Quality Improvement Methods for System-level Stimuli Generation

Roy Emek Itai Jaeger Yoav Katz Yehuda Naveh
IBM Research Laboratory in Haifa
Haifa University Campus, Haifa 31905, Israel
{emek, itaij, katz, naveh}@il.ibm.com

Abstract

Functional verification of systems is aimed at validating the integration of previously verified components. It deals with complex designs, and invariably suffers from scarce resources. We present a set of methods, collectively known as testing knowledge*, aimed at increasing the quality of automatically generated system-level test-cases. Testing knowledge reduces the time and effort required to achieve high coverage of the verified design.*

1 Introduction

Functional verification is acknowledged as the bottleneck in the hardware design cycle. Simulation is the main functional verification vehicle for large designs, and therefore stimuli generation plays a central role in this field. In this paper, we present a set of methods aimed at improving the quality of automatically generated test-cases for system-level verification. We use the term *Testing Knowledge* (TK) to refer to such methods that are generic, and that are useful for a a variety of hardware systems.

Coverage is our prime measurement for stimuli quality. Ultimately, higher coverage means better chances of exposing a bug. TK targets areas that are, because of their inherent complexity, bug prone, and consequently increases the coverage for many typical coverage models.

The idea of TK was previously implemented in test-case generators oriented at the verification of processors [1, 3, 4]. We are not aware, however, of a significant effort to develop a set of TK mechanisms for the system level. General purpose verification environments, such as *Specman*, *Vera*, and *TestBuilder* provide means for implementing TK. This is typically done in the form of non-mandatory (or 'soft') constraints. However, the 'soft constraint' mechanism alone does not contain any semantic knowledge of the verified design.

The contribution of this paper is threefold: first, we introduce the concept of TK as a central element of system verification methodology. Second, we present in a unified manner a set of TK mechanisms we found useful. And third, The *actor choice pattern* TK mechanism presented towards the end of section 3 is novel.

2 Motivation: TK and System Verification

Testing Knowledge The stimuli generation layer of a verification environment can be roughly partitioned to two: knowledge about the specification of the verified system, and knowledge about the way it should be verified. The former defines the set of *valid* tests that can be injected to the Design Under Verification (DUV). The latter, which implements the verification plan, is a mixture of two types of information: the first determines the main specification of the tests to be generated. It is expressed through 'test-templates' of the form "100 transactions of type *A*, and 50 transactions of type *B*". The second aims at increasing the quality of *all* the generated test-cases, and contains statements of the type "by default, the ratio between valid and erroneous transactions should be 4:1"—this is what we refer to as TK.

Given a test-template as input, a random test-case generator produces a set of distinct test-cases, all satisfying the requirements specified in the test-template. In most cases, generating high-quality, complex scenarios, require complex test templates. Developing these test-templates is a costly effort: this creates an incentive to move some of the burden from the test-templates to the testing-knowledge. By doing so, we allow all the generated tests, and not just a small fraction of them, to benefit from the intelligence implemented in a complex test-template.

In principal, random stimuli generation environments produce test-cases which are uniform over the domain of valid test-cases. TK biases the random selection of test-cases towards areas that have higher chances of increasing coverage and exposing hardware bugs.

System Verification A 'system' is a set of *components* connected using some sort of interconnect, capable of performing a given set of *transactions*. Components may include processors and other processing elements, memories, bridges, DMA engines, etc. An example of a transaction is a Memory Mapped I/O (MMIO).

System verification deals, in essence, with the validation of the integration of several previously verified components (cores, IP), and it copes with three major challenges: (a) large designs; (b) intricate system specifications, and (c) limited resources, and specifically short schedules.

A main direction considered as a possible solution to these challenges is reuse. Checkers (e.g., assertions) written for lower level interfaces and components can be reused at the system level. The same is true for Bus Functional Models (BFMs): a BFM written for the verification of a PCI/X core, for example, can be reused to inject input to the system as a whole.

TK embodies another type of reuse. Here, we gain knowledge about prone-to-bug areas in one system (or at the unit-level), and reuse this knowledge for the verification of others. An even stronger form of reuse may be achieved if the stimuli generator itself allows the integration of general TK for a specific domain (e.g., systems or SoCs).

Through these forms of reuse, and by cutting down the number of required test-templates, TK reduces the cost of implementing a test plan. In many cases, a complex coverage model cannot be covered by pure random generation. The verification team then faces the need to develop a large set of highly constrained test-template to attack the different 'corners' of this model. The idea of TK offers an alternative: developing a single TK mechanism instead of multiple test templates. This alternative has several advantages: a single TK mechanism is easier to maintain, can often be ported more easily to similar or follow-on designs, and most importantly, it coexists with other TK mechanisms in the same stimuli. The latter means that the combination of several TK mechanisms, aimed at different coverage models, typically results in stimuli that would not be generated by the multiple test-template approach.

3 Testing Knowledge Examples

Resource Contention System-level bugs are often related to scenarios in which several consumers contend for the usage of a single resource. It is therefore beneficial to create a TK mechanism that causes multiple agents in the system to access a single resource, preferably during a limited time interval.

One way to cause contention for resources is though an *address collision* mechanism. In many systems the concept of a system-address is the main way to identify resources. In these systems, it is typically beneficial to cause multiple accesses to the same addresses. This is especially true in systems with a complex cache hierarchy: Completely random accesses would practically only cause cache misses, while address collisions would also cause cache hits, cast-outs, and potentially various cache related corner cases.

An address collision mechanism can be implemented using a queue that contains pairs of the form $(address, length)$ ($length$ is the number of bytes accesses by a transaction). Following every transaction that uses an address, we push a new pair into the queue, and potentially remove a pair if the queue is full. With some predefined probability, the address and length properties of a new transaction are generated to form a collision with one of the pairs in the queue.

Note that while stimuli that aims at resource contention is not a new idea in the context of verification, the presentation here focuses on a reusable and abstract mechanism for creating such stimuli.

TK for Address Translation Most modern systems support several address spaces, and implement address translation mechanisms between these spaces. Examples for such translation mechanisms can be found in processors, in bus bridges, and in high-end interconnect such as InfiniBand.

One type of TK for address translation, which is a natural follow-on to resource contention, is reusing the same translation table entry multiple times, or invalidating a translation table entry that resides in a cache (Translation Look-aside Buffer).

A different type of TK related to address translation deals with the *placement* of accesses relative to virtual pages or segments. Most translation mechanisms partition the address space into a set of pages, or segments. The placement TK is aimed to better stimulate address translation related hardware. With some probability, this mechanism places an access in a manner that would cause one of three placement events (see Figure 1): (a) *Vicinity*: an access is placed near the beginning of a page (segment), or near its end; (b) *Boundary*: an access is placed at the very beginning of a page (segment), or at the very end of it; And (c) *Crossing*: an access starts in a certain page (segment), and ends in another.

Each of these three events may activate a control-logic that relates to a certain corner-case in the DUV. For a specific example, see Section 4.

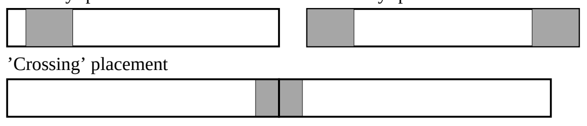

Figure 1. Placement testing knowledge

Topology and Configuration TK *actor choice pattern*

	P0	P1	P2	P3	P4	P5	P6	P7
Mem0								
Mem1		$\frac{1}{8}$						
Mem2	$\frac{1}{8}$			$\frac{1}{4}$		$\frac{1}{8}$		
Mem3							$\frac{1}{4}$	$\frac{1}{8}$

Table 1. Non-uniform probability matrix

(ACP) is a TK mechanisms aimed at systems with multiple instances of the same component types.

For example, consider a system with eight processing elements (PEs, e.g., processors) and four memories, in which each PE can access each memory. The probability of generating a transaction between any pair of a PE and a memory in this system is $\frac{1}{4\times8} = \frac{1}{32}$. ACP aims at generating more 'interesting' patters for transactions that involve several components (actors). The mechanism creates a non-uniform distribution function, and uses this function when choosing actors for newly generated transactions. The distribution function for a transaction that involves n actors can be represented as an n-dimensional matrix. An example for a non-uniform distribution matrix is shown in Table 1.

Sparse matrices like the one shown in Table 1 cause the traffic in a certain test-case to concentrate in only a fraction of the interconnect. The basic TK behind ACP is that for each test-case, a sparse probability matrix is created. This probability matrix can be completely random, or, it can take into account issues like the topology of the DUV, or its configuration. Over a large number of test-cases, we will then form different patterns of stress on different parts of the interconnect. The effectiveness and reusability of ACP is driven from the fact that it increases the quality of interconnect testing, with almost no knowledge of the structure of the system, and through an easy-to-use and easy-to-implement, mechanism.

4 Usage Experience

Table 2 compares the coverage achieved by two comparable tools, used simultaneously for the verification of the same design—a high-end mutii-processor system in IBM—and implementing the same verification plan. The first line compares the number of simulation cycles consumed for tests generated by each of the two tools. X-Gen [2], which contains TK, consumed about five times less simulation cycles than a previous tool, that does not. Because of the use of TK, the number of test-templates (shown in the second line) required to implement the verification plan was much smaller in X-Gen than in the previous tool.

The rest of the table shows the functional coverage achieved by the two tools on a set of coverage models. These are cross-product models, that relate to events on several elements of the system's interconnect and to their relationships in time (e.g. $event1$ x $event2$ x $cycle-difference$). X-Gen reached better coverage on all of the coverage models. Coverage results for several other models, not shown here, are similar. In addition, X-Gen exposed several bugs which can be directly tracked to its usage of TK. We do not describe these bugs here due to lack of space.

Category	**X-Gen** With TK	**Previous tool** No TK
Simulation cycles	x1	x4.79
No. of test-templates	737	7168
Coverage model 1	40.57%	37.10%
Coverage model 2	43.84%	26.88%
Coverage model 3	74.28%	68.30%
Coverage model 4	61.14%	59.17%

Table 2. Usage experience - coverage

5 Conclusions

We defined testing knowledge as a general term that relates to methods aimed to increase the quality of automatically generated stimuli. We then explained why the idea of TK is particularly suitable for system-level verification: it improves the quality of verification, with a relatively low cost, in a place were time and resources are scarce, and in which verification is a significant challenge. We stress the concept of TK as a basic, reusable element of the verification methodology.

We presented a set of TK mechanisms oriented at the system level, and compared the coverage gained by two environments used for stimuli generation for the same design: one using TK, and another that does not.

The TK mechanisms and methods described in this paper are implemented in X-Gen [2], a random test-case generator used for the verification of several systems in IBM.

References

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *32nd Design Automation Conference (DAC95)*, pages 279 – 285, 1995.

[2] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-Gen: A random test-case generator for systems and SoCs. In *IEEE International High Level Design Validation and Test Workshop*, pages 145–150, Cannes, France, October 2002.

[3] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test program generator. In *Design Automation & Test in Europe (DATE99)*, pages 434–441, Munich, March 1999.

[4] Obsidian software. *Raven(TM) Software User's Manual*, May 2001. http://www.obsidiansoftware.com/files/manual.pdf.