Constraint-based Random Stimuli Generation for Hardware Verification

Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek
IBM Haifa Research Lab
Haifa University Campus
Haifa 31905, Israel
{naveh,michalr,itaij,katz,vinov,marcus,shurek}@il.ibm.com

Abstract

We report on random stimuli generation for hardware verification in IBM as a major application of various artificial intelligence technologies, including knowledge representation, expert systems, and constraint satisfaction. The application has been developed for almost a decade, with huge payoffs. Research and development around this application is still thriving, as we continue to cope with the ever-increasing complexity of modern hardware systems and demanding business environments.

Introduction

IBM estimates that it has saved more than $100 million during the last decade in direct development costs and reduced time-to-market by using artificial intelligence (AI) technology for the verification of its processors and systems. The technology is used to generate tests, or stimuli, for simulating hardware designs prior to their casting in silicon. It aims to ensure that the hardware implementation conforms to the specification before starting the expensive fabrication of silicon. The technology reduces the number of bugs "escaping" into silicon, allowing to cast less silicon prototypes. At the same time, the technology reduces the size of the verification teams and the duration of their work.

The current version of the technology includes an ontology for describing the functional model and capturing verification expertise, as well as a constraint satisfaction problem (CSP) solver. The ontology allows the description, mostly in a declarative way, of the hardware's functionality and knowledge about its testing. A separate, special-purpose language is used to define verification scenarios. The system translates the functional model, expert knowledge, and verification scenarios into constraints that are solved by a dedicated engine. The engine adapts a maintain-arc-consistency scheme to the special needs of stimuli generation.

An early version of the technology was presented to the AI community a decade ago (Lichtenstein, Malka, & Aharon 1994). AI techniques were only rudimentally implemented then. Ten or so years of experience result in a much more sophisticated ontology, a totally new and dramatically stronger solver and great success in deployment. The current technology has become the standard in processor verification within IBM. It has been used in the development of numerous IBM Power processors, i/p-series servers, Cell and Microsoft's Xbox^{TM} core processors and systems, and various processors of Apple computers. The system has become a repository of extensive processor verification knowledge across multiple IBM labs and many processor architectures and implementations. It allows comprehensive reuse of knowledge, rapid reaction to changes, and gradual reduction in the need for human experts. This paper describes our experiences with the technology during the last few years.

Problem Description

Functional Verification

Functional verification is the process of ensuring the conformance of a logic design to its specification. Roughly, a hardware logic design is a stage in the implementation of the physical hardware component. In this stage, code written in a hardware description language (HDL) describes the structure of the component (hierarchy, modules, pin-interface), the allocation of state variables, and the component's behavior down to the binary function driving each electronic signal. This HDL code can be simulated using commercial software tools, and can be automatically synthesized onto silicon. Functional verification is widely recognized as the bottleneck of the hardware design cycle, and becomes especially challenging with the growing demand for greater performance and faster time-to-market.

Simulation-based Functional Verification

In current industrial practice, simulation-based verification techniques (Bergeron 2000), as opposed to formal methods, play the major role in the functional verification of hardware designs. These techniques verify the design's actual behavior by simulating the HDL description of the design and driving stimuli into this simulation, as illustrated in Fig. 1. The behavior of the simulated design is then verified by comparing it to the expected behavior implied by the specification.

The current practice for functional verification of hardware systems starts with the definition of a verification plan that enumerates behaviors to be checked and identifies ma-

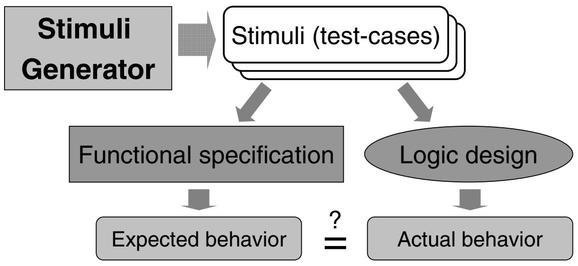


Figure 1: Simulation-based verification

jor risk areas. The verification plan also devises scenarios that isolate errors in these areas. Verification engineers then map the verification plan scenarios into concrete tests that are supposed to cover all those scenarios. This translation of functional scenarios into tests is non-trivial because the verification plan is typically formulated in natural language with a high level of abstraction, while the actual test must be precise and detailed enough to be executed in simulation.

Random Stimuli Generation and Test Templates

Because of the elusive nature of hardware bugs and the amount of stimuli needed to cover the scenarios specified in the verification plan, directed random stimuli generation (Aharon *et al.* 1995) has become the verification approach of choice. Here, many tests are generated automatically by a pseudo-random stimuli generator. The input to these generators is a user request, outlining the scenarios that should occur when the test is executed at simulation. For example, in Fig. 2(a) we see that the verification engineer may specify that all *Add* instructions generated in some batch of tests have at least one operand larger than 9999. This outline acts as a template from which the stimuli generator creates different tests by filling all missing detail with valid random values.

Validity and Expert Knowledge Rules

User requests are only one source of rules with which the test must comply. In addition, generated tests must be valid, meaning that the code in test must comply with the rules specified by the hardware architecture. The architecture defines the syntax and semantics of the programming instructions, including rules on how instructions may be combined to compose a valid program. Examples of architectural validity rules are shown in Fig 2(b).

Tests must also conform with quality requirements, following hints on how should a scenario be biased and stressed so that bugs are more likely to be exposed. This task is addressed by a large pool of expert knowledge rules, exemplified in Fig. 2(c). For example, a Fixed-Point-Unit expert may suggest that bugs are to be expected when the sum of an *Add* instruction is exactly 0, hence rule number 1 in Fig. 2(c). Altogether, for a typical architecture, up to a few thousand validity and expert knowledge rules are defined.

Finally, as each individual test corresponds to a very specific scenario, it is crucial that different tests, generated to satisfy the same set of input rules, reach out to different scenarios, hence providing some form of uniform sampling over all possible scenarios satisfying the rules.

(a) User Requests

1. At least one of the operands in an *Add* instruction is larger than 9999
2. In a multiprocessor system, the same processor issues 10 subsequent *Load* instructions to consecutive addresses

(b) Architectural Validity

1. *Memory address = base-register data + offset*
2. If memory address is aligned to 4 bytes then *Load-Word* instruction is atomic
3. Privileged instruction when user-mode bit is on results in exception

(c) Expert Knowledge

1. 30% of all *Add* instructions (a + b = c) should result in c = 0
2. 20% of all *Load* and *Store* instructions should cross page boundaries
   2.1 Another 20% should be exactly aligned to page boundaries
3. 70% of all transactions in a system should contend on the same bus

Figure 2: Three sources of rules. A few examples are shown for each source.

Model-based Stimuli Generation

IBM has long advocated the use of model-based stimuli generators (Aharon *et al.* 1995). Here, the generator is partitioned into two separate components: a generic engine that is capable of generating tests for any hardware architecture, and an input model describing the hardware architecture at hand and the expert knowledge. A number of technical challenges confront the designer of model-based, random stimuli generators:

- What is a good methodology for modeling the complex objects and rules imposed by the hardware architectures and expert knowledge?
- How can this information be easily migrated to new designs?
- What is the best way to describe test templates that can range from highly specific to highly random and that must bridge the gap between the abstract scenarios formulated in the verification plan and the concrete tests that are generated and simulated?
- Once the rules are formulated, how does the stimuli generator ensure that all user-defined and validity rules, and as many expert knowledge rules as possible, are satisfied?
- How can the generator produce many significantly different tests from the same test template?
- Finally, how is all this done in an efficient manner as to not obstruct the verification process?

As we will show, these challenges lend themselves naturally to AI-based solutions. In particular, we use AI techniques to model and satisfy complex sets of rules stated in a high-level language, and imposed, among other sources, by expert knowledge and belief systems.

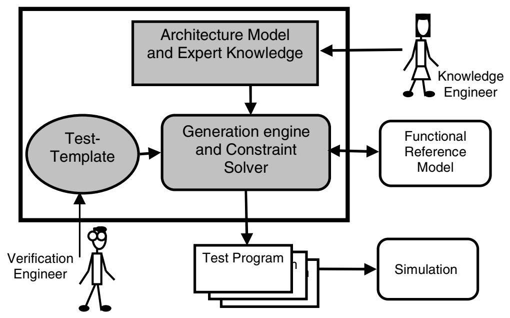

Architecture Model and Expert Knowledge
Knowledge Engineer
Test-Template
Generation engine and Constraint Solver
Functional Reference Model
Verification Engineer
Test Program
Simulation

Figure 3: Architecture of model-based stimuli generator

Application Description

Application Architecture

The architecture of our test generation application is shown in Fig. 3. It is a service-oriented architecture (SOA) derived from the separation of inputs central to model-based systems. The knowledge base contains both the declarative architectural description of the design under test and the expert knowledge repository. This knowledge base is developed and maintained by knowledge engineers who are verification experts. Test templates are written by verification engineers who implement the test plan. The generic engine, developed by software engineers, accepts the architecture model, expert knowledge, and test template, and generates a batch of tests. Each test contains a series of hardware transactions that are sent to execution on the design simulator. As part of the generation process, the generator uses a functional reference model to calculate values of resources after each generated transaction. These values are needed for the generation of subsequent transactions, and are also used for comparison with the results of the actual design simulator. Obviously, a mismatch between those results indicates a bug in the design. All parts of the application are written in C++. Graphical interfaces use the QT library.

Test Template Language

We designed a special test template language for writing partially specified verification scenarios. Here we exemplify this language for the special case of processor verification, in which the hardware transactions are single processor instructions. Fig. 4 shows an example of such a test template. The template describes a table-walk scenario that stores the contents of randomly selected registers into memory addresses ranging from address 0x100 to 0x200, at increments of 16.

As exemplified in Fig. 4, the test-template language consists of four types of statements: *Transaction statements* specify which transactions are to be generated and the various properties of each such transaction. For example, in line 3, a *load* instruction from an unspecified address to register number 5 is requested. *Control statements* Control the choice of their sub-statements in the generated test, e.g., line 6 specifies the selection of either an *add* or *sub* instruction.

```
1. Var: addr=0x100, reg;
2. Bias: resource_dependency(GPR)=30, alignment(4)=50;
3. Instruction: load R5 ← ? ; with Bias: alignment(16)=100;
4. Repeat(addr ‹ 0x200) {
5.    Instruction: store reg→addr;
6.    Select
7.          Instruction: Add ? ← reg , ? ; with Bias SumZero;
8.          Instruction: Sub;
9.    addr = addr + 0x10;
10. }
```

Figure 4: Table-walk test template.

```
Resource initialization section:
   Registers: R6 = 8, R3 = -25, … ,R17 = -16
   Memory: 0x100 = 7, 0x110 = 25, … , 0x1F0 = 16

Instruction initialization section:
   0x500: Load R5 ← 0xFF0
      :
   0x504: Store R4 → 0x100
   0x508: Sub R5 ← R6 – R4
   0x50C: Store R4 → 0x110
   0x510: Add R6 ← R4 + R3
      :
Expected results section:
     :
```

Figure 5: Test program for the table-walk scenario.

Programming constructs include variables, assignments, expressions, and assertions. *Bias statements* as specified on lines 2, 3, and 7 enable the user to control the activation percentage of expert knowledge rules of the types shown in Fig. 2(c). Bias statements are scopal, so the one on line 2 applies throughout the test, while the ones on lines 3 and 7 apply only to the instructions on those lines.

Fig. 5 shows some parts of a test resulting from the test-template in Fig. 4. The first part directs the simulator to initialize all relevant registers and memory sections with the specified data. The second part lists the instructions to be executed by the simulator. The third part (not shown) lists the expected results of all resources as calculated by the application's reference model. One can observe that all test-template specifications are obeyed by the test, while values not specified in the template are chosen at random.

Knowledge Base

The knowledge base includes a description of the design, its behavior, and expert knowledge. For example, the description of a single instruction is modeled by listing its assembly opcode and its list of operands. Each operand is comprised of attributes that describe the properties of the operand, and the values they may accept. Fig. 6 shows part of the model of a *Load-Word* instruction that loads four bytes from the

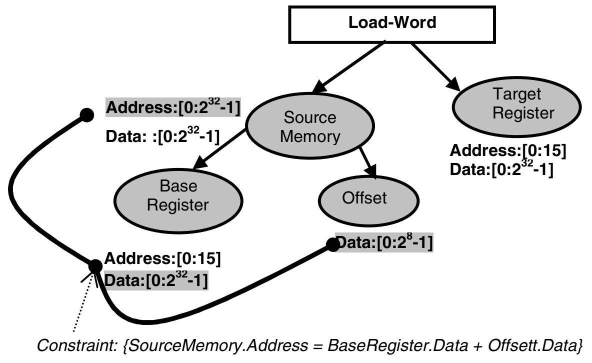
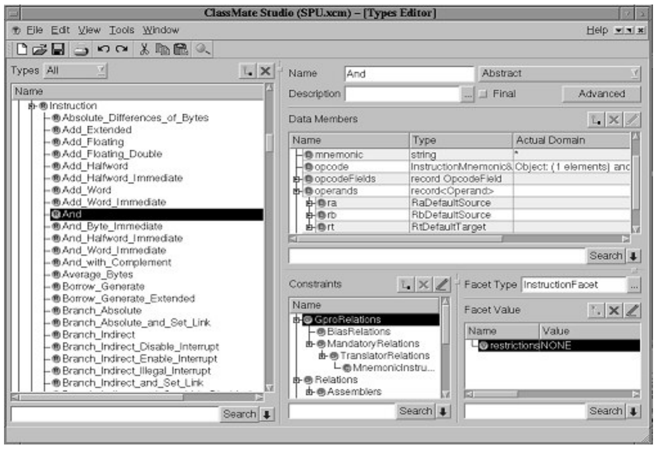


Figure 6: Partial model of a *Load-Word* instruction.



ClassMate Studio (SPU.xcm) – [Types Editor]

Name
- Instruction
  - Absolute_Differences_of_Bytes
  - Add_Extended
  - Add_Floating
  - Add_Floating_Double
  - Add_Halfword
  - Add_Halfword_Immediate
  - Add_Word
  - Add_Word_Immediate
  - And
  - And_Byte_Immediate
  - And_Halfword_Immediate
  - And_Word_Immediate
  - And_with_Complement
  - Average_Bytes
  - Borrow_Generate
  - Borrow_Generate_Extended
  - Branch_Absolute
  - Branch_Absolute_and_Set_Link
  - Branch_Indirect
  - Branch_Indirect_Disable_Interrupt
  - Branch_Indirect_Enable_Interrupt
  - Branch_Indirect_Illegal_Interrupt
  - Branch_Indirect_and_Set_Link

Name: And
Description:

Data Members

Name	Type	Actual Domain
mnemonic	string	*
opcode	InstructionMnemonic&	Object: (1 elements) anc
opcodeFields	record OpcodeField	
operands	record<Operand>	
ra	RaDefaultSource	
rb	RbDefaultSource	
rt	RtDefaultTarget	

Constraints

Name
- GproRelations
  - BiasRelations
  - MandatoryRelations
    - TranslatorRelations
      - MnemonicInstru...
- Relations
  - Assemblers

Facet Type: InstructionFacet

Facet Value

Name	Value
restrictions	NONE

Figure 7: Studio for modeling.

memory. The operands are arranged in a tree structure with the attribute names in bold and the legal values in brackets. An example of a modeled validity rule [the first rule in Fig. 2(b)] is also shown.

Instruction-specific testing knowledge [rules of the type shown in Fig. 2(c)] can also be modeled as part of the instruction model. For example, rule 1 in that figure may be modeled as

$$\text{Constraint}: \mathit{op1.data} + \mathit{op2.data} = 0,$$

where 'op1' and 'op2' are the two modeled operands of an *add* instruction.

Modeling is done through a graphical studio shown in figure 7. The studio supports simple navigation around the object hierarchy, searches, and queries.

Test Generation Engine

Test program generation is carried out at two levels: *stream generation* is driven recursively by the control statements in the test template. *Single transaction generation* (at the leafs of the control hierarchy) is performed in three stages. First, the transaction to be generated is formulated as a CSP: the CSP variables and domains are taken directly from the transaction's model, and the (hard and soft) constraints are constructed from the various sources of rules, as depicted by Fig. 2. Second, the CSP is solved, i.e., all properties of the transaction are assigned values such that all hard constraints, and a subset of the soft constraints, are satisfied. Third, the generated transaction is applied to the reference model, and the generator's internal reflection of resource states is updated accordingly.

Uses of AI

Our application relies heavily on various aspects of AI technology. First, all architectural knowledge and expert knowledge about the design under test is defined and kept as an ontology. Second, expert knowledge is applied as a hierarchical set of rules, realizing the belief system the modeler and user have about the relative importance of the rules. Third, production rules observe the test generation process and insert special transactions when conditions apply. Finally, the application's core solution technology is CSP.

Hardware Model as an Ontology

The modeling of hardware information and expert knowledge is done using an in-house modeling tool that combines the power of object-oriented and constraint-based systems. In addition to the standard features found in these systems, it has a number of features that make it particularly attractive for writing hardware models for stimuli generation. These include native support for constraints between objects and between subcomponents of objects; a type hierarchy mechanism for creating and classifying taxonomies commonly found in hardware transactions; a powerful type refinement mechanism that goes well beyond classical inheritance; packages that allow controlled redefinition of types for follow-on designs; and a rich set of extended data types, including collections, meta-types, and bitstreams for handling such things as arbitrary sized address and data values.

Expert Knowledge

The ontology described above allows verification experts to model their expertise as a set of rules as in Fig. 2(c). The important point here is that these rules are generic and applicable to a wide range of designs. Hence, the rules realize the expert's belief system about bug-prone conditions and other behavioral aspects of hardware designs in general.

Once the expert knowledge is modeled, it is applied by default to all tests generated by the application. However, the verification engineer may override this default behavior for any particular test template by either increasing or decreasing any of the biases implied by the expert rules, or by prioritizing between the various rules.

Production Rules

Our test-template language allows the definition of production rules, called *events*. An event consists of a *condition* and a *response* template. After each transaction is generated, the conditions of all the defined events are checked. Whenever a condition of an event is satisfied, its response template is generated. The response template normally inserts additional transactions into the test, and may trigger other events. A condition may refer to the processor state, e.g., the current value of some register, or to the generation state, e.g., the number and types of transactions generated so far (Adir, Emek, & Marcus 2002).

Constraint Satisfaction Problems

Why CSP There are two main reasons to use CSP as our core solution technology (Bin *et al.* 2002). First, CSP is declarative - it allows one to state the rules, and let the underlying CSP algorithm enforce them. In contrast, building a procedural program that enforces the rules of Fig. 2 in all possible instances is a virtually impossible feat. Second, CSP allows us to simply set prioritizations on the expert knowledge rules of Fig. 2(c), and again use a generic Soft-CSP algorithm to take this prioritization into account.

Why a Specialized Constraint Solver There are quite a few general-purpose constraint solvers available, both from academia and industry. However, CSPs arising from stimuli generation are fundamentally different in some respects from typical CSPs encountered elsewhere (e.g., job-shop scheduling or rostering). Below we list the most important aspects of our problems, and briefly discuss our customized solution.

While the technological details of our approach are unique to the domain of stimuli generation, the overall algorithmic framework of our constraint solver is the well-known maintain-arc-consistency (MAC) (Mackworth 1977) scheme. In the rare cases where constraint propagation, the fundamental building block of MAC, is computationally hard, we use stochastic search (Naveh 2004).

Distinguishing Aspects of our CSPs

- **Requirement to Randomly Sample the Solution Space** A certain design model together with a certain test template define a single (Soft) CSP. However, we expect to obtain many different tests out of this single template. Moreover, we want the tests to be distributed as uniformly as possible among all possible tests conforming to the template. This means that in terms of the solution scheme, we want to reach a significantly different solution each time we run the solver on the same CSP (Dechter *et al.* 2002).

  The basic way we achieve this is by randomizing all decisions in the search path of the MAC algorithm. This method can be shown to achieve reasonably disperse solutions. However, it implies that no variable/value ordering heuristics are allowed in the solution process, as the choice of variables/values must be done randomly. This leads to a major deviation from regular MAC-based techniques, which may rely heavily on such heuristics in order to reach a solution in a reasonable time.

- **Constraint Hierarchy** Expert knowledge is entered as a set of soft constraints, and these constraints may be applied in a multi-tiered hierarchy [Borning Hierarchy (Borning, Freeman-Benson, & Willson 1992)] according to their perceived importance in any specific verification scenario. While constraint hierarchies appear in other applications as well, it appears that stimuli generation stands out in terms of both the number of soft constraints in the model, and the depth of the hierarchy.

  In addition, the constraint hierarchy scheme competes with the requirement for uniformly-distributed solutions. To illustrate this, consider the case where a maximal number of soft constraints can be satisfied by only a single assignment. From a constraint-hierarchy point of view, this solution must be returned by the solver each time it is run on the problem. However, this obviously contradicts the notion of diverse tests for a given template. We overcome this problem by defining a "local metric" on the solution of the soft constraint hierarchy. We require that if a partial solution can be extended to satisfy additional soft constraints, it must be so extended. However we drop the requirement of satisfying a maximal number of soft constraints over the entire search space.

- **Huge Domains** Many of our CSP variables have exponentially large domains. The simplest examples are address and data variables that can have domains of the order of 2^{32} or larger (see Fig. 6). Handling such variables, and in particular pruning the search space induced by these variables by using constraint propagation, cannot be done using regular methods, which frequently rely on the smallness of the domain.

  We created a generic library that deals efficiently with set-operations over sets with huge cardinality. This allows us in many cases to write efficient propagation algorithms even when the input domains to constraint propagators are exponentially large. In the implementation of the set-operations we use a DNF (masks) representation of sets. A BDD representation was also tried, but so far did not prove useful for our problems.

- **Conditional Problems** Many of our problems are conditional, meaning that depending on the value assigned to some variables, extensive parts of the CSP may become irrelevant (Mittal & Falkenhainer 1989). Conditional problems occur also in other applications, e.g., manufacturing configuration, however we encounter a different flavor of these problems. For example, our full problem may consist of several weakly coupled CSPs, where the number of those CSPs is itself a CSP variable (e.g., in verification of multi-casting, the number of end-stations is part of the verification problem).

  To solve this problem efficiently we had to extend the MAC algorithm and incorporate assumption-based pruning (Geller & Veksler 2005). This scheme greatly enhances the strength of pruning under conditionality, as it simultaneously takes into account the state of all universes, in each of which only a certain subset of the conditional sub-problems exists.

- **Generic Modeling of Domain Specific Propagators** Some of our constraint propagators are extremely complex and require months to implement. However, the hardware specification may change on the same time-scale, rendering the implementation obsolete. We therefore invested a significant effort to generalize the more complex domain specific constraints, and develop parametric propagators that are relatively simple to manipulate and change when the design changes or when the next-generation design arrives (Adir *et al.* 2003).

Application Use and Payoff

In this section, we report concrete usage and payoff of a tool named Genesys PE. This stimuli generator is intended solely for the verification of processors and multi-processor configurations. A second tool, X-Gen, is a test-generator for the system level, and is described shortly in the next section. X-Gen is younger than Genesys PE and shows usage and payoff schemes resembling Genesys PE at a similar stage of development.

Since 2000, Genesys PE has been used as the major functional verification tool for all IBM PowerPC processor designs. In all these designs, Genesys PE has been widely deployed for the unit, core, and chip-level verification, and partially deployed for system-level verification. At the core and chip levels, where most of the complex uni-processor and multi-processor bugs are found, this is the only functional verification technology used for verifying the correctness of the processor's design.

AI Related Payoffs

Studying the AI-related payoffs of our application is relatively simple, as the tool Genesys PE has evolved from its predecessor, Genesys. The main differences between the two generations of tools is that the former did not include many of the AI features reported above. We can therefore directly compare how these AI enhancements impacted the tool in terms of verification productivity, quality, support of healthy processes, and operational costs. We found significant improvements in all four categories.

Improved Verification Productivity Genesys PE produces a more compact test suite than Genesys. This became clear early on in the deployment of the system, when a large body of existing Genesys test-templates was converted into Genesys PE templates. In one typical case, the verification plan required about 35,000 Genesys test-templates. Of these, 1900 test-templates were written directly in the Genesys scenario definition language, and 33,000 were auto-generated by scripts that enumerated them. In contrast, the same verification plan was implemented in Genesys PE with only 2,000 test templates. The functional coverage of both suites was the same. The reduced number of test templates allowed rapid testing of the design when late changes were introduced shortly prior to casting in silicon, thereby reducing time-to-market.

Improved Verification Quality The CSP-based test generation technology allows Genesys PE to expand the space of verifiable scenarios over Genesys. For example, scenarios that define conjunctive constraints (e.g., satisfy the conditions for multiple exceptions), were sometimes impossible to generate with Genesys, but often achieve full coverage with Genesys PE. In addition, the new test-template rule-definition language provides a large range of programming constructs that allow complex scenarios to be described in their most general form (Behm *et al.* 2004). In Genesys, only a subset of such scenarios could be written, and those did not always cover the desired events.

Improved Verification Processes Due to its sophisticated modeling approach, Genesys PE allows verification scenarios to be described in a design-independent manner. This capability has had a large impact on processor verification methodology inside IBM. It has led to the development of a large body of high-quality test templates that cover the PowerPC architectural and micro-architectural verification plan, now being used in the functional verification of every PowerPC design throughout the company.

Reduced Operational Costs Past projects that used the Genesys test generation technology included a team of about 10 verification engineers at the core and chip levels. Deploying Genesys PE reduced the size of the core verification team from 10 to two to four experts now responsible for coding the test-templates. The direct saving from this reduction is estimated at \$2-\$4M per a two-year project.

Market Value

The market value of Genesys PE may be calculated by considering the decreased time-to-market and the resulting increase in revenue. It is commonly estimated that a three month delay in arriving to market decreases revenue by 10 to 30 percent. We consider as an example IBM's Power5 based p-Series Unix server verified with Genesys PE (Victor *et al.* 2005). According to IDC reports (IDC-Report 2005), IBM sold over \$4B of these servers in the last year. For this product alone, IBM's most conservative estimates tag the additional revenue due to decreased time-to-market made possible by Genesys PE's early detection of functional bugs at \$50M. A conservative estimate for the overall savings of the application is \$150M.

Application Development and Deployment

The Genesys PE project was initiated in 1998 as a next generation processor stimuli generator. The existing tool at the time, Genesys, had been deployed for over eight years.

Developing the Test Generation Engine

The development work followed a feasibility study period in which a very simple test generator was developed to test the capabilities of a new constraint solver and to prove its ability to support Genesys PE's verification language. The architecture-independent test generation engine was developed by a team of four to six experienced programmers in C++, for UNIX/Linux and AIX platforms. In parallel, a separate team of two to three programmers continued to develop the constraint solver and other core technologies. The two development teams belong to the same organization unit. This enabled close cooperation while maintaining a clear separation between the different modules. Most of the functionality of the new tool was implemented over a period of 18 months.

Developing the Knowledge Base

The development of the knowledge base was separated into two phases. In the first phase, the generic, architecture-independent part of the model was developed by the same team that worked on the test generation engine. The effort consisted mainly of defining rules that implement generic testing knowledge, and rules that facilitate the test generation process. This part of the knowledge base is shared by all the designs that use Genesys PE. In the second phase, an automatic converter was used to convert the large body of design-dependent testing knowledge that existed in Genesys. This helped increase user's confidence that the system is at least as good as the old one, and allowed a quick bring up. However, it also meant that the initial system did not utilize the full capabilities of Genesys PE.

First Deployment

The successful integration of Genesys PE into user environments was a gradual and painful process that took several months. Initially users were reluctant to make the transition to the new tool; they already had a reliable working system, Genesys, and a full verification plan implemented in Genesys test-templates. Genesys PE, on the other hand, was a new and largely untested tool. Bringing it to work in production mode at user sites meant that thousands of tests were generated and simulated every night on hundreds of machines. This process unearthed numerous bugs in the generator that could not be detected during the development phase, since the development team could not match the human and computation resources for this task.

Users started to use Genesys PE to test verification scenarios that could not be fully described in Genesys, but continued to use Genesys for their ongoing verification work. This helped increase their trust and enthusiasm for the new tool, without compromising their ongoing verification schedule. It also gave the development team time to improve the tool's reliability and make it more robust.

The next stage in the integration process was to automatically convert the large body of Genesys test-templates to Genesys PE, and prove, using functional coverage measurements, that Genesys PE provides the same level of coverage as Genesys. Once this stage was successfully completed, the road was clear for having Genesys PE fully replace the Genesys tool inside IBM.

Knowledge Engineers Training

The deployment of Genesys PE also involved the training of Genesys knowledge engineers to maintain the Genesys PE knowledge base. Knowledge engineers had to become familiar with the concept of CSP solving, understand the difference between functions and constraints, and between hard and soft constraints, in order to model instructions and operands accurately enough to allow for successful test generation. It took knowledge engineers about a year to fully grasp the Genesys PE modeling approach.

Specialized Test Generation Tools

The new verification options introduced by Genesys PE have inspired users to request new tool capabilities that will make complex architectural mechanisms easy to verify. This has led to the development of two specialized test generators, DeepTrans (Adir *et al.* 2003) in the areas of address translation and FP-Gen (Aharoni *et al.* 2003) for floating point verification. Both tools have become part of Genesys PE.

Applying the Technology at the System Level

Following the successful application of the technology for processor verification, a new project called X-Gen was initiated in 2000, with the goal of applying the technology for system-level stimuli generation (Emek *et al.* 2002). X-Gen was designed with a similar knowledge-based architecture as Genesys PE and uses the same CSP solver. The main difference is the modeling language, which reflects on the differences in the domain: components, system transactions, and configurations become first-class members of the language.

In 2002, X-Gen was tested by running in parallel with a legacy stimuli generator for systems, which was not knowledge-based. The test showed that X-Gen was able to achieve higher coverage metrics in one-fifth of the simulation time and in one-tenth of test templates. This positioned X-Gen as the primary stimuli generator for IBM high-end systems, and since 2002 it has been used in the verification of most high-end system designs, including the p-Series server and the Cell-processor-based systems.

Maintenance

The multitude of designs simultaneously verified with our application and the instability of the hardware specification make maintainability crucial for long term success. We have found that the application's service-oriented architecture provides solutions to a number of key maintainability concerns, such as defining responsibilities, knowledge reuse, adapting to change, and ongoing upgrades.

Defining Responsibilities

The application's architecture helps define a clear separation of responsibility and source code ownership. Each part of the tool is maintained by its respective development team. Knowledge engineers (three to four per tool) provide on-site support for the users and adapt the knowledge base to design changes. Tool developers (seven to eight per tool) provide support for generic changes in the tools and second-line support for the users. Two to three core technology developers provide solutions to new requests coming from the application development teams.

Knowledge Reuse

The partition of the tool into a generic generation engine and a knowledge base allows a high level of reuse. All the capabilities of the test generator, and the generic testing knowledge, are immediately available for any new design. In addition, designs that belong to different generations of the same hardware architecture are modeled in a hierarchy that reflects their lineage. Thus common building blocks, such as instructions, operands, resources, and common testing-knowledge are also shared between the designs.

Adapting to Change

Hardware design verification usually starts when the hardware architecture is still evolving. Thus, any change to the hardware specification must be reflected in the knowledge base and reference model in a timely manner. The separation between the two modules allows them to be developed in parallel. In addition, the fact that the different modules are developed and maintained by different teams helps check the correctness of one module relative to the other.

Ongoing Upgrades

The test generation tools continue to develop in a process of staged delivery. This allows gradual evolution of the tools with user feedback. Knowledge engineers and tool developers maintain separate systems, and synchronize sources during each release - typically once a month. Tool developers are located in different geographical areas and time zones from the users and knowledge engineers. However, the use of a unified defects database and regular weekly phone conferences help ease communication difficulties.

Summary

We presented random stimuli generation for hardware verification in IBM as a complex application relying on various AI techniques. We discussed the challenges and huge payoffs of building this application. The research and development around this application is still thriving, and we are steadily exploring more sophisticated CSP and knowledge representation techniques to keep up with the ever-growing complexity of hardware systems and business requirements.

Acknowledgments

We are grateful to Yossi Lichtenstein for participating in the writing of this paper and for many useful comments, and to Merav Aharoni for assistance in graphics. The work presented here could not have been accomplished without the innovation, talent, and dedication of dozens of researchers and engineers at IBM's Haifa Research Lab, and without the continuous feedback of verification engineers across IBM.

References

Adir, A.; Emek, R.; Katz, Y.; and Koyfman, A. 2003. DeepTrans - a model-based approach to functional verification of address translation mechanisms. In *Fourth International Workshop on Microprocessor Test and Verification (MTV'03)*, 3–6.

Adir, A.; Emek, R.; and Marcus, E. 2002. Adaptive test program generation: Planning for the unplanned. In *Seventh IEEE International High-Level Design Validation and Test Workshop, HLDVT-02*, 83–88.

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

Aharoni, M.; Asaf, S.; Fournier, L.; Koyfman, A.; and Nagel, R. 2003. A test generation framework for datapath floating-point verification. In *Eighth IEEE International High-Level Design Validation and Test Workshop, HLDVT-03*, 17–22.

Behm, M.; Ludden, J.; Lichtenstein, Y.; Rimon, M.; and Vinov, M. 2004. Industrial experience with test generation languages for processor verification. In *41st Design Automation Conference (DAC'04)*, 36–40.

Bergeron, J. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers.

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

Borning, A.; Freeman-Benson, B.; and Willson, M. 1992. Constrain hierarchies. *Lisp and Symbolic Computation* 5:223–270.

Dechter, R.; Kask, K.; Bin, E.; and Emek, R. 2002. Generating random solutions for constraint satisfaction problems. In *Eighteenth National Conference on Artificial Intelligence*, 15–21.

Emek, R.; Jaeger, I.; Naveh, Y.; Bergman, G.; Aloni, G.; Katz, Y.; Farkash, M.; Dozoretz, I.; and Goldin, A. 2002. X-Gen: A random test-case generator for systems and socs. In *Seventh IEEE International High-Level Design Validation and Test Workshop, HLDVT-02*, 145–150.

Geller, F., and Veksler, M. 2005. Assumption-based pruning in conditional CSP. In van Beek, P., ed., *CP*, volume 3709 of *Lecture Notes in Computer Science*, 241–255. Springer.

IDC-Report. 2005. Worldwide server market shows growing it investment across platforms, according to idc (retrieved on Apr. 2, 2006). *www.idc.com/getdoc.jsp?containerId=prUS00223005*.

Lichtenstein, Y.; Malka, Y.; and Aharon, A. 1994. Model based test generation for processor verification. In *Sixth Annual Conference on Innovative Applications of Artificial Intelligence*, 83–94.

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

Mittal, S., and Falkenhainer, B. 1989. Dynamic constraint satisfaction problems. In *Eighth National Conference on Artificial Intelligence*, 25–32.

Naveh, Y. 2004. Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In *Local Search Techniques in Constraint Satisfaction (LSCS-04)*.

Victor, D. W.; Ludden, J. M.; Peterson, R. D.; Nelson, B. S.; Sharp, W. K.; Hsu, J. K.; Chu, B.-L.; Behm, M. L.; Gott, R. M.; Romonosky, A. D.; and Farago, S. R. 2005. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM Journal of Research and Development* 49(4/5):541–554.