abort.

If it becomes possible to enable abort-on-violation for the **unique**/**unique0**/**priority** constructs, many design teams would mandate that the switch be permanently enabled during simulation to catch these potentially fatal design errors.

If vendors are unwilling to provide the abort-on-violation capability, then this enhancement has done nothing more than to change the warning messages from "warning" to "violation" and little has been gained.

Engineers, unite! Tell your vendors that you would like a vendor option (such as a command line switch) to force **priority**/**unique** violations to cause the simulation to abort.

10. ACKNOWLEDGEMENTS

My thanks to my friends and colleagues Heath Chambers of HMC Design Verification and Shalom Bresticker of Intel Israel for offering valuable suggestions to improve the quality and content of this paper.

11. REFERENCES

[1] Clifford E. Cummings, '"full_case parallel_case", the Evil Twins of Verilog Synthesis,' SNUG'99 Boston Proceedings, Boston, MA, 1999. Also available at www.sunburst-design.com/papers

[2] Clifford E. Cummings, “Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!,” SNUG-2000 proceedings, March 2000. Also available at www.sunburst-design.com/papers

[3] Don Mills and Clifford E. Cummings, “RTL Coding Styles That Yield Simulation and Synthesis Mismatches,” SNUG-1999 Proceedings, March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[4] "IEEE Standard Verilog Hardware Description Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001

[5] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005

[6] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2009

[7] "SystemVerilog 3.0 Accellera's Extensions to Verilog," Accellera, 2002, freely downloadable from: www.systemverilog.org