On Coverage and Debug in Random Stimulus Generation

Marat Teplitsky
Cadence Design Systems, Israel
marat@cadence.com

Amit Metodi
Cadence Design Systems, Israel
ametodi@cadence.com

Reuven Naveh
Cadence Design Systems, Israel
rnaveh@cadence.com

Abstract—**Randomly generated stimulus in functional verification provides significant advantages in validating a wide range of scenarios. However, random generation requires a more sophisticated verification environment, one that is more difficult to build and maintain. To enjoy the benefits that random stimuli has to offer, verification teams need to rely on their tools to fully cover the specified DUT functionality and to provide the right debug technology to cope with the complications arise. In this paper, we explore these requirements. We investigate the distribution aspects required to maximize the functional coverage, and we discuss how to effectively debug a verification environment with random components.**

I. Introduction

Functional verification tests a hardware device by simulating the device-under-test (DUT) functionality. Verification is performed by stimulating the DUT with a wide variety of inputs to ensure that all aspects of the DUT have been tested. As design complexity grows, the traditional approach of direct testing is becoming increasingly obsolete.

In its place, a *random test generation* approach became the heart and soul of verification. The key to success of random stimulus is that by default it injects the DUT with different scenarios each time, and less manual work is required to generate a huge number of stimuli for the DUT.

Of course the stimuli cannot be completely random. The inputs and outputs of hardware modules obey certain protocols and specifications. To generate such random stimuli, a *constrained random verification* technology was developed. With random generators, verification engineers could have described the legal stimuli of the DUT using a constraint language. This constraint language became an essential part of the *Hardware Verification Languages* (HVL). First in the *'e'* language [1, IEEE Standard 1647], and later on, adopted by other languages such as SystemVerilog (SV) [3, IEEE Standard 1800] and SystemC [2, IEEE Standard 1666][1].

Furthermore, the constraint languages usage methodology has been extended to direct the random stimuli generator to cover the more interesting or bug-prone areas of the DUT. In even more extent, *'e'* hardware verification language standard extended its constraint language beyond random scalar numbers, to generation of objects and hierarchies. In this way, constraint rules could define the structural hierarchy of the verification topology or a data-item. This enhancement farther automated the verification process and reduced the need in procedural allocations.

[1]SystemVerilog and SystemC conjunct a design language and a hardware verification language under the same title. In this paper we refer to all of them as HVLs.

Still, constrained random verification poses a considerable challenge for both verification teams and tool providers. Having a complex verification environment reduces the ability of the verification engineer to predict the exact functional coverage of a random test, and makes it harder to debug, and direct the verification once something is not working as expected. Early stages of a verification project, for example, are often especially sensitive, when the coverage model is often not complete, while early validation of critical features is vastly important.

In the context of this paper we demonstrate our work using the IntelliGen constrained random generator and the *'e'* hardware verification language. IntelliGen is part of the Specman tool, which is Cadence's implementation of the *'e'* language. IntelliGen is designed to work with environments with hundreds of thousands of variables and constraints, and is the tool of choice for many of the most demanding verification environments in the industry. It provides state of the art debug capabilities and unmatched distribution control for verification environments of any scale.

Throughout this paper, we discuss two concepts of randomness in verification: *random distribution* and how it affects *coverage*, and *debug* of constraints.

One of the key elements in our work on IntelliGen is to provide out of the box distribution that reflects the user intentions. One of the weaknesses of other verification constraint languages is that constraints unwillingly and unintentionally drive the distribution into certain areas of the solutions space, while leaving other important scenarios uncovered. In Section II we demonstrate how random distribution helps or interferes with coverage of DUT functionality.

Another concept presented in this paper is debug of the constraint system. Encountering bugs is an integral part of verification and requires proper debug tools. Constraint solving, unlike traditional code debug, is a complex process happening under the hood of the stimuli generator, and traditionally it is a black box to the verification engineer. In Section III we discuss two topics in constraints debug. We demonstrate how randomness may become a hazard in the debug process, harming the reproducibility of tests, and the ways to control it. Afterward, we provide insights on how constraint solving can be debugged using an explanation based constraint debugger.

II. Random Distribution

The fundamental purpose of functional verification is to validate the correctness of all the DUT functions. The space of covered DUT functions is referred as *functional coverage*, or simply *coverage*. Modern HVLs allow users to define the coverage space, and using the Metric Driven Verification (MDV) methodology [5] to plan and monitor the tested functionality. [2]

Accordingly, the desired effect of using random generated stimuli in verification is full coverage [15]. To achieve this goal, many automatic tests should be created. These tests should vary, and the randomly generated values should be properly distributed over the solution space. It looks only natural that with random stimuli, enough random tests will easily cover the DUT functionality.

Unfortunately, this is not always the case in the verification industry. In order to understand why, understanding the properties of these "properly distributed" stimuli is required.

A. What constitutes a "good distribution"?

An intuitive answer to the question of "what is a good distribution?" might be a "uniform distribution" over the solution space. And indeed, previous discussions on distribution in functional verification emphasized the need for the results to distribute uniformly over the solution space. This approach is embedded in the SystemVerilog IEEE standard [3], as well as in the works of Dechter [10] and Koelman [11].

An important definition of good distribution is used by Naveh et. al. ([14]). There, the authors call for the tests to be distributed as uniformly as possible among all possible tests that conform to the constraint model, with the intent to reach a significantly different solution each time. While this definition may be used to justify a uniform distribution approach, uniform distribution over all possible solutions is actually not required, because two different solutions might still create very similar tests. Much more important than uniformly sampling the solution space, is sampling all "interesting" values of the different variables and giving equal priority to different scenarios rather than to different solutions.

Our claim is that the purpose of randomization is to cover as much functional scenarios as possible, and that uniform distribution over the solution space does not reflect this intention.

So what is a good distribution? Quoting Naveh [14], the definition is quite fuzzy: *"... we want the tests to be distributed as uniformly as possible among all possible tests that conform to the CSP. Essentially, we want to reach a significantly different solution each time we run the solver on the same CSP"*. The truth is that this is a difficult approximation problem. It's up to the tool providers to come up with as good as possible approximations, knowing they are not optimal and always can be improved.

[2]For our purpose, we refer to the tested functionality as covered regardless if it was monitored or not.

Building on the fact that the *'e'* standard does not require a uniform distribution, IntelliGen solver was designed to provide best out-of-the-box coverage for its users. Its mechanisms try to predict what is the functional intention of the constraints, and generate accordingly. By doing so, it provides quicker coverage closure, and reduces the need to adjust the verification environment as we discuss in the next subsection.

But even when verification engineers are not happy with their random distribution, they are not completely helpless. Users are able to compensate for the imperfection of the tools, and direct the random generation towards the proper distribution using constraint directives that the stimuli generators provide. Specifically, there are two main ways to control distribution: through distribution constraints and through order directives.

Distribution constraints, such as "select" in *'e'* and "dist" in SystemVerilog, are constraints which constrain a variable or an expression to be generated according to a given distribution.

On the other hand, order directives, such as "solve before" in SystemVerilog and "gen before" in previous Specman random generator, Pgen, are constraints which 'tell' the random generator to solve a specific variable or variables before other variables.

While order directives are the recommended way to work around distribution problems, according to books [19], [18] and user guides on SystemVerilog,

our point of view, based on the experience of Specman customers, is that to control the distribution is much better done through distribution constraints and order directive are constructs which suffer from the following flaws:

- The concept of 'order' is actually related to the technical solving process and is not intuitive to the user, from whose perspective all the variables of a problem are solved at once. Therefore, order directives require the user to understand the internals of the constraint solver.
- It is hard to define a clear and consistent semantic regarding the enforcement of several order directives and their impact on other variables of the problem.
- As order directives change the regular solving flow and may cause a severe performance slowdown or even undesired solving failures due to the solver's attempt to enforce them.

To summarize, order directives are used to work around tool and language limitations in cases where the distribution is insufficient for the user's needs. The alternative approach is that tests should be generated according to the default random generation of the tool, while specific scenarios are directed through the usage of distribution constraints.

III. Debugging

Verification is performed by generating random stimuli according to a given set of constraints. Coding errors in the constraints environment may cause conflicts in generation. Alternatively, a mistake may generate a stimulus that does not satisfy the DUT expectations. Such errors are an inherent part of the verification process and require debugging of the constraint solving.

Debug of constraint solving is different from the traditional concept of procedural code debug, with which most engineers are familiar. While there are different approaches to debug constraint solving, a profound requirement for debug is the ability to reproduce the failure. In Section III-A, we discuss the concept of reproducibility and its extensions to ease the debug process.

In addition, while debug of constraint solving is essential to fix errors in the constraint system, the constraint solver itself is a black-box. Debug tools, therefore, should cope with the task of explaining the solving process in terms familiar to their users. In Section III-B, we demonstrate how such principles applied in INTELLIGEN's Generation Debugger.

A. Reproducibility

Maintaining reproducibility of the solving process, which is random, is a non-trivial, but very necessary task. Any difference, even the smallest, may dramatically change the flow of the test, leading to irreproducibility. Such differences may derive from environment settings or be inherent in the solving technology.

While absolute reproducibility is essential, verification tests have an even more pressing requirement: *Code modifications that are irrelevant to the generation of a certain variable must not affect its generated values when using the same seed.*

While the emphasize on reproducibility is common to all HVLs, each verification language has slightly different capabilities. Despite its importance, among all tools known to us, random stability in a variable level is unique to SPECMAN -*'e'* . The rule we apply is that adding fields and constraints to a struct does not affect the generation of other fields that are not connected to the new fields by constraints.

B. Generation Debugger

As the DUTs grow and the constraint models become more complex, the need to debug mistakes in the constrained systems also grows, and debug takes a more substantial part in the verification process.

Although debug of constraints has been well studied [12], [17], [4], [8], [7], [6], it still poses a challenge for verification teams. The leading approaches in verification include adding/removing constraints, tracing of the solving process[12], or printing constraints participating in a conflict, but these are insufficient in complex scenarios. Another branch of tools includes complex visualization programs, designed for users who are familiar with constraint solving terms and algorithms.

While verification engineers are not experts in constraint solving, nor should they be expected to become ones, they remain unhappy with black box stimuli generation solutions, that they cannot debug. A constraint debug tool that provides the right level of abstraction to explain the behavior of the system as a sequence of individually understandable steps can be a big advantage over existing technologies, whose solution process is opaque.

In SPECMAN, the INTELLIGEN constraint solver is shipped with a generation debugger (GENDEBUGGER [16]). It is an explanation-based constraint debugger [9], whose design is based on over 10 years of experience in debugging constraint-solving issues.

In GENDEBUGGER, the solving process is depicted as a sequence of elementary solving steps such as: *assignment* of random value to variable, *reduction* of one or more variable domains due to constraints, and *value cancellation* due to a conflict between values in a constraint. Since each solving step is directly related to specific constraints, the user can be informed which constraints led to which change.

GENDEBUGGER also gives the user the ability to inquire any variable or constraint. For example, it allows easy tracing of reasons for variables range reductions and value omissions.

IV. SUMMARY

Constrained Random Generation is a powerful weapon in the verification jungle. We discussed two of it's less covered aspects.

This paper raised the question of proper distribution for random stimulus. We proposed that the widespread approach of uniform distribution is not the right choice, as it comes with a heavy burden on its users. Our claim is that a better approach is to approximate the user intent. In this way a better coverage is achieved, without forcing the user to interfere with the solving process.

In the Debug part, we showed that constraint debug is an essential requirement in verification. We demonstrated how to explain the constraint solving using simple steps of constraints and variables. Moreover, we saw that constraints and their random effects need to be controlled for the purpose of reproducibility.

We think that our industry has more room for improvements on the questions raised by this paper, and we hope that as verification becomes more and more sophisticated, we will get increasingly better answers for these problems.

REFERENCES

[1] Ieee standard for the functional verification language e. *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008)*, pages 1–495, 2011.

[2] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

[3] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, 2013.

[4] Manuel Carro and Manuel Hermenegildo. Tools for search-tree visualisation: The apt tool. In *Analysis and Visualization Tools for Constraint Programming*, pages 237–252. Springer, 2000.

[5] Hamilton B Carter and Shankar G Hemmady. *Metric Driven Design Verification: An Engineer's and Executive's Guide to First Pass Success*. Springer, 2007.

[6] Pierre Deransart, Manuel V Hermenegildo, and Jan Maluszynski. *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*. Number 1870. Springer, 2000.

[7] Grégoire Dooms, Pascal Van Hentenryck, and Laurent Michel. Model-driven visualizations of constraint-based local search. *Constraints*, 14(3):294–324, 2009.

[8] François Fages. Clpgui: a generic graphical user interface for constraint logic programming over finite domains. *arXiv preprint cs/0207048*, 2002.

[9] Mohammad Ghoniem, Narendra Jussien, and Jean-Daniel Fekete. Visexp: Visualizing constraint solver dynamics using explanations. In *FLAIRS Conference*, pages 263–268, 2004.

[10] Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In *Principles and Practice of Constraint Programming-CP 2006*, pages 711–715. Springer, 2006.

[11] Nathan Kitchen and Andreas Kuehlmann. Stimulus generation for constrained random simulation. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 258–265. IEEE Press, 2007.

[12] Micha Meier. Debugging constraint programs. In *Principles and Practice of Constraint ProgrammingŮCP'95*, pages 204–221. Springer, 1995.

[13] Reuven Naveh and Amit Metodi. Beyond feasibility: Cp usage in constrained-random functional hardware verification. In *Principles and Practice of Constraint Programming*, pages 823–831. Springer, 2013.

[14] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI magazine*, 28(3):13, 2007.

[15] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer, 2004.

[16] Aaron Rich, Giora Alexandron, and Reuven Naveh. An explanation-based constraint debugger. In *Hardware and Software: Verification and Testing*, pages 52–56. Springer, 2011.

[17] Christian Schulte. Oz explorer: A visual constraint programming tool. In *Programming Languages: Implementations, Logics, and Programs*, pages 477–478. Springer, 1996.

[18] Chris Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2008.

[19] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-based verification*. Springer, 2006.