AI Magazine Volume 28 Number 3 (2007) (© AAAI)

Constraint-Based Random Stimuli Generation for Hardware Verification

Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek

■ We report on random stimuli generation for hardware verification at IBM as a major application of various artificial intelligence technologies, including knowledge representation, expert systems, and constraint satisfaction. For more than a decade we have developed several related tools, with huge payoffs. Research and development around this application are still thriving, as we continue to cope with the ever-increasing complexity of modern hardware systems and demanding business environments.

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 the casting of fewer 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 expert knowledge, 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 (MAC) 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, and 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 and system verification within IBM. It has been used in the development of numerous IBM PowerPC processors, i/p-series server systems, the Cell, and Microsoft's Xbox core processors. 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.

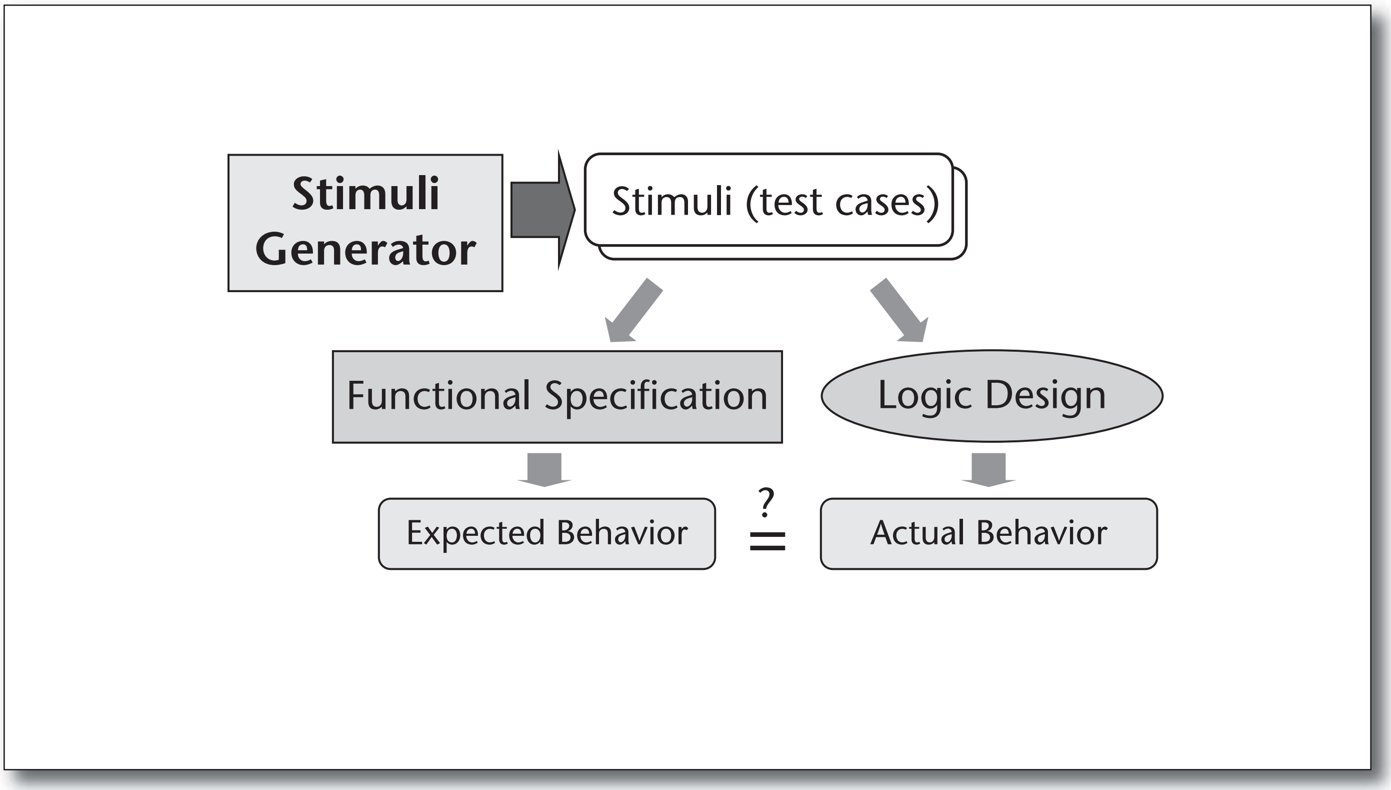


Figure 1. Simulation-based Verification.

Problem Description

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 into gate-level circuits. 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 model, as illustrated in figure 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 major 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 nontrivial because the verification plan is typically formulated in a 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

(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 percent of all *Add* instructions ($a + b = c$) should result in $c = 0$
2. 20 percemt of all *Load* and *Store* instructions should cross page boundaries
   2.1 Another 20 percent should be exactly aligned to page boundaries
3. 70 percent 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.

al. 1995) has become the verification approach of choice. Here, many tests are generated automatically by a pseudorandom stimuli generator. The input to such a generator is a user request outlining the scenarios that should occur when the test is executed at simulation. For example, in figure 2a 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 a 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 figure 2b.

Tests must also conform with quality requirements, following hints on how a scenario should 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 figure 2c. 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 figure 2c. 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.

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

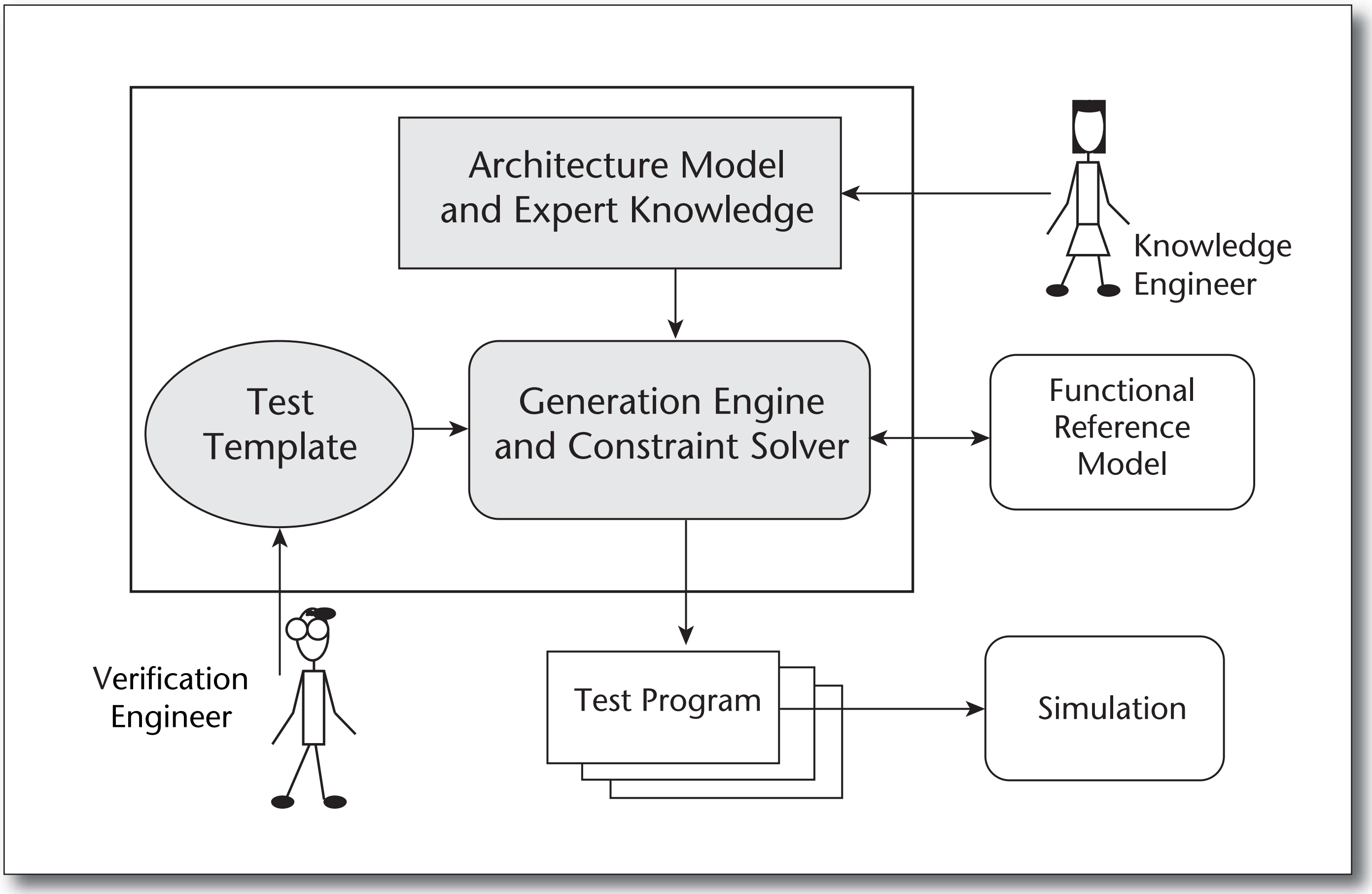

Architecture Model and Expert Knowledge
Knowledge Engineer
Test Template
Generation Engine and Constraint Solver
Functional Reference Model
Verification Engineer
Test Program
Simulation

Figure 3. Architecture of Model-based Stimuli Generator.

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.

Application Description

The architecture of our test-generation application is shown in figure 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

```
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.

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.

It is important to note that the architecture thus described is completely generic, and has been used within IBM in a number of test generators for verifying systems (X-Gen [Emek et al. 2002]), processors at the architecture (Genesys PE [Adir et al. 2004]) and microarchitecture (Piparazzi [Adir et al. 2003a]) levels, hardware units (FPGen, DeepTrans [Adir et al. 2003b; Aharoni et al. 2003]), and systems on a chip (SoCVer [Nahir et al. 2006]). Each brings its own set of problems and challenges that are unique to the specific hardware domain. While all of these generators adhere to the architecture described, for consistency reasons we choose one of them, Genesys PE, as the running example used throughout this article.

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. Figure 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 figure 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 substatements in the generated test; for example, line 6 specifies the selection of either an "add" or "sub" instruction. *Programming constructs* include variables, assignments, expressions, and assertions. *Bias statements* as specified on lines 2, 3, and 7 of figure 4 enable the user to control the activation percentage of expert knowledge rules of the types shown in figure 2c. 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.

Figure 5 shows some parts of a test resulting from the test template in figure 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

```
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.

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 comprises the attributes that describe the properties of the operand and the values they may accept. Figure 6 shows part of the model of a "Load-Word" instruction that loads four bytes from the 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 figure 2b) is also shown.

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

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

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

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 figure 2. Sec-

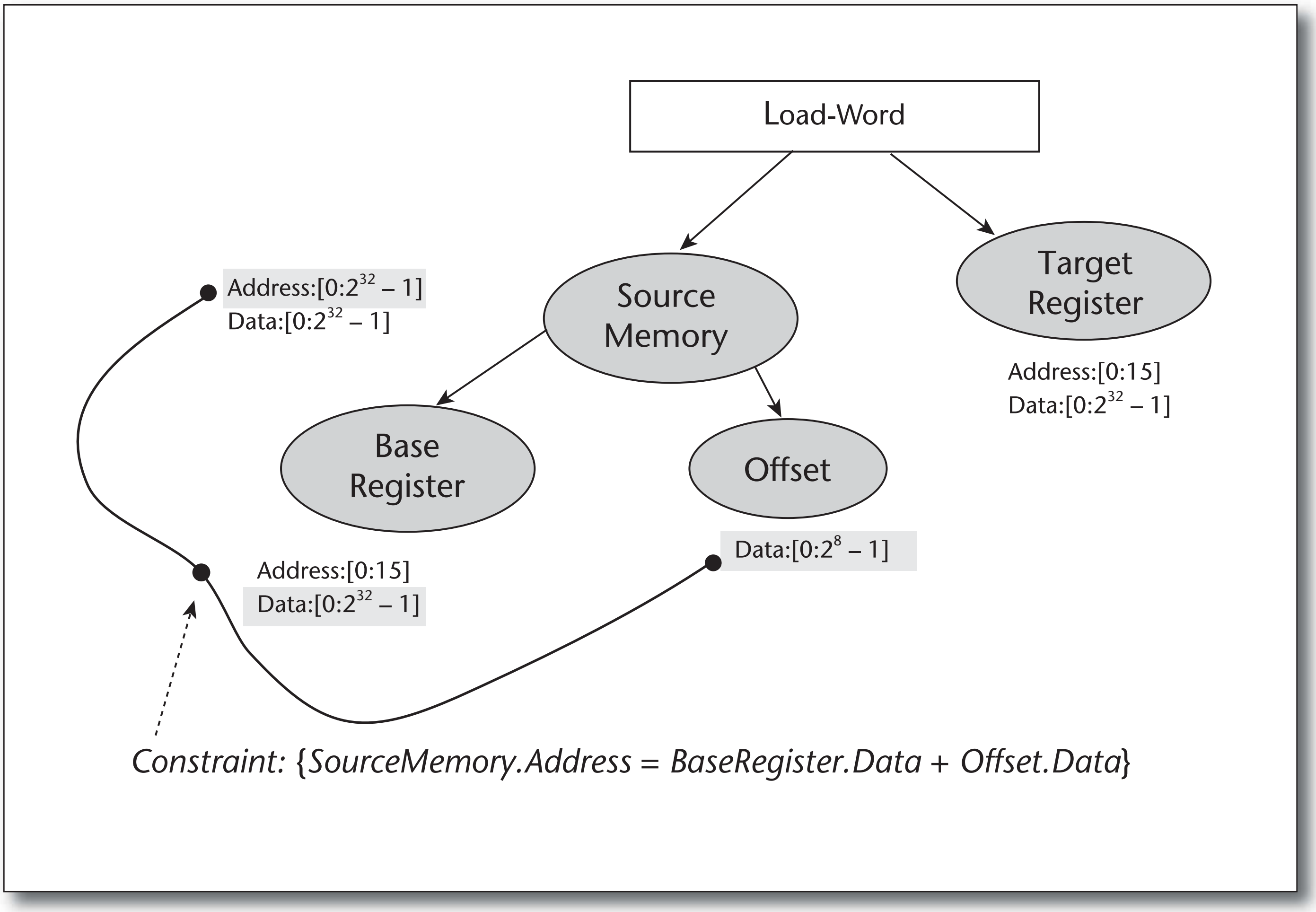


Figure 6. Partial Model of a Load-Word Instruction.

ond, the CSP is solved, that is, 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.

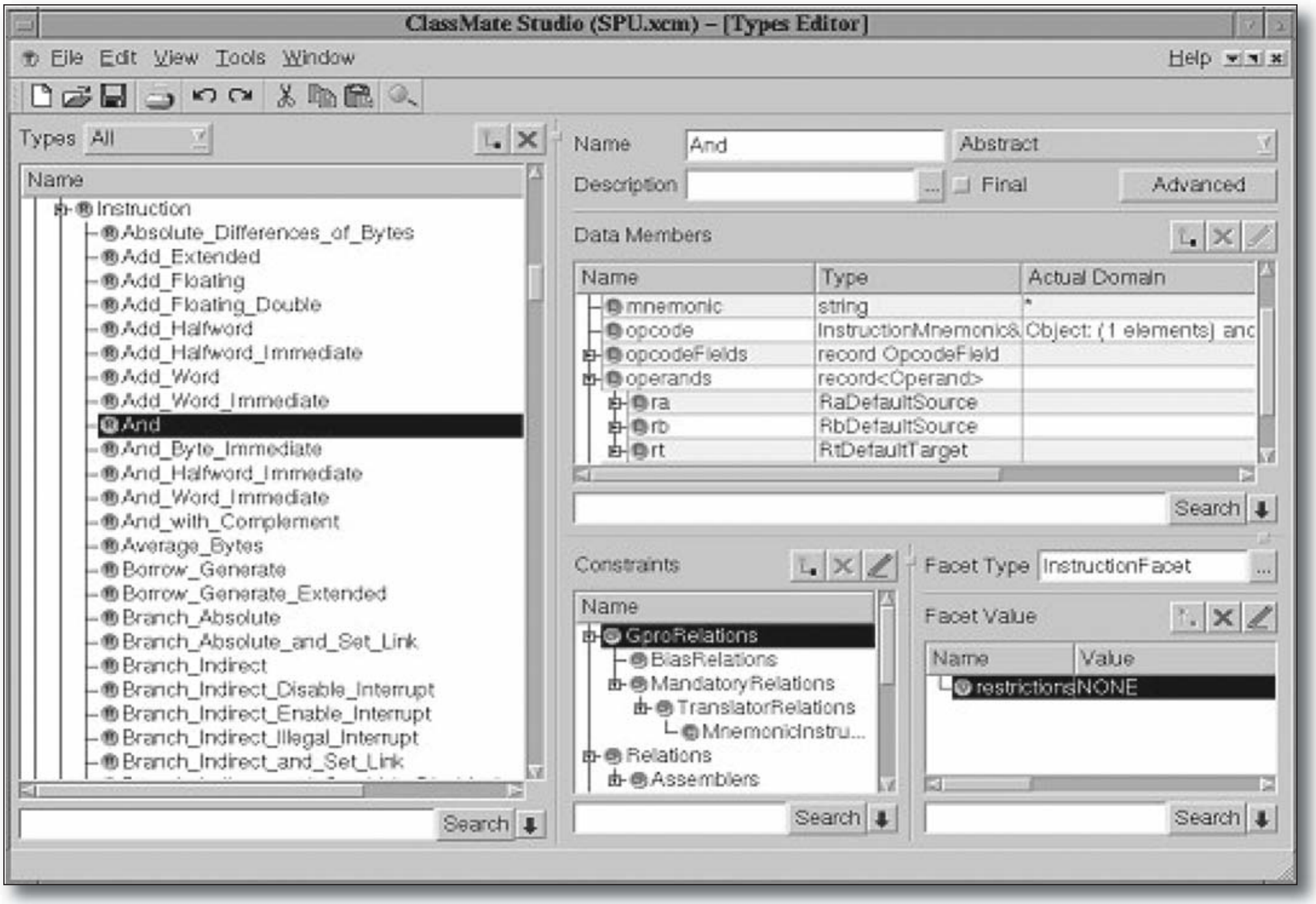
At the system level, a CSP induced by a single transaction involves a large number of component-level transactions that need to be consistently resolved. The resulting constraint network is typically too large and too complex to be solved in a single step — primarily due to its highly dynamic nature. In this case we further partition the problem and generate every transaction in two steps. First, we solve an abstract constraint network determining the structure of the transaction and the identity of the participating components. Only then do we construct and solve a detailed network determining the actual variables. This approach can be viewed as a form of CSP abstraction (Emek et al. 2002).

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, called *ClassMate*, 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 genera-


ClassMate Studio (SPU.xcm) – [Types Editor]
File Edit View Tools Window
Help
Types All
Search
Advanced
Data Members
Constraints
Facet Value

Name: And — Abstract
Description: — Final

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

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

Facet Type: InstructionFacet

Name	Value
restrictions	NONE

Figure 7. Studio for Modeling.

tion. 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, metatypes, and bit vectors for handling such things as arbitrary-sized address and data values.

Modeling is done in ClassMate with a graphical studio shown in figure 7. The studio supports browsing and editing (including refactoring) of all objects in the taxonomy. In addition, it provides a search and query mechanism that enables users to readily navigate between an object's definition and its usages.

Expert Knowledge

The ontology platform described above allows verification experts to model expert knowledge as a set of rules as in figure 2c. 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, for example, the current value of some register, or to the generation state, for example, the number and types of transactions generated so far (Adir, Emek, and Marcus 2002). An example of a capability of the test generator enabled by this mechanism is the generation of relocatable interrupts. These are interrupts whose handlers do not reside in predefined memory locations. In this case, the event's condition identifies that the insertion of handler code is required, and the event's body generates the desired handler code.

Constraint-Satisfaction Problems

There are two main reasons to use CSP as our core solution technology. 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 figure 2 in all possible instances is a virtually impossible feat. Second, CSP allows us to simply set prioritizations on the expert knowledge rules of figure 2c 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 (for example, job-shop scheduling or rostering). In the next two subsections, we list, respectively, the most important distinguishing aspects of our problems and the ways we tackle those problems.

While the technological details we describe 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. A basic building block of MAC is a procedure for achieving consistency over a single constraint (the Revise procedure in Mackworth [1977]). This procedure reduces the domains of the constrained variables by removing all values (and only these values) that cannot be part of any assignment that satisfies the constraint. In the context of a constraint over n variables, consistency is achieved by projecting the R^n relation defined by the constraint onto the individual variable domains. The term *propagator* refers to this single-constraint consistency-achieving procedure. Figure 8 demonstrates the projection of a constraint R onto two input domains, A and B, and the resulting reduced domains: A' and B'.

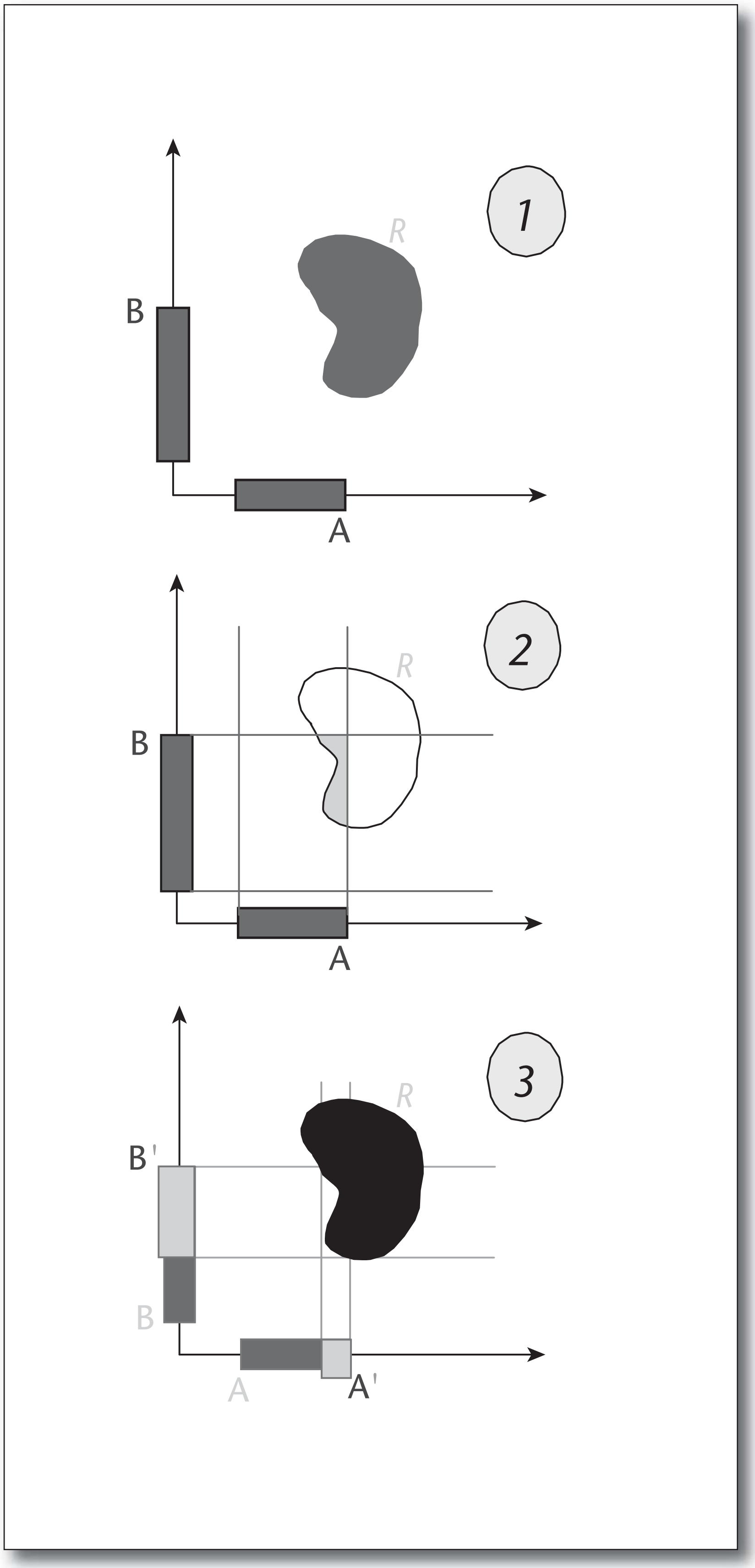


Figure 8. Propagation of Constraints by Projecting Domains.

In cases where constraint propagation is computationally hard or otherwise prohibitive, we use stochastic search.

Distinguishing Aspects of our CSPs

Requirement to Randomly Sample the Solution Space. A certain design model together with a certain test template defines 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).

Constraint Hierarchy. Expert knowledge is entered as a set of soft constraints, and these constraints may be applied in a multitiered hierarchy (Borning Hierarchy [Borning, Freeman-Benson, and 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.

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^{64} or larger (see figure 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.

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 (in the early literature these problems were also known as "dynamic CSP" [Mittal and Falkenhainer 1989]). Conditional problems occur also in other applications, for example, manufacturing configuration; however, we encounter a different flavor of these problems. In one typical scenario our full problem may consist of several weakly coupled CSPs, where the number of those CSPs is itself a CSP variable. For example, in verification of multicasting, the number of end stations is part of the verification problem.

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.

Computationally hard propagators. These are abundant in the verification of floating-point units, where typical constraints may require $a \times b = c$, and that the number of on bits in the binary representation of *a, b,* and c is some constant.

The Generation Core Toolbox

Our solutions to the problems described in the previous subsection are implemented in the Generation Core toolbox. This collection of tools and C++ class libraries provides services and building blocks for the construction of stimuli generation tools. The set of services includes (1) ClassMate, described previously, (2) GEC, a MAC-based CSP solver, (3) Stocs, a stochastic CSP solver, (4) Expression-Relation Propagator (ERP), which is an expression-driven constraint propagator, and (5) Primitive Domains (PDs), a collection of set-representation classes and set operators for dynamic representation and manipulation of CSP variables.

The GEC solver manipulates a network of constraints, where each constraint is represented by its propagator. Constraint networks may be dynamically constructed and modified to enable, for example, problem partitioning by abstraction. GEC allows well-distributed sampling of the solution space through careful randomization of solution-path decisions. Note that by committing to randomize the solution path we rule out the use of variable and value ordering heuristics.

The supported soft-constraint hierarchies, conditional problems, and approximated propagators require special attention because of this sampling aspect. To overcome the inherent conflict between the notions of constraint hierarchies and randomized solutions, we define a "local metric" on the solution of the 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. We believe that this also matches well the natural interpretation of soft, expert knowledge constraints in the context of test generation.

All propagators implement a common interface. Propagators are treated uniformly, as black-box procedures, by the solution scheme, whether implemented by the user to capture a specific constraint or automatically synthesized by ERP. GEC defines a common interface for CSP variables, supported by the PD class hierarchy.

We describe in the following paragraphs some of the more valuable components and mechanisms of the Generation Core.

Expression-Relation Propagator (ERP). Given a combined arithmetic/logical expression representing a constraint relation, for example, $(a = b + c)$ or $(c = d)$, ERP constructs the respective propagation procedure required by the solver. As with other expression-based constraint languages, for example, OPL (Hentenryck 1999), by using ERP we avoid the error-prone and labor-intensive process of developing a special-purpose procedure for each constraint. ERP Expressions include arithmetic, logical, and bit-wise operators. Apart from the definition of the grammar, the algorithm only requires a local propagation function for each of the operators in the grammar and is usually easily expandable to include more operators. The algorithm transforms the parse tree of an expression into a constraint network representing the expression. When the resulting network is acyclic, achieving arc consistency on the network through the usage of local propagation functions produces the required propagation for the combined constraint specified by the expression. The domains are reduced to hold all, and only, the necessary values (Freuder 1982). We use a combination of several methods to deal with more complex, cyclic constraints. For more details, refer to (Bin et al. 2002).

Primitive Domains: Set Representation and Set Operators. The method used to represent the domains of CSP variables throughout the solving process affects the space needed to store the variables and the efficiency of performing operations on them. Our framework requires the support of exponentially large variable domains, efficient implementation of operators that are commonly used by propagation functions, and an efficient uniformly distributed random selection of an element from a domain. All this, while exhibiting a compact memory signature.

The PD class library supports the variable types commonly used in our CSPs. For each of the supported types, the library offers one or more efficient ways to represent domains of the type, and the required operations performed on the domains. The most commonly used types are: integers, bit vectors, Booleans, enumerators, and strings.

Supported operations can be divided into three main groups: Standard set operations, such as union and intersection; set-arithmetic operations, such as addition and multiplication; and set-bit-wise logic operations, such as bit-wise xor. Note that these operations are defined over the Cartesian product of the operand domains, for example, for integer domains A and B, $A + B = \{(a + b) | a \in A, b \in B\}$.

The representation method selected for each variable is a modeling decision, usually based on the operations in which the variable is involved. Automatic conversions between representation methods are handled by GEC but should be avoided due to their typically high computational cost. The challenges are best illustrated by the integer and bit-vector domains. We support three alternative representation methods for these domains. Unfortunately, none is efficient for all operations, as described in the following paragraphs.

Set of ranges. The values in a domain are represented as a set of nonoverlapping intervals (for example, {[0,100], [200,299]}). Ranges are a natural way to represent integers, and it is efficient to perform certain arithmetic operations, such as addition and subtraction, over them. Ranges are also efficient in performing set operations, such as union and intersection, with other sets of ranges. On the other hand, performing bit-wise operations on ranges can be hard.

Set of masks (DNF). A mask is a bit vector with don't-care ("X") values for some of its bits. Each mask represents the set of bit vectors that may be obtained by determining don't-care bits. For example, the mask XXXXXX00 represents all the bit vectors that end with two 0s. Any set of integers can be represented as a set of masks. Masks are very effective for bit-wise operations and can be used efficiently in set operations with other masks. They are also convenient as input media for hardware-related CSPs. However, performing arithmetic operations on masks is hard.

Binary Decision Diagrams (BDDs). Binary Decision Diagrams (Wegener 2000) are data structures commonly used to represent Boolean functions. A BDD is associated with a set of values in a domain by representing the characteristic function of the set $(f_D(x) = 1 \Leftrightarrow (x \in D)$. Comparisons and basic set operations can be done very efficiently on BDDs. BDDs also have efficient algorithms for some arithmetic operations, such as addition (Wegener 2000). On the other hand, BDDs are less efficient than masks in handling bit-wise operations. Figure 9 demonstrates a "plus" operation with DNF and BDD representations.

Assumption-Based Pruning. We extended MAC to be applicable to conditional problems, and this way enabled propagation of constraints under high conditionality (Geller and Veksler 2005). Under this scheme, termed *assumption-based pruning*, propagation takes into account the state of all universes, in each of which only a certain subset of the conditional subproblems exists.

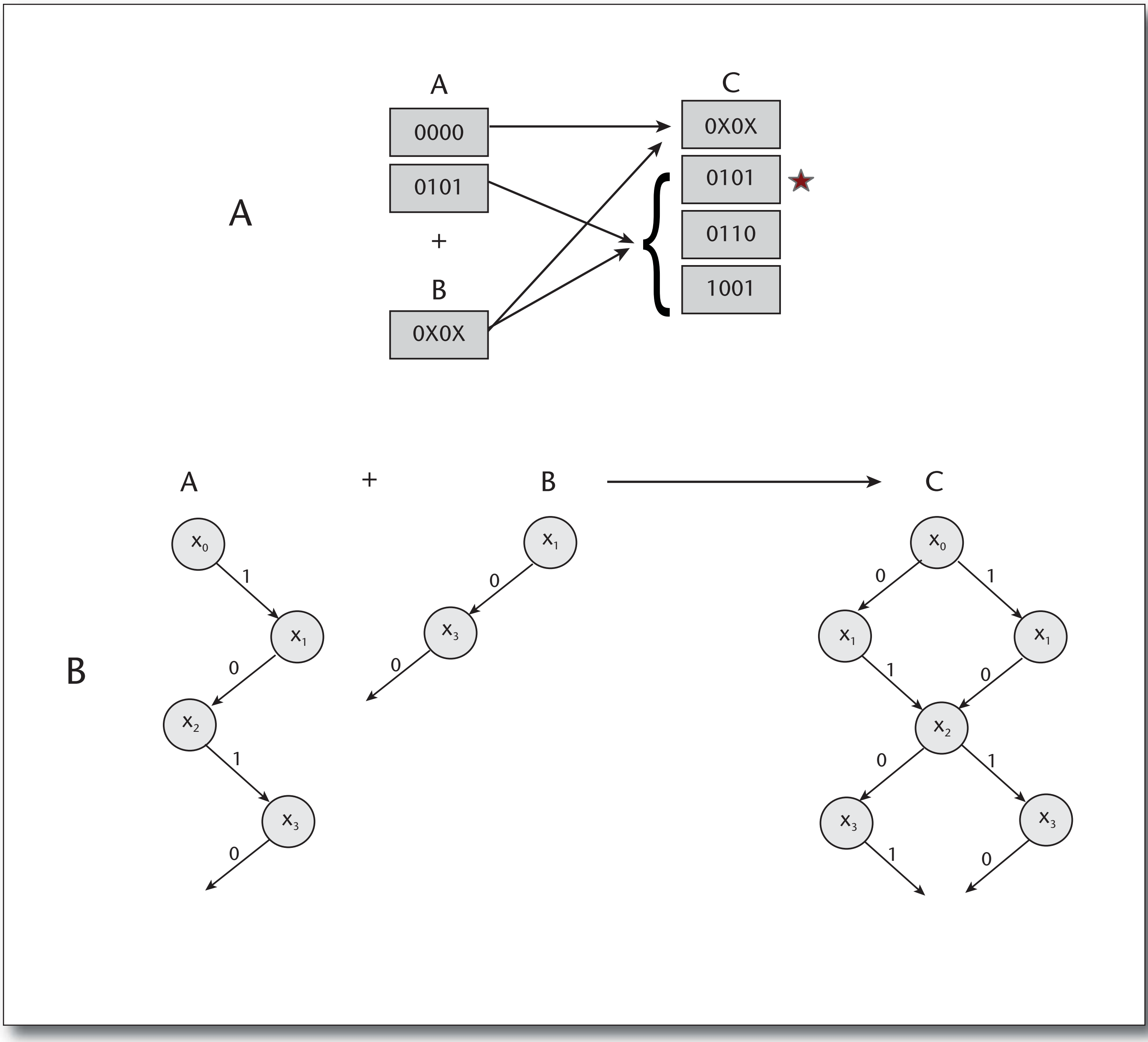


Figure 9. "plus" Operation (a + b = c) on Sets with (a) DNF and (b) BDD Representation.

The asterisk (*) in (a) marks a redundant mask in the result of the operation.

Managing Approximated Propagators. Some constraints are hard to propagate. An example is the constraint $a = b + c$, where the domains of a, b, and c are large and highly fragmented. An approximated projection procedure will aim to reduce the computational complexity at the cost of producing inexact output domains (for example, leaving domain values that cannot be extended to a solution of the constraint). GEC accepts approximated projectors but requires that when each variable is assigned a single value, the projector should serve as a predicate and validate the assignment. This ensures that the final variable assignment reached by GEC is indeed a solution.

The simplest approximation method would be to keep the affected variable domains intact if computing the propagation is not feasible. GEC provides an optional mechanism to optimize managing this special case. Propagation precondition is a mechanism that associates Boolean predicates with constraints. A constraint propagator is activated if and only if its precondition predicate evaluates to true. It is essential that at some point along the solution process, for example, when the domains become small enough, the predicate will evaluate to true, allowing GEC to take into account this constraint. The precondition mechanism makes GEC aware of pending propagators. This enables optimization schemes such as postponing lower priority propagators (in the presence of a constraint hierarchy) and variable or value choice decisions affecting variables that are constrained by the pending propagator.

Global State Constraints. In a CSP representation of a test program, many variables are constrained by the global state of the processor (mostly registers and memory values) and, in turn, influence that state. The huge number of memory addresses rules out a CSP model in which every memory cell has a corresponding CSP variable. In our framework, we use a global data structure (typically a map or a hash table) to maintain the state of each statelike entity. In the CSP model, each state entity is represented by a single variable of a special, abstract type. Other CSP variables synchronize with the state entity through constraint propagators linked to this abstract variable.

Complex Propagators. To cope with frequent changes of hardware specification, we have developed parametric propagators for the more complex domain-specific constraints. These objects capture the most abstract functionality of the propagator. When the design changes or when the next-generation design arrives, only simple manipulation of the parameters (which are part of the hardware model) is necessary in order to obtain the full functionality of the new propagator (Adir et al. 2003b).

Stocs. We use this stochastic CSP solver in two main cases, first, to solve problems with computationally hard propagators; and second, when the implementation time of some complex propagators is prohibitive. Details of SVRH, the algorithm behind Stocs, of the solver's constraint-based architecture, and of some use cases may be found in Naveh (2004).

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 multiprocessor configurations. A second tool, X-Gen, is a test generator for the system level and is described shortly at the end of the 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 core and chip-level verification and has been partially deployed for unit and system-level verification. At the core and chip levels, where most of the complex uniprocessor and multiprocessor bugs are found, this is the only functional verification technology used for verifying the correctness of the processor's design.

AI-Related Payoffs

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. With Genesys PE, a verification plan would be implemented using a compact suite of test templates compared to its predecessor, Genesys. This became clear early on in the deployment of the tool, 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 autogenerated 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 (for example, 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 microarchitec-The templates are 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 2–4 experts now responsible for coding the test templates. The direct saving from this reduction is estimated at $2–$4 million for 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,[1] IBM sold more than $4 billion 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 $50 million. A conservative estimate for the overall savings of the application is $150 million.

Application Development and Deployment

The Genesys PE development project was initiated in 1998 as a next-generation processor stimuli generator. The existing tool at the time, Genesys, had been deployed for more than 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 in 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 users' 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 to 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 and to 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.

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 also uses GEC. The main difference is the modeling language, which reflects on the differences in the domain: components, system transactions, and configurations become first-class entities 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 ever evolving and changing 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) are responsible for generic upgrades of the tools and provide 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 the knowledge engineers. However, the use of a unified defects database and regular weekly phone conferences helps ease communication difficulties.

Summary

We presented random stimuli generation for hardware verification in IBM as a sophisticated application relying on various AI techniques. We discussed the challenges and huge payoffs of building this application. Overall, we are happy with our solution. As detailed above, we made existing AI technology adequate to our distinguishing needs by developing it beyond the known state-of-the art.

Lessons Learned

One important lesson can be learned from our ongoing attempts to design the domain-specific languages (for example, those used by Genesys PE and X-Gen) as pure constraint languages that are also natural to the domain experts and knowledge engineers. Had we succeeded with that we could have synthesized the tools directly from ClassMate's metamodel, thus significantly reducing implementation time of the more specific languages and allowing fast buildup of new verification domains. However, we consistently found that natural declarative modeling of hardware architectures (and even testing knowledge) is significantly different from modeling the test-generation problem, which may be a pure CSP. It turns out that a great deal of domain insight and exploration is required in order to come up with a natural domain description language and an effective transformation to the CSP representation from that language. This challenge is mostly about finding the right abstractions over the specific domain, identifying its central pillars, and then embedding them into the CSP construction process of the expert system. Our general conclusion is that at least in some cases of complex knowledge domains, it is inherently impossible (or at least very difficult) to compromise between the CSP language and the natural description of the domain. This in spite of the fact that CSP is widely acknowledged as the most expressive satisfaction paradigm known.

Next Horizons

The research and development around the application presented here 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. A large portion of our current and future research lies in trying to find optimal description languages for our models and in developing simple, efficient, and generic CSP transformations for constructs of these languages. An additional outcome of the unavoidable coexistence of domain and CSP languages is the necessity, when the application fails to reach the expected outcome, to perform CSP analysis and debugging by domain experts, which are not necessarily CSP experts. Therefore, friendly analysis and debugging capabilities of the underlying CSP model are another major research direction we are taking. Finally, closing the verification loop by utilizing coverage-directed generation is a Holy Grail of the verification field. As our pioneering work in this area suggests (Fine and Ziv 2003), this also may be achieved by AI techniques, namely machine learning using Bayesian networks. We are actively exploring this direction.

Acknowledgments

We are grateful to Yossi Lichtenstein for participating in the writing of this article 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.

Note

1. See the 2005 IDC report, "Worldwide Server Market Shows Growing IT Investment Across Platforms, According to IDC" (www.idc.com).

References

Adir, A.; Almog, E.; Fournier, L.; Marcus, E.; Rimon, M.; Vinov, M.; and Ziv, A. 2004. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design and Test of Computers* 21(2): 84–93.

Adir, A.; Bin, E.; Peled, O.; and Ziv, A. 2003a. Piparazzi: A Test Program Generator for Micro-Architecture Flow Verification. In *Proceedings, Eighth IEEE International High-Level Design Validation and Test Workshop, HLDVT-03,* 23. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Adir, A.; Emek, R.; Katz, Y.; and Koyfman, A. 2003b. DeepTrans—A Model-Based Approach to Functional Verification of Address Translation Mechanisms. In *Proceedings, Fourth International Workshop on Microprocessor Test and Verification (MTV'03),* 3–6. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Adir, A.; Emek, R.; and Marcus, E. 2002. Adaptive Test Program Generation: Planning for the Unplanned. In *Proceedings, Seventh IEEE International High-Level Design Validation and Test Workshop, HLDVT-02,* 83–88. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

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 *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC95),* 279–285. New York: Association for Computing Machinery.

Aharoni, M.; Asaf, S.; Fournier, L.; Koyfman, A.; and Nagel, R. 2003. A Test Generation Framework for Datapath Floating-Point Verification. In *Proceedings, Eighth IEEE International High-Level Design Validation and Test Workshop, HLDVT03,* 17–22. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Behm, M.; Ludden, J.; Lichtenstein, Y.; Rimon, M.; and Vinov, M. 2004. Industrial Experience with Test Generation Languages for Processor verification. In *Proceedings of the 41st Design Automation Conference (DAC-03),* 36–40. New York: Association for Computing Machinery.

Bergeron, J. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Dortrecht, The Netherlands: 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. Constraint Hierarchies. *Lisp and Symbolic Computation* 5(3): 223–270.

Dechter, R.; Kask, K.; Bin, E.; and Emek, R. 2002. Generating Random Solutions for Constraint Satisfaction Problems. In *Proceedings, Eighteenth National Conference on Artificial Intelligence,* 15–21. Menlo Park, CA: American Association for Artificial Intelligence.

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 *Proceedings, Seventh IEEE International High-Level Design Validation and Test Workshop, HLDVT-02,* 145–150. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Fine, S., and Ziv, A. 2003. Coverage Directed Test Generation for Functional Verification Using Bayesian Networks. In *Proceedings of the 41st Design Automation Conference (DAC-03),* 286–291. New York: Association for Computing Machinery.

Freuder, E. C. 1982. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM* 29(1): 24–32.

Geller, F., and Veksler, M. 2005. Assumption-Based Pruning in Conditional CSP. In *Principles and Practice of Constraint Programming (CP 2005),* ed. P. van Beek, Lecture Notes in Computer Science Volume 3709, 241–255. Berlin: Springer.

Hentenryck, P. V. 1999. *The OPL Optimization Programming Language*. Cambridge, MA: The MIT Press.

Lichtenstein, Y.; Malka, Y.; and Aharon, A. 1994. Model-Based Test Generation for Processor Verification. In *Proceedings, Sixth Innovative Applications of Artificial Intelligence Conference,* 83–94. Menlo Park, CA: American Association for Artificial Intelligence.

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 *Proceedings, Eighth National Conference on Artificial Intelligence,* 25–32. Menlo Park, CA: American Association for Artificial Intelligence.

Nahir, A.; Ziv, A.; Emek, R.; Keidar, T.; and Ronen, N. 2006. Scheduling-Based Test-Case Generation for Verification of Multimedia SoCs. In *Proceedings of the 43rd Design Automation Conference*. New York: Association for Computing Machinery.

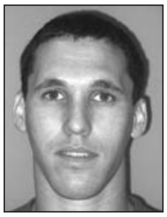
Naveh, Y. 2004. Stochastic Solver for Constraint Satisfaction Problems with Learning of High-Level Characteristics of the Problem Topography. Paper presented at the First International Workshop on Local Search Techniques in Constraint Satisfaction, Toronto, Ontario, Canada, 27 September.

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.

Wegener, I. 2000. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Yehuda Naveh (naveh@il.ibm.com) received his B.Sc. degree in physics and mathematics, M.Sc. degree in experimental physics, and Ph.D. degree in theoretical physics, all from the Hebrew University of Jerusalem, Israel. He joined IBM Haifa Research Lab in 2000, after four years of working as a research associate at Stony Brook University in New York. His current interests are the theory and practice of constraint programming and its application to hardware verification, workforce management, circuit design, and vehicle configuration.

Michal Rimon is a research staff member at the IBM research laboratory in Haifa. Her research interests include knowledge-based systems, functional verification, planning, and constraint satisfaction. Michal has a BS in mathematics and computer science from Tel-Aviv University and an MS in information systems management from the Technion, Israel Institute of Technology.



Itai Jaeger is a research staff member in the Verification and Services Technologies department at the IBM Haifa Research Lab. His research interests include functional verification, test-program generation, and constraint-satisfaction problems. Jaeger has a BA from the Technion in computer science and a BA in mathematics from the same institute.

Yoav Katz completed his B.Sc. and M.Sc. in computer science at the Technion Institute of Technology in Haifa, Israel. Since 2002, he has been working in IBM Haifa Research Lab in the Verification and Services Technologies department, developing software tools that aid hardware functional verification. Today he serves as a project lead of the team that develops Genesys-Pro, a processor-level test-case generator.

Michael Vinov is a research staff member at the IBM Research Laboratory in Haifa. His research interests include computer architectures, test-program generation, functional verification, and parallel computing. Vinov has a BS in computer engineering from Moscow Institute of Radio Technique, Electronics, and Automation and an MS in computer engineering from the Technion, Israel Institute of Technology.

Eitan Marcus received a BSc degree from Columbia University and an MSc degree from Carnegie-Mellon University, both in computer science. He joined the Simulation-Based Verification Technology Department at the IBM Haifa Research Laboratory in 1998, where he has been working on processor test-program generation, functional coverage, and constraint-based languages and systems for test generation. His other research interests include knowledge representation and reasoning.

Gil Shurek (shurek@il.ibm.com) has been a research staff member at the IBM Research Laboratory in Haifa since 1991. His research interests include ontology modeling and ontology-based software, test-program generation, shared-memory formalization, constraint satisfaction of casual constraint systems, functional verification flows and trade-offs, formal verification techniques.