X-GEN: A RANDOM TEST-CASE GENERATOR FOR SYSTEMS AND SOCS

Roy Emek, Itai Jaeger, Yehuda Naveh, Gadi Bergman, Guy Aloni,
Yoav Katz, Monica Farkash, Igor Dozoretz, Alex Goldin
{emek,itaij,naveh,gadib,guya,katz,farkas,igord,alexgo}@il.ibm.com
IBM Research Laboratory in Haifa

Abstract

We present X-Gen, a model-based test-case generator designed for systems and systems on a chip (SoCs). X-Gen provides a framework and a set of building blocks for system-level test-case generation. At the core of this framework lies a system model, which consists of component types, their configuration, and the interactions between them. Building blocks include commonly used concepts such as memories, registers, and address translation mechanisms. Once a system is modeled, X-Gen provides a rich language for describing test cases. Through this language, users can specify requests that cover the full spectrum between highly directed tests to completely random ones. X-Gen is currently in preliminary use at IBM for the verification of two different designs—a high-end multi-processor server and a state-of-the-art SoC.

Introduction

During the last few years, complex hardware designs have shifted from custom ASICs towards System on a Chip (SoC) based designs, which include ready made components ('cores'). The verification of such systems requires new tools and methodologies that are up to the new challenges raised by the characteristics of systems and SoCs. At the heart of these challenges stands the requirement to verify the integration of several previously designed components, in a relatively short time.

Often, a single organization develops a family of similar systems during a short period of time, where a number of systems are developed in parallel or as a follow-on to one another. Therefore, the verification environment used for these systems needs to be flexible enough to be applicable for a follow-on or for a similar system. To achieve this, parts of the verification environment should be reusable. These requirements from the verification environment also arise when the same cores are used in several systems.

As systems become more complex, their interfaces become increasingly intricate. As a result, test-case generation and checking mechanisms [3] have to be expressed at a higher level of abstraction. This forms a gap between the high-level functional specification of the system and the concrete, low-level interfaces of its components. For example, consider the system-level transactions that form the interface of the system as a whole versus assembly language instructions for a processor core.

The verification challenges reflected in these observations characterize a wide variety of systems, ranging from complex SoCs to full-scale, multi-processor (MP) systems.

Model based test generation scheme [2] proved to be highly effective in other domains. In this scheme, a generator is partitioned into a generic, system independent engine, and a model which describes the verified design. This approach has a number of advantages. First, test-case generation for new designs becomes an easier task. Second, there is a structured, well defined way to integrate new knowledge about the verified design into the tool. And third, a generic test-case generator can include generic knowledge that applies to many designs.

In this paper, we present X-Gen: a model-based random test-case generator aimed at coping with the challenges discussed above and generating potent test-benches for a wide variety of systems. A high-level depiction of X-Gen is shown in Figure 1. As input, X-Gen accepts a model of the design under test (DUT), containing component types, the connections between them and the interactions they perform. X-Gen also accepts a set of user-defined requests (a *request file*). For a given request file, it generates a set of different test-cases, each of which realizes the request file.

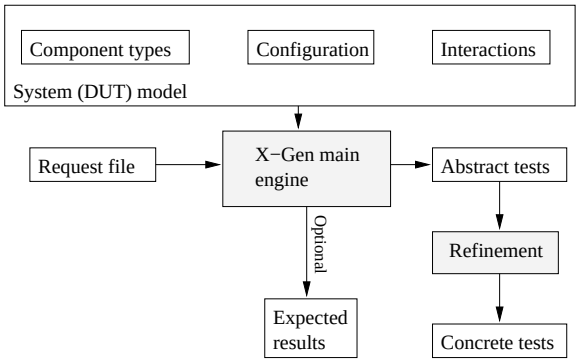

Figure 1. X-Gen: High-level Description

While X-Gen initially generates abstract system-level tests, these tests are refined in a later stage into concrete tests in a form suitable for simulation (e.g., on an RTL simulator).

This enables the main body of X-Gen to ignore low-level details which are irrelevant to the system-level verification process.

X-Gen provides a framework and a set of building blocks for system and SoC modeling. Once a system is modeled, requests are expressed through a rich request language, supported by a GUI. X-Gen also provides extensive internal testing knowledge (e.g., in a system that contains caches, it is typically beneficial to access the same address more than once).

X-Gen's uniqueness, when compared to general purpose verification tools or languages [5, 6], lies in the fact that both the general framework and the building blocks it provides are oriented at systems and SoCs (see last section for details).

It should be noted that X-Gen is dedicated to stimuli generation. In most cases, it is used in conjunction with some separate, on-the-fly checking mechanism. X-Gen may also generate, when requested, end-of-test expected results [1]. Using X-Gen in this mode prevents it from generating certain classes of tests.

The rest of this paper is structured as follows: the next section describes the system modeling framework. We then describe the request file language, the generation scheme, and the way X-Gen is currently used in IBM. Finally, we compare X-Gen to general purpose verification tools and languages.

System Model

X-Gen's modeling framework contains three basic concepts: the *component types* of the system, the *interactions* (or transactions) between them and the system's *configuration* (See figure 1). The modeling of component types and interactions is done through a special-purpose graphical editor, in which forms are filled to describe the various aspects of the system. Typically, several similar systems are modeled through the use of several configurations, based on a single collection of component types and interactions.

Component Types

Component types describe the types of hardware components used in the system being verified. Examples include a processor core, a bus bridge, and a bus functional model (BFM). Rather than describing components, X-Gen's modeling language describes *types* of components. This enables the use of multiple instances of the same component type in a single system (as in the case of an MP system), or the use of same component type in several different systems.

A component type contains three main aspects: *ports* through which it connects to other components, *internal state*, and *behavior*. Each component type contains a set of ports. For example, a simple Ethernet to ATM bridge has two ports. A port in X-Gen consists of the set of *properties* that define its interface to neighboring components. Examples of properties may be address, data, access-size, L2 cache line, and interrupt priority.

Internal state is the set of resources that reside in a component and affect its behavior. Examples of resources include registers, memory, etc. The contents of these resources will generally change during the execution of the test. As part of the generation process, the corresponding state of each resource is updated according to the interactions in which the component participates. The generated test case includes the initial values generated for these resources. When X-Gen is used in 'expected results' mode, their end-of-test expected values are calculated as well.

A component behavior is modeled via constraints that describe the relationships between properties on its ports or between these properties and its internal state. A processor, for example, may constrain the `address` and `access size` properties and require that all 8-byte accesses to memory are 4-byte aligned. A more complex constraint may relate to the PCI-to-fabric address translation performed by a bus bridge. This constraint will restrict the PCI address property and the fabric address property, according to a system of memory-based translation tables pointed to by a set of registers.

X-Gen's modeling environment provides building blocks for both internal states and constraints. Internal state building blocks include registers, memory, etc. Modeling aids for constructing constraints include a declarative language that contains arithmetic, bit-wise, and logic operators [4]. In addition, X-Gen provides building blocks for modeling constraints that represent translation tables, system-wide consistency rules, and other common patterns of behavior. Constraints that cannot be defined using any of the above, are written as C++ procedures.

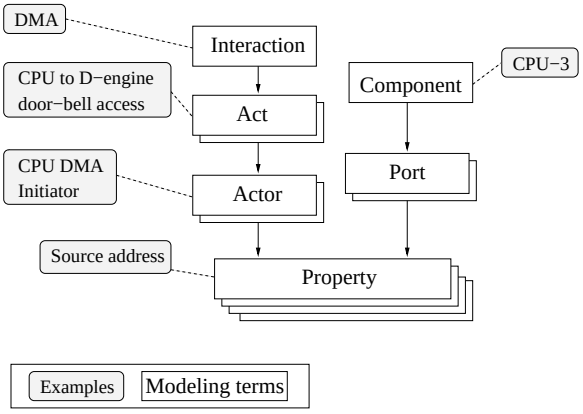

Figure 2. Interaction Model Overview

Configuration

Given a set of component types users construct system configurations. First the system's topology is defined by instantiating components (e.g., `cpu-1`, ...,`cpu-8` of type `Processor`) and depicting the connections between them. Then static characteristics of these components (e.g., the address space for a memory component) are being set to complete the configuration's definition. A request file writer may refer to the components defined in the configuration,

for example, by requesting X-Gen to generate a set of interactions in which `PCI-01` participates.

In many cases, the same set of component types is used in different systems. From X-Gen's point of view, 8-way and 2-way multi-processor servers from the same family differ only in their configuration: their sets of interactions and component types are identical. Moreover, a single 8-way system may have several configurations (e.g., different memory maps).

Interactions

Interactions describe the way components interact with each other, according to the specification of the system.

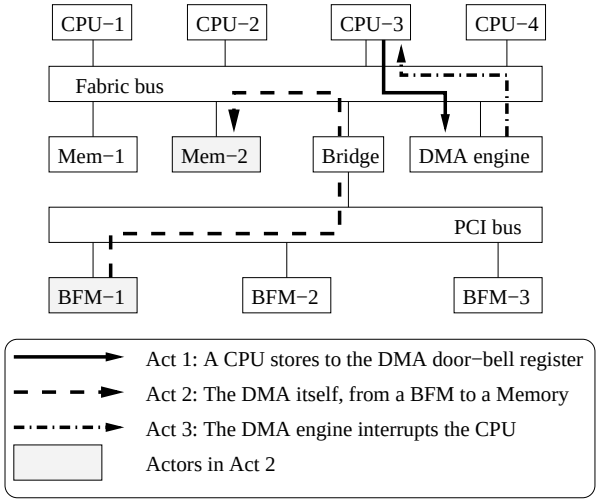

Figure 3. Interaction Example: DMA

Figure 2 shows X-Gen's interaction view, and Figure 3 shows an example of an interaction. The interaction model consists of one or more acts that embody the activity that takes place during the interaction. Several components may participate in each act.

Actors relate to the principal components participating in an act. Different instances of component types, or components of different types, can serve as the same actor in different occurrences of an interaction during a test. Note that not all components along the path of an act need to be defined as actors (Components not defined as actors are calculated automatically by X-Gen).

The relationship between interactions and components is expressed through properties which are defined for each actor in an interaction. Any component that plays the role of an actor in an act must contain the actor's properties on one of its ports (see figure 2).

An interaction model also includes constraints over actor properties, actor identities, or a combination of these.

The interaction example of Figure 3 shows a DMA. In this interaction, a processor performs a series of store instructions to specific addresses within a DMA engine, notifying it of the source address, the target address, the transfer size, and possibly other characteristics of the DMA operation. As a result, the DMA engine transfers the required block of memory, and sends an interrupt to the originating processor. X-Gen's model of such an interaction would typically contain three acts: (a) the series of stores performed by the processor; (b) the DMA transfer itself, and (c) the interrupt. The model would also include constraints both on the identity of the participating components (e.g., the DMA engine is a component of type 'D-engine', the originating processor and the interrupted processor are the same component) and on other properties of the interaction (e.g., the size of the DMA must be a multiple of 128 bytes, the door-bell store must access an address defined by a base register in the DMA engine).

Testing Knowledge

Testing knowledge is expert knowledge incorporated into the system model in order to stress bug-prone areas; it does not require explicit input from the request-file writer.

Examples of testing knowledge building blocks provided by X-Gen include: (1) a collision mechanism that biases the test-case towards the reuse of certain system resources (e.g., many accesses to the same cache-line); (2) a placement mechanism that biases memory accesses towards events such as cross-page (an access that begins in one page and ends in another) or segment boundary; and (3) weighted random choice of the values for a certain property.

Request Language

A user-defined request file serves as a template for a set of tests; it describes the characteristics required for tests aimed at specific verification goals. Through request files, users can provide a full or partial description of a required scenario, and leave all the surrounding parameters unspecified. This enables X-Gen to hit a targeted event in a practically unlimited number of different ways. Thus, when a targeted scenario is defined by the user in loose terms, X-Gen, through its randomness and testing knowledge may generate the actual bug-revealing test-case around the loosely defined scenario.

Interactions are the basic building blocks of the request file language. Users may constrain different interaction attributes; among these are the identity of the actors that participate in the interaction and their properties. Even though actors represent only some of the components that participate in an interaction, users may use the *participant* construct to require that a specific component participate in an interaction, without explicitly specifying the identity of any of the actors. For example, in a sequence of interactions consisting of a single act between two actors, a BFM and a memory component, users may stress a certain bus-bridge in the system by directly requesting that this bus-bridge participate in all the interactions. This way, the BFM and memory actors may randomly vary from one interaction to another, while the bus-bridge remains the same.

The request file language contains several high-level statements used to group interactions or other high-level

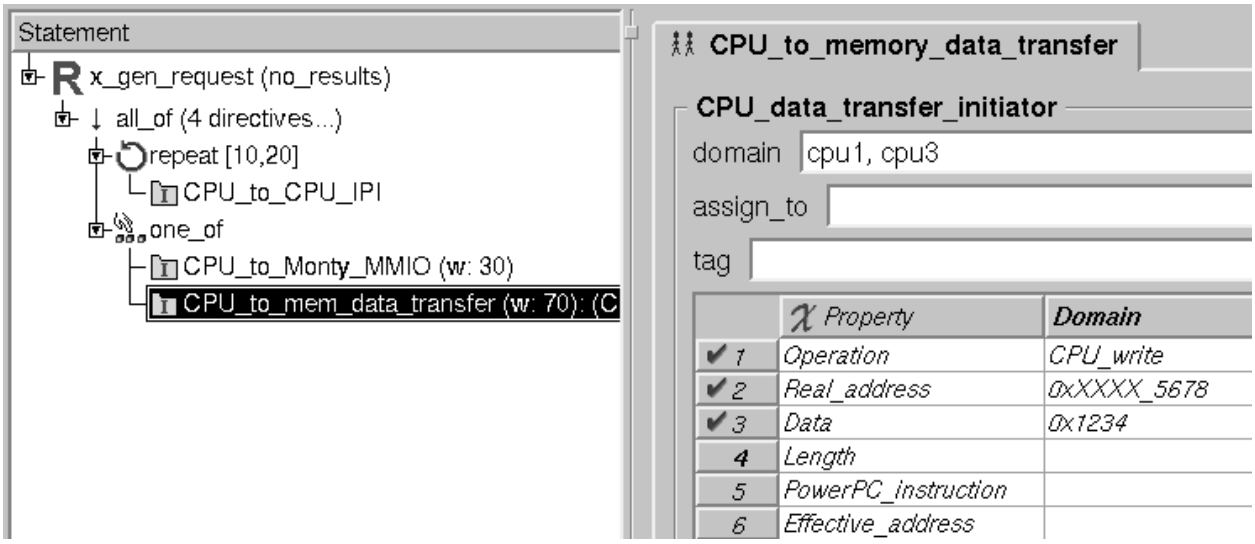

Figure 4. X-Gen Request File (GUI Screen Shot)

statements, (see Figure 4): (1) an `all_of` statement generates all its sub-statements; (2) a `one_of` statement provides a weighted choice between its sub-statements; and (3) a `repeat` statement generates multiple instances of its sub-statements. As these high level statements can be used to group single interactions, or to group other high-level statements, a request file can be viewed as a tree with interactions at its leaves and high-level statements at its intermediate nodes.

Users can influence any single interaction in the request file by directly specifying actors and properties. To bias a large set of interactions, users attach *directives* to high level statements. A directive affects all the interactions contained in the sub-tree of the statement to which it is attached. When attached to the root of the request file, directives provide a compact way to bias all the interactions in the test. Moreover, they provide a way to establish different characteristics in different parts of the request file: For example, 90% of the DMAs that pass through bus-bridge A are 'short,' while 90% of the DMAs that pass through bus-bridge B are 'long.' X-Gen provides a large number of directives, that represent testing knowledge, which affects different aspects of the system behavior. Directives are defined as part of the system model.

X-Gen's request language provides, in addition to the concepts described above, a programming-language-like aspect that enables users to depict very specific scenarios. This aspect includes variables, a rich expression language, variable assignments, and the usage of variables to constrain the selection of values for actors and properties. It may be used to form practically any type of complex relationship between a set of interactions.

Normally, interactions generated by X-Gen may be executed in any order by the verified system, not necessarily in the order in which they were generated. For example, X-Gen may generate an interaction for processor P_1 and then an interaction for another processor P_2. However, because the two processors run in parallel the interaction of P_2 may be executed first. X-Gen provides means to explicitly control the execution order. This is done by specifying one of the three order-control constructs `sequence`, `mutual-exclusion`, or `rendezvous` in the scope of an `all_of` or a `repeat` statement. When this is done, all interactions (or high-level statements) in this scope are executed in a sequence, in mutual-exclusion from each other, or in rendezvous with each other, respectively. The descriptions of the mechanisms with which these types of scheduling scenarios are enforced are part of the system's model.

To summarize this section, X-Gen's request language embodies a dual effort methodology (see Figure 5), where a single test generated by X-Gen contains both a user-directed scenario, targeted at a specific logical area of the verified design, and 'intelligent' background noise, stimulating the rest of the system in a non-trivial random manner. Variables and expressions provide the ability to specify the former, while directives and the use of testing knowledge enable the latter.

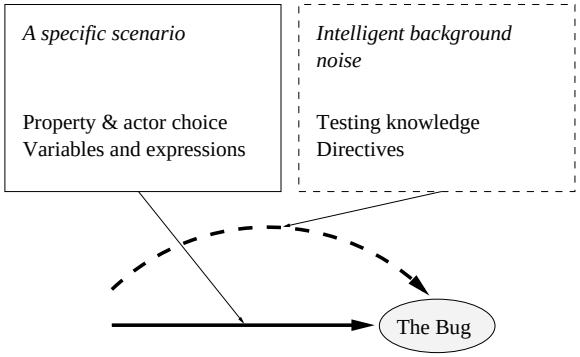

Figure 5. Dual Effort Methodology

Generation scheme

For a given request file, X-Gen's generation process can be divided into two layers: traversing the high-level statement tree and generating the interactions at its leaves. The statement tree traversal is done in a hierarchical manner, where each statement is responsible for the generation of all its sub-statements. An `all_of` statement, for example, would generate all its sub-statements, while a `one_of` statement would randomly pick one of its sub-statements, and then generate it.

For a given interaction, X-Gen randomly chooses its actors and assigns random values to the interaction's properties. These selections must comply with the constraints defined on the interaction and the constraints imposed by the components that participate in the interaction. The request file may further restrict the choice of actors and property values; the selections made by X-Gen must also satisfy these restrictions.

X-Gen generates an interaction by constructing and solving two constraint networks: one represents actor selection and the other represents property selection. The solution of the former network is used as the basis for the construction of the latter. After identifying the actors, X-Gen calculates the path between the actors of every act, and constructs a constraint network that contains the properties of all the participating components along the paths. Both networks are solved using a variant of the well-known *maintaining arc consistency* (MAC) algorithm [4, 7].

The random choice of actors and property assignments is influenced by testing knowledge that was modeled for the interaction or for the components participating in the interaction. Testing knowledge is incorporated into the constraint networks as soft constraints that are activated with some preset probability. These kind of constraints influence the resulting test when they do not contradict any architectural requirement or explicit user request. Users may use directives in the request file to set the probability with which each such constraint is activated, and to determine its exact behavior.

The solutions of the actor and property constraint networks for all the interactions in the request file form the abstract test generated by X-Gen.

Refinement

Often, some of the low-level details necessary in a test are irrelevant to the system-level verification process. Such a detail in the case of an MP system, for example, is the identity of a register used by a CPU in a `store` instruction (e.g., R_3 or R_8). To decouple the relevant details from the low-level ones, X-Gen uses a two phase generation scheme: first, an abstract test is generated. The lion's share of generation effort is invested in this stage. Then, the abstract test goes through a simple stage of refinement to produce the actual concrete test, suitable for simulation.

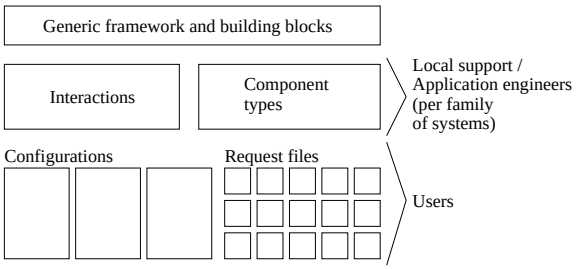

Figure 6. X-Gen Usage Scheme

The refinement process does not contain any significant random decisions, and cannot be controlled by users through the request file. As it relies on the system model, the refinement engine can be used for different systems.

For an example of the refinement process consider the DMA interaction discussed previously. In this case, the abstract test may include the DMA source address, target address, and transfer-size, as well as the door-bell address (or addresses) to be accessed by the processor to kick-of the DMA. This information would then be expanded during the refinement stage to include a set of `store` instructions to the door-bell register for the processor, possibly followed by a loop waiting for an interrupt.

Usage

X-Gen's usage scheme (Figure 6) consists of two different roles: (1) application engineers, or local support, model the interactions and component types of the verified system, using X-Gen's modeling environment and building blocks; and (2) users write configuration files and request files. The rationale behind this usage scheme is the pace at which different aspects of the test-case generation process change, and their correlation with a specific verification task: the generic parts of X-Gen are relatively stable. Interaction and component types, as well as system-specific testing knowledge, evolve and need to be modified more often. Several configurations typically exist for a given set of interactions and component types. Finally, request files may be created and modified on a daily basis, and aim at a specific verification goal.

X-Gen is currently in preliminary use by two IBM development organizations for the verification of both a high-end MP server and a state-of-the-art, highly complex, MP SoC. X-Gen is part of a verification environment that includes hundreds of request files, and in which a large number of tests are generated and simulated each day. Tests generated by X-Gen have already revealed bugs that escaped other generators.

Comparison with Other Tools

X-Gen is different from general purpose verification tools, such as Specman [5] (Verisity) or Vera [6] (Synop-

sys). It was designed and built with systems and SoCs in mind: the concepts it relies on (components, interactions, etc.) are specifically targeted at systems. This way, the additional level of abstraction between the atomic property and the entire system is inherent to X-Gen.

In contrast, tools which do not have this natural ability to deal with the different abstraction levels are at a disadvantage when used for system verification. With such tools, a preliminary necessary step before getting into the verification process is creating precisely this additional abstraction level.

Moreover, because X-Gen is aware of the system-level abstraction, it can provide extensive aids for modeling related testing knowledge.

There are other aspects in which X-Gen is different from general purpose verification tools. X-Gen clearly separates between the system model and the test description (request file). This separation provides for a better verification methodology, as these different aspects of the verification process are best conceptualized in different languages. On the other hand, a single-language methodology reduces overhead and is cheaper to use in small, homogeneous teams. Larger designs, such as complex systems or SoCs, require verification teams large enough to be split into modeling and test case writing teams—thus permitting the use of the language suitable for each part of the process.

X-Gen's refinement process allows the body of the system model to ignore parts of the generation process which are irrelevant to the system-level verification process. General purpose verification environments do not support this.

As opposed to Vera and Specman, The X-Gen interfaces for modeling the system, and most of its request file language, are not a programming language. This is demonstrated by the fact that system modeling and request file writing is performed under a graphical environment, rather than in a text editor.

Vera and Specman's capabilities exceed generation. Both tools provide aids for coverage and checking, while X-Gen concentrates on test-case generation.

Summary

In this paper, we described X-Gen, a model-based test-case generator targeted at systems and SoCs. X-Gen's modeling platform includes component types, configuration, and interactions. X-Gen provides a set of building blocks and modeling aids to ease the creation of new components, interactions, and system-level testing knowledge. X-Gen's request language enables the generation of tests across the full spectrum, from completely random through testing-knowledge biased random, to highly directed. Different logical sections in the test may be created with different degrees of this freedom.

References

[1] Allon Adir and Gil Shurek. Generating concurrent test-programs with collisions for multi-processor verification. In *IEEE International High Level Design Validation and Test Workshop*, Cannes, France, October 2002.

[2] 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.

[3] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[4] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, August 2002.

[5] Verisity Design. Spec-based verification. http://www.-verisity.com/resources/whitepaper/specbased.html.

[6] Faisal Haque, Jonathan Michelson, and Khizar Khan. *The Art of Verification with Vera*. Verification Central, 2001.

[7] Alan Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99 – 118, 1977.