

# **Creating Stimulus and Stimulating Creativity: Using the VMM Scenario Generator**

Jonathan Bromley

Doulos Ltd, Ringwood, England

jonathan.bromley@doulos.com

## **ABSTRACT**

The Verification Methodology Manual for SystemVerilog (VMM) standard library provides a scenario generator that can be built automatically to match any user-defined stimulus data class. This ready-to-run generator can then be customized to provide structured streams of randomized stimulus for a verification environment. This paper presents an introduction to simple customization of the VMM scenario generator, and then reports the experience with the management of more complex stimulus. It will also compare the tradeoffs between procedurally-generated scenarios and purely declarative, constraint-driven scenarios, and examine the potential of well-designed scenario generation to enhance effective re-use of verification components.

## Table of Contents

1.	Introduction.....	3
2.	Ad Hoc Approaches.....	3
3.	The VMM Scenario Generator.....	3
4.	Limitations of Preconfigured Scenarios.....	3
5.	Procedurally Constructed Scenarios.....	4
6.	Extensibility of Scenarios.....	4
7.	Hierarchical Scenario Generation.....	4
8.	Scenarios as a Contributor to Verification Component re-use.....	4
9.	Coordination of Scenarios - Interrupts, Multiple Streams.....	4
10.	Conclusions.....	4
11.	Acknowledgements.....	5
12.	References.....	5

## **1. Introduction**

This paper reviews the motivation for structured random stimulus generation in published verification methodologies, and outlines the solutions offered by the Verification Methodology Manual for SystemVerilog (VMM). This paper will introduce a rather straightforward verification problem that nevertheless requires structured stimulus. This example will be used throughout the rest of the paper. Section 3 describes the specific mechanisms provided in VMM for creating structured stimulus (scenarios). This section is written in a tutorial style, as it is the author's experience that many VMM users have chosen not to adopt scenarios in their own verification environments. Section 5 describes an alternative approach to using the existing VMM scenario generator, creating stimulus procedurally rather than by randomization. Although this approach is insufficiently flexible for most realistic situations, it offers some hints for how to generate more elaborate hierarchical scenarios. Section 7 describes in detail the author's prototype extensions of the VMM scenario generator to create hierarchically structured scenarios. Section 9 sketches how the modified generator could facilitate the coordination of stimulus across multiple data streams. Finally, section 10 summarizes the results and indicates some further work that is required to make the extensions easily useful in mainstream verification problems.

## **2. Ad Hoc Approaches**

Sketch some approaches that can provide the desired stimulus whilst keeping some randomization. Limitations of ad hoc approaches: re-use, integration in self-contained verification components.

## **3. The VMM Scenario Generator**

Outline scenario generator macro, components/classes created by it, simple example of its use.

### **3.1 Simple Scenarios in VMM**

Read-modify-write and burst examples worked through in VMM, with code examples. The atomic scenario. Populating the scenario\_set to get new scenarios in the mix. Configuring the scenario generator for an appropriate mix of scenarios. Scenario IDs, multiple scenarios in a single class. The apply method.

### **3.2 Structuring a Scenario: Populating the Items Array**

The items array as an array of factory objects. Scenario structure imposed by the attributes of each of these factory objects. Techniques for populating the array as part of randomization, using allocate\_scenario and fill\_scenario. Alternative approaches that don't depend on re-populating the items array.

## **4. Limitations of Preconfigured Scenarios**

All techniques described so far depend on constructing the entire scenario up-front and then delivering its items one by one. Discuss situations in which that's inappropriate or inconvenient – user may want constraints on individual items to be affected by other activity at the time of the

item's application. Can't re-randomize the item at that time, because scenario-wide constraints would not work correctly for single-item randomization.

## **5. Procedurally Constructed Scenarios**

The apply method can construct the scenario on-the-fly. Show possible techniques, including use of randcase and randsequence to create interesting activity. Examples of on-the-fly scenario generation influenced by other activity in the testbench.

## **6. Extensibility of Scenarios**

Practical considerations of how to add to a set of available scenarios. How to make the available set visible to "user" code, especially pre-existing code? "Extensible enum" design pattern, compare and contrast with scenario\_kind mechanism. Randomization of scenario type – examples of how it works with scenario\_kind.

## **7. Hierarchical Scenario Generation**

(NOTE: some more technical work yet to do on this) Allow each item in a scenario to represent either a single transaction or a scenario. Based on type matching: items[] is an array of scenarios, not transactions; if current item is an atomic scenario, then its apply method drives out a single transaction; otherwise, its apply method calls apply for each member of the items array. Inspiration from 'e' sequences, but I won't mention that explicitly in the paper! Example of application to instruction stream generation for a CPU. Maintaining a count of scenario nesting depth. Scenario debug and visualization.

## **8. Scenarios as a Contributor to Verification Component Re-use**

Scenario generator as part of a reusable verification component. Controlling a verification component's built-in scenario generator: predefined controls, additional constraints supplied in class extensions. Collecting meaningful coverage on scenario generation. Extensibility of built-in scenario generator – writing and installing custom extensions.

## **9. Coordination of Scenarios - Interrupts, Multiple Streams**

(NOTE: quite a lot of technical work yet to do on this!) Using the inject method to insert an "interrupt" scenario into an existing scenario stream. Using notifications to coordinate this activity.

Multiple coordinated streams: using start/stop methods and notifications of scenario generator; other approaches including brief overview of XVC mechanism (from VMM book).

## **10. Conclusions**

This paper has indicated the motivation for enhanced scenario generation, and described a prototype hierarchical scenario generator that has provided useful initial results. For this mechanism to be widely useful, further work is required to integrate it more closely with other standard VMM mechanisms that have been ignored or sidestepped in this prototype implementation. In particular, it is necessary to respect VMM's conventions concerning callbacks, notifications and message logging. All this can be integrated in the new base classes

without much difficulty, and the author aims to report on a more robust implementation in due course.

Finally, it is clear that the new base classes need to be specialized for transaction data type. In the VMM Standard Library, such specialization is undertaken by macros that write the base classes corresponding to a user's transaction data type. It is clear that similar macros need to be created to support the proposed new techniques.

## **11. Acknowledgements**

## **12. References**

to VMM book of course!