Using a constraint satisfaction formulation and solution techniques for random test program generation

by E. Bin
R. Emek
G. Shurek
A. Ziv

Automatic generation of test programs plays a major role in the verification of modern processors and hardware systems. In this paper, we formulate the generation of test programs as a constraint satisfaction problem and develop techniques for dealing with the challenges we face, most notably: huge variable domains (e.g., magnitude of 2^{64}) and the need to randomly generate "well distributed" samplings of the solution space. We describe several applications of our method, which include specific test generators targeted at various parts of a design or stages of the verification process.

Functional verification is widely acknowledged as the bottleneck of the hardware design cycle.[1] Because simulation is the main vehicle for the functional verification of large and complex designs, stimuli generation for simulation plays a central role in this field. The IBM Haifa Research Laboratory (HRL) has been developing methodology and tools for stimuli generation for more than a decade. These tools aim at designs that are driven by complex micro-architectures, that support rich interface semantics, and that range from processor subunits to large multiprocessor systems.

The generated stimuli, usually in the form of test programs, are designed to trigger architecture and micro-architecture events defined by a verification plan.[2] The input for a test program generator is a specification of a test template. An example of such a test template would be a set of tests that exercise the data cache of the processor and that are formed by a series of double-word `store` and `load` instructions. The generator produces a large number of distinct well-distributed test program instances that comply with the user's specification. The variation among different instances is achieved through a large number of random decisions made during the generation process. In addition, generated test programs must meet two inherent requirements: (1) tests must be valid, that is, their behavior should be well defined by the specification of the verified system; (2) test programs should also be of high quality, in the sense that they should expand the coverage of the verified system and focus on potential bugs.

Early tools developed at HRL demonstrated a biased pseudorandom dynamic generation scheme, coupled with a traditional expert system paradigm.[3] Evolution of the tools led to the model-based test generation scheme,[4] where a generator is partitioned into a generic, system-independent engine and a model that describes the verified system. In recent years, technology has shifted toward constraint-based formulations of the generation task and generation schemes driven by solving constraint satisfaction problems (CSPs).[5]

A constraint satisfaction problem consists of a finite set of *variables* and a set of *constraints*. Each variable is associated with a set of possible values, known as its *domain*. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A *solution* to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. A *constraint network* is a hyper-graph whose nodes are the variables of a CSP and whose hyper-edges ("arcs") represent constraints; the members of an arc are those variables that appear in the constraint it represents.

Validity, quality, and test specification requirements are naturally modeled through constraints. As an example of a validity constraint, consider the case of a translation table $RA = trans(EA)$, where EA stands for the effective address and RA stands for the real (physical) address. A quality constraint may require, for example, that several `load` and `store` instructions access the same cache line, thus causing contention on resources shared between different processors. For CSP to drive test program generation, the program, or its building blocks, should be modeled as constraint networks. A random test program generator can, therefore, be viewed as a CSP solver. It constructs a CSP from the user requirements and the system model, and produces a large number of distinct program instances that satisfy the constraints.

Constraint satisfaction problems that represent test programs share several characteristics. Test program generation requires random, well-distributed solutions over the solution space,[2] as opposed to the traditional requirement of reaching a single solution, all solutions, or a "best" solution.[5] Huge domains are the result of large address spaces in modern architectures. The combination of huge domains (e.g., 2^{64} values), linear constraints (e.g., $a = b + c$) and nonlinear nonmonotonic constraints (e.g., $A = B \oplus C$, where A, B, and C are bit vectors, and $\oplus$ is the bit-wise `xor` operation) make storing and operating on these domains a difficult task. Other characteristics include a hierarchy of hard and soft constraints[6] and dynamic modeling[7] (i.e., new variables being "born" when values are assigned to other variables).

Maintaining Arc Consistency (MAC)[8] is a commonly used family of CSP solution algorithms. Arc consistency is a strong filtering (pruning) component, which allows MAC to operate well on problems with large search space. Most of our tools use refinements and variations of this scheme. We use AC-3[9] to achieve arc consistency because of its relative simplicity and because it is less affected by the domain sizes than other arc consistency algorithms. In some of our applications, we break a single, large problem, into a set of smaller loosely connected ones, and solve them separately.

We use a common set of algorithmic and modeling aids developed at HRL, known as the Generation Core toolbox. These include: (1) a mechanism for automatic construction of procedures that achieves consistency over single arcs/constraints; (2) a library of data structures for set representation, used to represent and perform various operations on variable domains.

We apply these tools to four different random test program generators, all based on CSP techniques. FP-Gen is oriented toward floating point unit verification; Piparazzi is a micro-architecture test generator; Genesys-Pro is aimed at the architectural level; and X-Gen is oriented toward the verification of an entire system.

IBM, as well as other micro-processor manufactures (AMD, STMicroelectronics, Transmeta Corporation), have gathered extensive experience using test generators developed by HRL for numerous processors and systems. This experience shows that CSP-based generators allow a compact and natural description of the verified systems and the required set of tests. Additionally, when the verified system is modified, the CSP-based generators can easily be updated. Finally, these tools achieve high coverage of the verified systems, often reaching cases that are less likely to be generated with other technologies.

The rest of the paper is structured as follows. In the section that follows, we describe related work and alternatives to CSP. Then we discuss the characteristics of CSP induced by the field of random test program generation. In the section that follows, we focus on the techniques we use for formulating the CSP and for providing solutions to it. We then describe several applications of our method, which include specific test generators targeted at various parts of a design, or stages of the verification process. The last section contains our concluding remarks.

Related work and alternatives to CSP

To date, there is no published work in the field of stimuli generation and test program generation that models the complete problem as a CSP, or that uses a CSP framework during the generation process. Reference 10 presents the results of extensive research and practice using the technique of input vector generation for production testing, a field known as Automatic Test Pattern Generation (ATPG). Traditional ATPG deals with the propagation of gate-level signals and not with high-level behavior scenarios. Similar techniques, however, may be used to explore execution paths at a somewhat higher abstraction level.[11] The method involves reducing the number of equations and variables through algebraic manipulation. In the context of test program generation, Reference 12 presents a primitive grammar for constraint expressions. The expressive power of the grammar, however, is limited, and the solution scheme deals only with unary constraints. Another test generation technology that deploys a specialized, nongeneric scheme is described in Reference 13. In Reference 14, the power of a general CSP framework is illustrated for the specific subdomain of address handling, where a MAC scheme is deployed to solve constraint-expressions in the service of a test program generator.

In addition to CSP, there are several methods that restrict the values assigned to a set of variables. Of these, linear programming,[15] SAT,[16] and Planning[17] are probably the most widely known. We believe CSP best suits the needs of random test program generation because, when compared to other methods, it imposes few restrictions on the type of variables and constraints.

Representing a test program as a SAT[16] problem is unnatural because of the large domains and the type of constraints involved. Still, we believe further research is required regarding the issue of conversion from random test generation CSPs to SAT.[18] Integer linear programming,[15] as its name implies, can naturally handle linear constraints, but dealing with other types of constraints, which are common in the domain of test generation, requires greater effort. Planning[17] is usually defined as the task of finding a series of actions that lead from a given initial state to a final goal state. Modeling and generating of test programs using planning techniques suffers from two main drawbacks. First, the vast majority of test program specifications do not include a goal state requirement. Second, most of the actions (operators) available for a complex hardware design (e.g., machine instructions in the design of processors) modify broad aspects of the verified system's state. Planning techniques are better suited for problems in which each operator modifies a small set of the system's properties.

Characteristics of CSP for test generation

In this section we describe a number of properties of the CSP technique. We show that some of these properties, particular to test program generation, raise special challenges.

Well-distributed random sampling of the solution space. The number of possible bugs in a system, and thus the number of possible scenarios that can lead to their discovery, is huge. It is impossible to exactly specify all the test programs that are needed for full coverage, and even harder to generate all the tests. This means that users of test program generators intentionally underspecify the test requirements and expect the test generators to fill in the gaps between the specification and the required tests. In other words, a test generator is required to explore the unspecified space and to help find the bugs for which the user is not directly looking.[19]

There are two ways to explore the unspecified space, systematically or randomly. A systematic approach has several drawbacks that make it impractical: good systematic exploration is possible only if the explored space is small or well-understood. Otherwise, any systematic exploration can cover only a small and unrepresentative subspace. In our case, the space to be explored is huge and a good systematic exploration requires a thorough understanding of the unspecified space—something we do not usually have.

Therefore, the best way to cover the unspecified space is to generate pseudorandom tests. That is, tests that satisfy user requirements and at the same time uniformly sample the derived test space.[4] However, finding a uniformly distributed random solution is equivalent to counting the number of possible solutions to the underlying CSP, which is known to be a hard problem.[20]

Large variable domains. Modeling random test generation as a CSP requires modeling architectural and micro-architectural resources and their content as CSP variables. These resources include registers, memory cells, and cache lines. In modern processors, these resources can take a very large number of values (e.g., 2^{64} for a 64-bit wide register). Consequently, CSP variables that represent these resources have very large domains.

These domains raise significant challenges in the representation and the solution of such problems. Obviously, it is impossible to explicitly represent a set with cardinality of 2^{64}. Explicit representation of constraints over variables with large domains is also problematic, and as in the case of variables, the constraints have to be represented implicitly. Moreover, large domains make conversion of n-ary constraints to binary constraints impractical. Finally, the size of these domains and the way in which they are represented can make the uniform selection of values difficult. For example, uniform selection of a value from a Disjunctive Normal Form (DNF) representation of a set (see the subsection "Set representation," later) requires that the clauses be disjoint, but then converting to disjoint clauses can increase their number exponentially.

Dynamic modeling of CSPs. The traditional definition of a constraint satisfaction problem contains a fixed, predefined set of variables. Many real-world problems, however, are hard to model this way, because the structure of the problem often depends on the values assigned to variables.[7] Random test generation problems modeled as CSPs often include dynamic modeling aspects. Consider, at the system level, variables representing attributes of components participating in a certain transaction. The identity of these components may depend on the values assigned to some of the variables. For example, consider a case where an access to an odd address is routed through component A, whereas an access to an even address is routed through component B.

Constraint network topology. A test program is typically comprised of a series of transactions—instructions at the processor level and interactions at the system level. Many of the variables in a CSP model of a test program represent attributes of transactions. In many cases, there is a large number of constraints among variables of the same transaction, whereas constraints among different transactions are sparse. The resulting constraint network comprises a large number of small, dense clusters, where each cluster represents a single transaction.

The term *global constraints*[21] is used to describe constraints that affect a large portion of the variables in a CSP. In spite of the clustered nature of CSPs that represent test programs, such CSPs often include constraints that affect variables from a large number of clusters. These constraints can be viewed as global constraints. For example, all the `load` and `store` instructions in a test are affected by a single constraint: any two `load` instructions that retrieve data from the same memory location must read the same value, unless a third `store` instruction between them modifies it.

Constraint hierarchy. As input, test generators accept a specification of a test template, as well as validity and quality requirements derived from the model of the verified system. For a test to be valid, its behavior should be well defined by the specification of the verified system. The simulation environment often imposes additional validity restrictions on tests. For example, it may require that all resources used by the test program are initialized. The two layers of requirements—validity and quality—derive two types of constraints: validity constraints are always mandatory, while quality constraints are not. Quite often, quality constraints contradict each other. Consider the case of a floating point add instruction `fadd` $c \leftarrow a, b$, where a, b, and c are uniformly chosen from the set of possible inputs. The probability of hitting the corner cases of $a + b = 0$, $a + b = +\infty$ or $a + b = -\infty$ is extremely low.[22] A CSP model of the `fadd` instruction would thus include three contradicting "soft" quality constraints that aim at these cases.

Extensive research was done regarding constraint systems in which not all the constraints are mandatory.[6] The term *constraint hierarchy* is often used to describe a list of sets $C_0, \ldots, C_n$ of constraints, where (A) the constraints in C_0 are mandatory, while the remaining constraints are "soft"; (B) all the constraints within a given set C_i are of equal strength; (C) constraints at a given level C_i completely dominate those at weaker levels. Given two solutions to a hierarchy of constraints, Borning et al.[6] defined several solution comparators, used to determine which of the two solutions is "better." A solution to the complete CSP is a maximum over all solutions, according to the partial order imposed by the comparator. We found the *locally predicate better* comparator useful for the domain of random test program generation. This comparator denotes that a solution x is better than y, if and only if it satisfies each constraint that y does at each level through some level k, and at least one additional constraint at level k.

Nonlinearity. Many of the constraints in a CSP that represents a test program have a nonlinear nature. Common examples include translation tables, bitwise operators, and disjunctive constraints. A translation table is typically a large array of arbitrary entries that is used to transform an input value to an output value (e.g., an effective address to a physical address). Bit-wise operators, such as `xor` and `and` arise from the corresponding instructions implemented by most processors. An example for a disjunctive constraint can be found in the dependency concept, where an instruction x depends on another instruction y if either of its registers was used by y.

Directional constraints. A constraint, as previously mentioned, is a relation over the Cartesian product of the domain of its variables. In the random test generation world, we often face directional constraints. Given one subset of the constraint's variables, it is computationally easy to calculate matching values for the other variables. However, given a different subset of the constraint's variables, calculating values for the remaining variables is either computationally or technically difficult. A constraint involving a memory address, a number of bytes, and the corresponding data illustrates the issue of directionality. Given the address and the number of bytes, finding the corresponding data is easy. Going in the other direction, from the data to the address, requires a search of a very large space.

Solution techniques

To face the challenges that arise from the characteristics just described, we developed a set of solution techniques especially suited for the random test program generation domain. Some of these techniques are adaptations of well-known algorithms (e.g., the MAC scheme). In this section, we first present a modeling scheme and environment, then we review the MAC algorithm, and finally we describe aids for data and constraint representation used to enable efficient deployment of these algorithms. Together, these tools and techniques constitute an efficient solution scheme for the vast majority of problems tackled by our test generators.

Modeling. When using the CSP for random test program generation, the first step is formulating the system and the test requirements as a CSP. This section presents two aspects of a formulation (modeling) methodology. First, we discuss how a CSP representing a test program can be partitioned into multiple subproblems to efficiently cope with large CSPs. Then we describe how these subproblems can be modeled using common building blocks and object-based techniques, thus reducing the amount of human effort involved in the modeling process.

Partitioning and abstraction. In some of our random test program generators, we chose to break up the problem of generating a single test program into a series of loosely connected subproblems. Modeling these subproblems is easier than modeling a single large problem, and solving them is computationally easier than solving the entire problem. We choose the subproblems to correspond to a single transaction: for processor architectural test generation, an instruction, and for an entire system, an interaction. We choose to partition the problem at this level for the following reasons:

- The "density" of constraints within a transaction (instruction or interaction) is much greater than the density of constraints involving different transactions (see the section "Constraint network topology").
- After every transaction, the system or processor goes through a complex stage of state update. An `add` $R_c \leftarrow R_a, R_b$ instruction, for example, updates the value of the register R_c and several other condition registers. As a result, modeling an *Execute* constraint becomes a complex task. It is much easier to solve the CSP that represents a single instruction, and then compute the next state of the system based on the singleton assignments associated with each relevant resource.
- The representation of a test program as a single CSP imposes a significant restriction on program length. Assuming CSP solution engines are limited in the number of variables, the larger the test program, the more difficult it is to treat a test program as a single CSP. For this reason, test program generators that represent the entire program as a single CSP usually generate shorter tests.

In certain applications, the CSP induced by a single transaction is still too large or complex to solve in a single step. This is primarily due to the highly dynamic nature of the constraint network that represents a transaction. Therefore, we add another axis of partition and generate every transaction in two steps. First, we solve a CSP that determines the structure of the transaction, thus eliminating most of its dynamic aspects. Only then do we construct a more detailed CSP and solve it. This approach can be viewed as a form of CSP abstraction, a known method in the literature.[23]

The partitioning approach runs into difficulties when dealing with interdependencies between the subproblems. These interdependencies typically arise when it is necessary to generate architectural and micro-architectural events that involve a large number of transactions. An event in which an instruction causes an exception that stops the execution of all other instructions currently executing is such a case. Solving the test program CSP sequentially, and ignoring constraints imposed by later instructions, may cause the generation to fail. A simple solution is to perform backtrack and return to the beginning of the current section of the test. However, this approach often fails when the problem is tightly constrained.

A better solution is to propagate the constraints imposed by later transactions to earlier transactions. We perform this by constructing a high-level intertransaction CSP that includes constraints among transactions and the variables they affect directly. Iteratively, before every transaction is generated, we reach arc-consistency over the high-level network. The domains of the CSP variables in the currently generated transaction that are shared by the high-level network are then reduced accordingly. After the transaction is generated, the corresponding variables in the high-level CSP are reduced to the single values of the transaction solution.

When generating a single transaction at a time, every transaction must be aware of the decisions made for previous ones. If, for example, register R_1 contains the value 0x1234 as a result of the first instruction, the CSP model of the second instruction must represent this fact. The large number of registers in a processor and, moreover, the huge number of memory addresses, rules out a CSP model in which every register and every memory cell has a corresponding CSP variable. Instead, we use global data structures (a map or a hash table) to maintain the values stored in registers and in memory at every stage of generation. Some of the constraints in a CSP transaction model can then use these data structures. For the above example, the constraint `synchronize register data` uses the global structure and affects two variables: the register index, which is in our case 1, and the number contained in it, which is 0x1234.

Building blocks: Increasing modeling productively. Identifying commonly used subproblems and defining them as building blocks has proved to be an effective way to simplify and accelerate the modeling process. Building blocks are specific to the application domain. While modeling the architectural structure and semantics of processor instructions as CSPs, we found operands to be useful building blocks. For example, a `Register-Operand` is a sub-CSP defined over the following variables: *register-file* (e.g., floating-point, fixed-point, condition), *register-id*, *contents*, and *sense* (which signifies whether the operand serves as the input of the instruction or as its output). The instruction `xor` $R_t \leftarrow R_a, R_b$ calculates `xor` (the *exclusive or*) of the contents of two registers of the same register file (R_a and R_b) and stores the result in R_t. The CSP model of the `xor` instruction uses three instances of the `Register-Operand` building block. Moreover, the model of almost every arithmetic and logic instruction defined by the PowerPC*[24] architecture may use the same building block.

Building blocks may also be used to construct other, more complex building blocks. For example, a `Base-Index-Operand` is a sub-CSP that specifies an operand located in main memory. The address in memory is specified using two registers and is defined to be the sum of their respective contents. A `Base-Index-Operand` is defined over the following variables: *address*, *length*, *contents*, and *sense*. It is also defined over two instances of the `Register-Operand` building block, named *Base* and *Index*. The following constraint applies: *address* = *Base.contents* + *Index.contents*.

To support efficient modeling, a building block should capture a well-defined semantic entity. It should also be possible to specify most of the relations between its variables independently of the CSP that may host it as a building block. Subtle variations may be applied to a building block by adding constraints at the usage environment.

We found that an object-based framework supports the required encapsulation; `Register-Operand` and `Base-Index-Operand` are both instances (member objects) of the class `Operand`. Every instance of the class `Instruction` is an aggregate of instances of the class `Operand`. A full-blown object-oriented framework suggests additional important modeling capabilities: Inheritance, for example, provides a powerful mechanism to construct building blocks that are variations of existing building blocks. There has been some work on object-oriented CSP modeling, for example in Reference 25.

The MAC algorithm. CSP solution algorithms can be roughly divided into two families: exhaustive search and stochastic search. Stochastic search algorithms[26] typically require a local heuristic to find solutions to hard CSPs. We found it difficult to use these methods, because in our applications local heuristics are often unknown.

Most exhaustive CSP solution methods contain the following four components: a *filtering* algorithm to prune the search tree before or during the search; a *variable ordering* algorithm to decide which variable domain is to be reduced next; a *value ordering* or *domain reduction* algorithm to decide which value(s) to examine next; a *backtracking* algorithm to use when a dead end is reached. In general, strong filtering mechanisms are better suited to problems in which the search space is large.[27] In such cases, the overhead of the pruning action is smaller than the cost of searching large branches without reaching a solution. Maintaining Arc Consistency (MAC)[8] is a family of CSP solution algorithms that use a procedure for achieving arc-consistency as its filtering component. A constraint (arc) is said to be *consistent* if, for any variable of the arc, and any value in the domain of that variable, there is a valid assignment to the other variables of the arc that satisfies the constraint. A constraint network is said to be arc-consistent if all of its constraints are locally consistent. We use the term *projection* to refer to the process of achieving consistency over a single constraint. This process reduces the domains of affected variables by removing values that cannot participate in any solution. Following is an outline of the MAC algorithm.

We choose MAC because its strong filtering component is well suited to the very large search space associated with test programs.

```
1. Achieve arc consistency over the network (e.g., execute AC-3)
   a. If any of the domains becomes empty, backtrack
   b. If all the domains are singletons—success
2. Choose a variable $x$
3. Choose a value $v \in Domain(x)$, and assign $x \leftarrow v$
4. Go to step 1
```

Most of our tools use refinements and variations on the MAC scheme with random value ordering and, in some cases, random variable ordering. We choose MAC because its filtering component is relatively strong—CSPs that represent test programs usually have a very large search space, mostly due to the large size of their domains. There are several known algorithms to achieve arc-consistency over a constraint network.[9] We choose AC-3 because of its relative simplicity and because it is less affected by the domain sizes than other arc consistency algorithms. Following is an outline of AC-3.

```
$Q \leftarrow$ {Constraints of the CSP}
while $Q$ is not empty
  Select and delete any constraint $C$ from $Q$
  Achieve local consistency over $C$ (projection)
  For each variable $v \in vars(C)$, $v$ modified by the projection
    For each $C' \neq C$, $v \in vars(C')$
      $Q \leftarrow Q \cup \{C'\}$
    Endfor
  Endfor
endwhile
```

The previously described characteristics of CSP for test program generation allow several enhancements of the basic MAC/AC-3 algorithm. The most significant of these—random solution, dynamic modeling, approximations, and constraint hierarchy—we describe below.

To reach a random solution to a CSP, we use uniform random variable ordering and uniform random value ordering. This approximates uniform distribution over the solution space. Many heuristics are based on intelligent value and variable ordering[28] and aim to direct the search to promising areas and thus, ease and speed up the search. At the same time, these heuristics reduce the ability to explore the whole search space and find uniformly distributed random solutions. Therefore, the CSP solvers used by random test generators have to balance between the use of heuristics to speed up the solution and the use of naive techniques that allow uniform exploration of the search space. Specifically, in some cases our solvers use heuristics for variable ordering, but avoid using heuristics for value ordering.

We use two approximation techniques to deal with constraints that are hard to project. We define *projection precondition* as a mechanism that associates Boolean predicates with constraints. A constraint is projected only if the predicate is evaluated to true. If the predicate is not satisfied, the projection is postponed. This approach is often used with directional constraints (see the section "Directional constraints") and is demonstrated by one of the constraints involved in a `load` instruction model. When the domain of the address variable is too large, it is impractical to project a constraint that synchronizes address and data, and we therefore have a precondition attached to the `address` variable. Our variant of the MAC algorithm is aware of constraints that have not yet been projected, and avoids instantiating the variables they affect. Another approximation method is the *inexact projection* procedure, aimed at reducing the computational complexity. An example is the constraint $a = b + c$, where the domains of a, b, and c are very large. Bounds propagation[29] is a known example for this approach. It is essential to both methods (precondition and inexact projection) that, when the domains of the affected variables become small enough in the course of the solution process, the precondition predicate is true and the projection is exact. This ensures that even though parts of the solution process are approximated, the solutions themselves are exact (e.g., an assignment provided by the solver always satisfies all the constraints in the CSP).

As previously described, CSP models of test programs often contain dynamic components, in which the structure of the problem depends on values assigned to some of the variables. To cope with this difficulty, subnetworks of the CSP are declared `conditional` and an additional Boolean existence variable is associated with them. A conditional subnetwork is part of the solution only if its corresponding existence variable is assigned true. If a constraint $C(v_1, v_2, \dots)$ affects variables from the conditional subnetwork, it is implicitly transformed into $v_e \rightarrow C(v_1, v_2, \dots)$, with v_e being the existence variable of the subnetwork. When an existence variable is instantiated to false, its corresponding conditional subnetwork is ignored. Moreover, when C cannot be satisfied, the projection procedure of $v_e \rightarrow C(v_1, v_2, \dots)$ reduces v_e to false, thus removing the entire conditional subnetwork from the CSP.

Test programs are often modeled through a hierarchy of constraints, as previously described. The MAC algorithm, with relatively minor modifications can find a solution to a hierarchy of constraints, where a solution is measured by the locally predicate better.[6] Step 1 of the MAC algorithm is performed iteratively, where at each iteration, the set of constraints Q over which consistency is achieved is expanded by one constraint, according to the algorithm below.

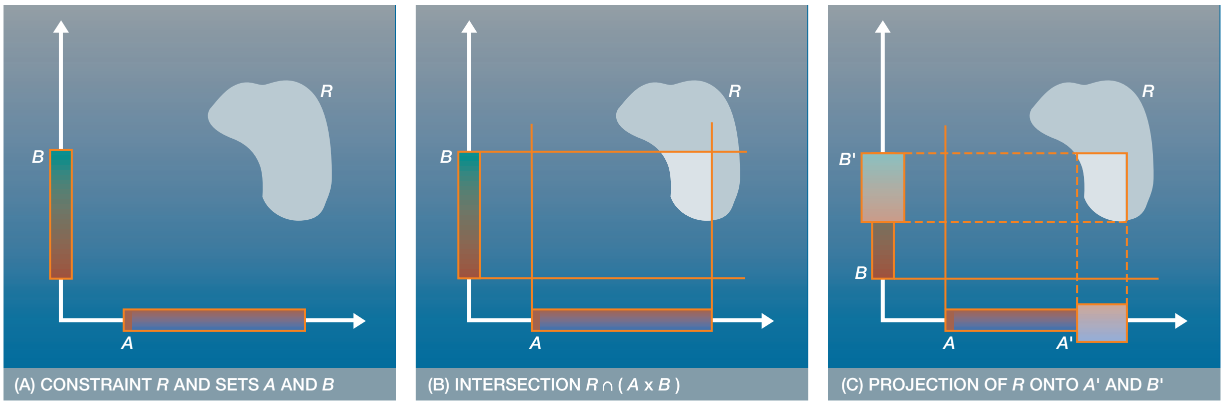
```
$Q \leftarrow C_0$ Mandatory constraints.
Using AC-3, achieve consistency over $Q$
For $l = 1 \dots n$
  While $C_l \neq \emptyset$
    $c \leftarrow$ a constraint from $C_l$
    $C_l \leftarrow C_l \backslash \{c\}$
    If AC-3($Q \cup \{c\}$) succeeds
      $Q \leftarrow Q \cup \{c\}$
    Else
      Undo the last AC-3 execution
    Endif
  Endwhile
Endfor
```

The exact algorithm that deals with a combination of constraint hierarchy, random solution, approximations and conditional subnetworks is more complex, and its precise description is beyond the scope of this paper.

Projection. The original MAC[8] algorithm was designed for binary constraints networks and uses the `Revise` procedure to achieve arc consistency on a single binary constraint. In theory, any CSP can be transformed to a network of binary constraints.[30] The transformation procedure builds a new variable for every n-ary constraint and the domain of that variable is the set of tuples allowed by the constraint. The large domains encountered when modeling test programs as CSPs make it impractical to use this transformation. Other traditional approaches to the implementation of MAC with n-ary constraints (General Arc Consistency or GAC[31]) include representing it as an explicit list of tuples, or holding relatively large data structures regarding the support of every value in a variable domain.[31] Neither of these methods is applicable to CSPs defined over very large domains. Pruning methods other than consistency (e.g., bounds propagation[29]) assume constraints are constructed only from monotonic, or even linear, operators.

We choose a different way to use MAC with n-ary constraints, by providing a consistency achieving procedure—a *projector*—for each constraint in the CSP. Such a procedure accepts a domain for each variable affected by the given constraint as input, and produces reduced domains with the following two properties: (1) *minimality*—any value which does not participate in any tuple that satisfies the constraint is removed; (2) *preservation*—all the values in the original domains that participate in a tuple that satisfies the constraint, remain in the output domains. Figure 1 demonstrates the projection of a constraint R onto two input domains, A and B, and the resulting reduced domains: A' and B'.

Figure 1 Projecting constraint *R* onto two input domains


R
B
A
(A) CONSTRAINT R AND SETS A AND B
R
B
A
(B) INTERSECTION R ∩ (A x B)
R
B'
B
A
A'
(C) PROJECTION OF R ONTO A' AND B'

The complexity of building a constraint projector is often not dependent on the linearity of the constraint. For example, projecting the constraint $a = b \oplus c$, ($\oplus$ being the bit-wise `xor` operator) is relatively easy.

Modeling and solving aids. Most of the ideas described in this section are implemented in the Generation Core toolbox. This C++ class library, developed at HRL, provides services and building blocks for the construction of stimuli generation tools. The set of services includes an advanced modeling environment consisting of an object-oriented database and a semi-visual browser. When designing a new generator, one defines a class hierarchy for the database, leading to an application-specific modeling language. This class hierarchy is derived from a set of base classes that define a generic projector interface and a generic interface for variables. A MAC solving engine, which supports preconditions, directed projectors, conditional sections of CSP networks, and a constraint hierarchy (following the ideas in the section on the MAC algorithm) is also provided. We describe below two special components: (1) a constraint projection procedure constructed automatically from an expression that defines the constraint relation; and (2) a set of classes that provide efficient representation methods for large variable domains and that support a large set of efficiently implemented operations on these domains.

Expression-driven constraint projector. We developed an algorithm for the automatic construction of procedures for projecting constraints over large domains and using nonmonotonic, nonlinear operators. By using the algorithm, we avoid the error-prone and labor-intensive process of developing a special-purpose procedure for each constraint. We use this algorithm to project a large variety of constraints.

Our algorithm accepts an expression that represents the constraint, e.g., $(a = b \times c) \vee (c = d)$, and produces a procedure for projecting the constraint. Expressions are given in a context-free grammar and include arithmetic, logical, and bit-wise operators. Apart from the definition of the grammar, the algorithm only requires a local projection function for each of the operators in the grammar. For example, a binary `plus` operator requires a ternary projection function, which operates on the two additive variables and the sum. Our experience shows that building new local projection functions, thus expanding the supported grammar, is relatively easy for many commonly used operators.

The algorithm transforms the parse tree of an expression into a constraint network: (A) every variable in the expression appears once in the new network; (B) operators in the original expression are constraints in the network; (C) additional variables represent intermediate calculations, such as the value $b \times c$ in the example above. When the resulting network is acyclic, achieving arc consistency on the network through the usage of local projection functions produces the required result. The domains are reduced to hold all, and only, the necessary values. This result can be derived from Reference 32.

We use a combination of several methods to deal with more complex constraints. First, in the case of a constraint comprised of several subconstraints joined by the *or* operator (e.g., $(a = b) \vee (b = a + c) \vee (c = 3 \cdot a)$) we project each subconstraint separately and combine the results to produce the projection of the full constraint. Consider the above constraint with the input domains $A = \{1, 2, 3\}$, $B = \{3, 4, 5\}$, and $C = \{4, 5\}$. The projection of the separate subconstraints produces the results shown in Figure 2A.

To produce the projection of the full constraint we calculate the union of the entries in every column. When a certain variable does not participate in a subconstraint, we take the original (input) domain of that variable, unless the subconstraint cannot be satisfied. In this case, we ignore the subconstraint. In the above example, our algorithm produces the results shown in Figure 2B. The sets in parentheses show the projection of the subconstraints as used during the calculation of the full constraint projection.

In other cases, an expression that produces a cyclic graph can be projected using the algorithm for the acyclic cases. If one of the variables along the cycle is assigned the same single value in every projection of the constraint, then the cycle can be broken. A typical example of this case is the constraint $(a = b) \wedge (b = c)$. Figure 3A shows the graph that represents this constraint as produced from the parse tree of the expression. The two intermediate variables x_1 and x_2 are assigned `true` in any tuple that satisfies the constraint. Therefore, the cycle can be broken by separating x_1 into two nodes with singleton domains `true`, as shown in Figure 3B.

For more complex constraints we try to decompose the graph that represents the expression into a set of acyclic components by finding cycle cut-set variables.[33] When the cross product of the cycle cut-set variables domains is small enough, our algorithm iteratively assigns them every possible combination of values. Doing so, we break the full constraint into a series of disjunctive subconstraints. If, for example, we iterate on the domains $D_a = \{1, 2, 3\}$ of the variable a, the original constraint C is replaced by $(C \wedge a = 1) \vee (C \wedge a = 2) \vee (C \wedge a = 3)$. At least one of the variables in each cycle of the subconstraints has a singleton variable. These subconstraints can, therefore, be projected using the acyclic projection algorithm. We combine the projection of the disjunctive subconstraints using the join algorithm depicted above.

Set representation. The method used to represent the domains of the variables of a CSP is one of the main

Figure 2 Projecting a disjunctive constraint

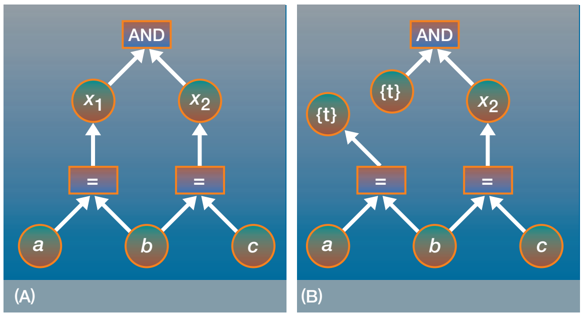
SUBCONSTRAINT	*A*	*B*	*C*
$a = b$	{3}	{3}	–
$b = a + c$	{1}	{5}	{4}
$c = 3 \cdot a$	{ }	–	{ }

(A)

SUBCONSTRAINT	*A*	*B*	*C*
$a = b$	{3}	{3}	({4,5})
$b = a + c$	{1}	{5}	{4}
$c = 3 \cdot a$	{ }	({ })	{ }
RESULTS	{1,3}	{3,5}	{4,5}

(B)

Figure 3 Singleton variables in a cycle



factors that affect the efficiency of a CSP solution algorithm. The representation method affects the space needed to store the variables and the efficiency of performing operations on them. Specifically, the MAC algorithms with random solutions used by our random program test generators present the following requirements: (A) support representation of very large variable domains; (B) be compact to allow efficient use of the memory space of the program; (C) support efficient operations required by the MAC algorithm to perform projection, and (D) support an efficient uniformly distributed random selection of an element from the domain (this operation is needed to find a random solution for the CSP).

The techniques we use to model random test generation as CSPs lead to the use of many types of variables with domain size ranging from very small (e.g., 2) to very large (e.g., 2^{64}). We developed a class library that supports the variable types commonly used in our CSP. For each of the types supported by this class library, the library offers one or more efficient ways to represent domains (sets of values) of this type, and basic operators that correspond to operations often performed on these types. Currently, the library supports integers, bit-vectors, Booleans, and strings.

Integers and bit-vectors are the two most commonly used types in our CSPs. These two types are often interchangeable, since the binary representation of an integer can be viewed as a bit-vector, whereas the binary number represented by a bit-vector can be viewed as a signed or unsigned integer. Supported operations can be divided into three main groups: set operations, such as union, intersection and random selection of elements; arithmetic operations, such as addition and multiplication; and bit-wise logic operations, such as bit-wise `xor`. The arithmetic operations are more natural to integer domains, whereas the bit-wise operations are more natural to the bit-vector domains. Still, both types need to support both sets of operations.

There are several possible methods for implicit representations of a set of values over bit-vectors and integers. Unfortunately, none is efficient for all operations. The representation methods currently used in our class library are:

Set of ranges—The values in a domain are represented as a set of nonoverlapping intervals (e.g., {(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" values for some of its bits. It 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 (i.e., all the numbers divisible by 4). Masks are very effective representations for bit-wise operations and can be used efficiently in set operations with other masks. They are also convenient as input media. However, performing arithmetic operations on masks is hard.

Binary Decision Diagrams (BDDs)—Binary Decision Diagrams[34] are data structures commonly used to represent Boolean functions. A BDD represents a set of values in a domain through the characteristic function of the set ($member_D(x) = 1$ *iff* $x \in D$). Some operations performed on integers and bit-vectors can be done very efficiently on BDDs (e.g., set operations). BDDs also have efficient algorithms for some arithmetic operations, such as addition.[34] BDDs, on the other hand, are less efficient than masks in handling bit-wise operations.

The representation method of each variable is selected by the modeler of the CSP, based on the operations in which the variable is involved. When an operation is performed on two variables with different representations, the CSP engine converts the representation to the one most efficient for the operation. In general, we try to avoid conversions between representation methods due to their cost.

Applications

We developed a set of random test program generators, based on the solution framework described in the previous section. These generators are targeted at different components of the verified design and are used during various stages of the verification process.

We focus in this section on Genesys-Pro, a multiprocessor-oriented architectural-level test program generator. We briefly describe three other generators: FP-Gen, a generator dedicated to floating-point-unit verification; Piparazzi, oriented toward the verification of the control flow of a single processor; and X-Gen, oriented toward verifying an entire system. The tools have different verification scopes, and the abstraction level in each one of them is set accordingly: tools with broader verification goals have a less-detailed understanding of the verified design.

Most of these generators are model-based. That is, they are partitioned into a generic, system-independent part, and a model that describes the verified design. The modeling language of each tool is designed to suit its specific verification needs. For example, the modeling language of X-Gen contains system-level terms and concepts, such as components and the connections between them, whereas the modeling language of Genesys-Pro focuses on processor-related concepts such as instructions and registers.

When designing a CSP-based random test program generator, there is an inherent tension between the "problem language" and the "solution language." The vocabulary of verification engineers includes words such as registers, instructions, and pipelines, and they prefer this terminology for requirements

Table 1 Size properties of typical CSPs

			Piparazzi	Genesys-Pro	X-Gen
Subproblem	**Variables (by domain)**	<10	1433	45	5
		10..1000	406	11	42
		$1000..2^{64}$	254	19	30
		$2^{64} <$	27	1	8
		Total	2120	76	85
	Constraints (by arity)	≤ 10	4903	49	96
		10..100	813	2	
		$100 <$	188	(1)	(1)
		Total	5904	51	96
	Run time (Sec.)		399	0.17	0.45
No. of Subproblems			3	100	3250
Complete CSP	**Variables (by domain)**	<10	4299	4500	9500
		10..1000	1218	1100	137500
		$1000..2^{64}$	762	1900	100000
		$2^{64} <$	81	100	25500
		Total	6360	7600	272500
	Constraints (by arity)	≤ 10	14709	4900	312000
		10..100	2439	200	
		$100 <$	564	(1)	(1)
		Total	17712	5100	312000
	Run time (Min.)		20	0.58	24

description. CSP modelers, on the other hand, must consider the need to efficiently represent the test program as a CSP. Therefore, they use the terminology of constraints and variables.

The size properties of typical CSPs solved by these generators are displayed in Table 1. We display the total number of variables and constraints in the CSP, as well as information about the size of a typical subproblem (see section "Partitioning a CSP," above). Constraints are classified according to their arity and variables according to the size of their domains. Run time was measured on a 375 MHz IBM RS/6000* machine.

Processor architecture. Genesys-Pro is the main test program generation tool for verification of PowerPC processors in many of IBM's development laboratories. The model used by Genesys-Pro contains a description of the functional specification of the processor (i.e., its architecture). This description includes the instruction set of the processor, its architectural resources (e.g., registers), and global rules of behavior (e.g., the value read from a register is the last value written to it). For each instruction in the instruction set, the description contains a list of operands, the resources it reads or modifies, and its behavior. For the entries marked (1) in Table 1, Genesys-Pro and X-Gen break a single problem into multiple constraint networks (subproblems). Some of the constraints have a relatively small arity when viewed in the context of a single constraint network, but use global data structure that represent relationships with hundreds or thousands of variables from other networks.

The model for each instruction contains a constraint network that describes the relations between the various operands and the resources the instruction reads and modifies. For example, the model of a `load` instruction contains a constraint specifying that the memory address accessed by the instruction is equal to the sum of the content of a base register, whose number is the second operand of the instruction, and a displacement, which is the third operand (see an example later in this section). The constraint networks of the instructions, combined with the global behavior rules, provide the validity requirements of a generated test program.

Table 2 CSP model of a "load" instruction—variables

	Address	Data-in	Data-out
Register x	*x.addr*: (0 . . 31)	*x.in* (64-bits)	*x.out* (64-bits)
Register y	*y.addr*: (0 . . 31)	*y.in* (64-bits)	*y.out* (64-bits)
Displacement	—	*disp.in* (16-bits)	—
Memory	*mem.addr*: (64-bits)	*mem.in* (64-bits)	*mem.out* (64-bits)

In addition to the model of the verified processor, Genesys-Pro has two other components: testing knowledge and user directives. The testing knowledge specifies generic events of interest used to enhance the quality of the generated tests (e.g., the result of an `add` instruction is zero). The testing knowledge is described as constraints on the resources of an instruction and their values, or as constraints between resources and values of several instructions in the test.

Through its user interface, Genesys-Pro allows user directives that can range from specific requirements, leading to generation of an event, to general directives that attempt to provide wide coverage to some part of the design. Among other things, the user directives include specification of the instructions that need to be generated, and guidelines on how, and how often, to use the testing knowledge.

Because Genesys-Pro is required to generate long tests of hundreds or thousands of instructions, it cannot solve the entire test-wide CSP as a single CSP network. Therefore, it uses the following generation scheme: Genesys-Pro generates a test program one instruction at a time, where the instructions are generated in the order of their execution. The generation of each instruction in the test consists of the following steps:

1. Selecting the instruction to be generated
2. Building the CSP network for the instruction. The network includes constraints from the model of the instruction, specific user directives for the instruction, projection of global constraints on the instruction, and constraints from the testing knowledge according to the user directives and the guidelines in the testing knowledge. Usually, constraints from the testing knowledge are soft constraints.
3. Solving the CSP for the generated instruction and updating a reflection of the processor state accordingly

Example: A CSP model of an instruction. Consider the case of a `load` instruction of the form `load` $R_x \leftarrow R_y(disp)$. The instruction loads the data at memory address $[R_y] + disp$ into the register R_x, where $[R_y]$ is the value held in the register whose index (address) is y. Table 2 defines the variables in a simplified CSP model representing the instruction. We model four sets of variables: register x, register y, displacement, and memory. Each set (except for displacement) contains three variables: an address, the data before the instruction is executed (data-in) and the data after the execution (data-out). Variable domains are given in parentheses. The constraints in Table 3 must be satisfied for the instruction to be valid. A more realistic CSP model of a `load` instruction would contain a larger number of variables and validity constraints, as well as some quality (soft) constraints.

To solve this kind of constraint network, we use some of the solution techniques described in the previous section. The constraint network itself represents only a single instruction, whereas the CSP solved by Genesys-Pro represents a complete test program, often with hundreds or thousands of instructions. This demonstrates the partition principle previously described. We use the MAC-based algorithm described in a previous section to solve this part of the CSP by providing a constraint projector for each of the constraints. Some of these should be especially noted: (A) Due to the huge domains of the variables involved, the projection of $mem.addr = y.in + disp.in$ is approximated. When the domains of two of the three variables become small enough (in the course of the solution process), the projection can be performed precisely. (B) The last three constraints in the table form the relationship between this instruction and previous ones. They use global data structures to access the resources values (memory, registers) resulting from previous instructions. (C) The constraint `Initial_value` (*mem.addr*, *mem.in*) has a *precondition* on the variable *mem.addr*. As long as the domain of this variable is too large, we cannot project the constraint because it is practically impos-

Table 3 CSP model of a "load" instruction—constraints

Constraint	Description
$mem.addr = y.in + disp.in$	The memory address is the sum of the displacement and the value held in register y.
$x.out = mem.in$	The functionality of the instruction: the content of the memory is loaded into R_x.
$mem.in = mem.out$	The memory is not modified by the instruction.
$x.addr = y.addr \rightarrow (x.in = y.in \wedge x.out = y.out)$	If R_x and R_y are the same register, they contain the same input and output data.
$x.addr \neq y.addr \rightarrow y.in = y.out$	If R_x and R_y are not the same register, R_y is not modified by the instruction.
$mem.addr \in A \cup B$, where $A = [0x0 \ldots 00000$ to $0x0 \ldots 01FFF]$ and $B = [0x0 \ldots 12000$ to $0x0 \ldots 2C000]$	The memory address must be within a given physical address space.
$mem.addr \bmod 4 = 0$	The memory address must be aligned to a 4-byte boundary.
`Initial_value`$(x.addr, x.in)$	Some of the registers may already be initialized from previous instructions. Therefore, the combinations of register-address and register-value are restricted by these initializations.
`Initial_value`$(y.addr, y.in)$	Same as above.
`Initial_value`$(mem.addr, mem.in)$	Same for memory.

sible to iterate over a huge number of memory addresses and check their contents.

The domains of the variables are represented using the techniques described in the section "Set representation." Specifically, we would usually use either DNF or BDD representation to model variables of very large domain size.

The constraint network representing a `load` instruction is a single subproblem solved by Genesys-Pro. The generator solves one instruction at a time, and updates the global data structures that represent interinstruction constraints. Together, this series of instructions constitutes a test program.

Floating-point unit. FP-Gen is a focused tool designed to bias and generate operand data for the IEEE (Institute of Electrical and Electronics Engineers) standard floating-point instructions.[35] A bias (or constraint) on the data of an operand is a set of values to which the operand data are constrained. Resolving constraints on input operands is relatively straightforward, even though the requirement of uniformity among all the solutions is sometimes hard to obtain. In contrast, resolution of constraints on the data for both the intermediate result(s) and the result of instructions adds a layer of complexity that involves instruction semantics. The algorithms of FP-Gen are based on a thorough case analysis,[22] which spans CSP stochastic methods, dedicated analytical conversions, and linear programming.

Micro-architecture control flow. Piparazzi is designed for the verification of the micro-processor control. Modern micro-processors have several micro-architectural mechanisms that improve performance, but increase the complexity of the design, thereby increasing the risk of bugs. Examples of such mechanisms include super-scalar, out-of-order, pipelining, caching, and remapping of resources.[36] Piparazzi's input language lets the user specify micro-architectural events, for example two instructions are executed simultaneously in a specific pipeline. The CSP constructed and solved by Piparazzi represents various micro-architectural aspects of the instructions and of the micro-architectural mechanisms of the processor. The solution algorithm used by Piparazzi is MAC-based, where multiple instructions are solved as one constraint network.

System level. X-Gen is targeted toward system level verification. In this context, a system is a collection of components that operate together—such as processors, buses, memories, caches, bridges, and I/O devices. A model of the verified system comprises three aspects: (1) component types, such as memory, CPU, or a certain bus-bridge; (2) a configuration file that describes the instances of these component types (e.g., CPU-0 and CPU-1) and the topology of the system—logical and physical connections between components; and (3) interactions, which describe the way components operate together. For example, an interrupt sent from an I/O device to a CPU is a type of interaction. The CSP solved by X-Gen contains variables and constraints that represent the above three aspects. It is partitioned along two axes: every interaction is generated separately and broken into two constraint networks. An abstract network determines the identity of the components participating in the interaction, and a concrete network then determines the actual attributes such as address and data.

Conclusions

In this paper we have shown how random test program generation can be modeled as a CSP. We described the common characteristics of the CSPs that represent test programs, some of which are unique to this domain. To address the challenges that arise from these characteristics, we developed a set of solution techniques that address all aspects of the problem, from modeling techniques via the solution algorithms to data representation methods.

This set of techniques is the basis for several random test program generation tools that the IBM Research Laboratory in Haifa has developed and continues to develop. These tools range from dedicated tools for specific units to general-purpose tools designed to test processors and multi-processor systems. The CSP modeling and solution framework simplifies the work of developers of new tools and allows them to concentrate on the unique characteristics and requirements of the tool.

We continue to look for new techniques and algorithms to improve our CSP modeling and solution framework for random test program generation and provide better services to tool developers. The solution engine is the main area in which we are looking for methods to improve our current techniques. Specifically, we are looking at ways to combine our MAC-based search algorithm with other search techniques, such as hill-climbing algorithms[37] and SAT-based solvers,[16] that are used to solve local subproblems. We are also looking at ways to incorporate incremental techniques[38] and techniques that take advantage of knowledge gained in a solution of one problem to solve similar problems.

Another possible direction to improve the search algorithm is the use of constraint logic programming (CLP) engines, such as CHIP,[39] as search engines, either for the whole CSP or for specific subproblems. Some of the difficulties we foresee with this approach are the ability of CLP engines to handle the large domains that characterize random test program generation and the representation of complex constraints over them.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers, Boston, MA (January 2000).
2. L. Fournier, Y. Arbetman, and M. Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator," *Proceedings of the Design Automation and Test in Europe Conference* (DATE99), Munich, March 1999, ACM, New York (1999), pp. 434–441.
3. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwatzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM Systems Journal* **30**, No. 4 (1991), 527–538.
4. A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM," *Proceedings of the 32nd Design Automation Conference* (DAC95), IEEE, New York (1995), pp. 279–285.
5. V. Kumar, "Algorithms for Constraint-Satisfaction Problems: A Survey," *A.I. Magazine* **13**, No. 1 (1992), 32–44.
6. A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf, "Constraint Hierarchies," *Proceedings of OOPSLA'87*, ACM, New York (1987), pp. 48–60.
7. S. Mittal and B. Falkenhainer, "Dynamic Constraint Satisfaction Problems," *Proceedings of AAAI'90*, American Association for Artificial Intelligence, Menlo Park, CA (1990), pp. 25–32.
8. A. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence* **8**, No. 1 (1977), 99–118.
9. C. Bessiere, E. C. Freuder, and J.-C. Regin, "Using Constraint Metaknowledge to Reduce Arc Consistency Computation," *Artificial Intelligence* **107**, No. 1 (1999), 125–148.
10. M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*, IEEE, New York (1995).
11. J. Lee and J. Patel, "Architecture Level Test Generation for Microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **13**, No. 10 (1994), 1288–1300.
12. A. J. Offutt, "An Integrated Automatic Test Data Genera-

tion System," *Journal of System Integration* **1**, No. 3 (1991), 391–409.
13. A. Chandra and V. Iyengar, "Constraint Solving for Test Case Generation—A Technique of High Level Design Verification," *Proceedings of the IEEE International Conference on Computer Design* (ICCD), IEEE, New York (1992), pp. 245–248.
14. L. Fournier, D. Lewin, M. Levinger, E. Roytman, and G. Shurek, "Constraint Satisfaction for Test Program Generation," *Proceedings of the IEEE 14th Annual International Phoenix Conference on Computers and Communications*, IEEE, New York (1995), pp. 45–48.
15. V. Chvátal, *Linear Programming*, W. H. Freeman and Company, New York (1983).
16. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," *Satisfiability Problem: Theory and Applications*, D. Du, J. Gu, and P. M. Pardalos, Editors, Volume 35 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Providence, RI (1997), pp. 19–152.
17. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).
18. T. Walsh, "SAT v. CSP," *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming* (CP'00), *Lecture Notes in Computer Science* series, Vol. 1894, Springer-Verlag, New York (2000), pp. 441–456.
19. A. Hartman, S. Ur, and A. Ziv, "Short vs Long: Size Does Make a Difference," *Proceedings of the IEEE High Level Design Validation and Test Workshop* (HLDVT'99), IEEE, New York (1999).
20. M. R. Jerrum, L. G. Valiant, and V. V. Vazirani, "Random Generation of Combinatorial Structures from a Uniform Distribution," *Theoretical Computer Science* **43**, No. 2–3 (1986), 169–188.
21. N. Beldiceanu and E. Contjean, "Introducing Global Constraints in CHIP," *Mathematical and Computer Modelling* **20**, No. 12 (1994), 97–123.
22. L. Fournier and A. Ziv, "Solving the Generalized Mask Constraint for Test Generation of Binary Floating Point Add Operation," *Theoretical Computer Science*, special issue on Real Numbers and Computers (to appear).
23. T. Ellman, "Abstraction by Approximate Symmetry," *IJCAI'93: Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers, San Francisco, CA (1993), pp. 916–921.
24. *The PowerPC Architecture*, C. May, E. Silha, R. Simpson, and H. Warren, Editors, Morgan Kaufmann Publishers, San Francisco, CA (1994).
25. M. H. Sqalli and E. C. Freuder, "Constraint-Based Modeling of Interoperability Problems Using an Object-Oriented Approach," *Proceedings of the Thirteenth Innovative Applications of Artificial Intelligence Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Menlo Park, CA (2001).
26. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems," *Artificial Intelligence* **58**, No. 1–3 (1992), 161–205.
27. C. Bessière and J.-C. Régin, "MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems," *Proceedings of the 2nd International Conference on Principles and Practices of Constraint Programming* (CP'96); *Lecture Notes in Computer Science* series, Volume 1118, Springer-Verlag, New York (1996), pp. 61–75.
28. D. Frost and R. Dechter, "Look-Ahead Value Ordering for Constraint Satisfaction Problems," *IJCAI'95: Proceedings of the International Joint Conference on Artificial Intelligence*, C. Mellish, Editor, Montreal, August 1995, Morgan Kaufmann Publishers, San Francisco, CA (1995).
29. W. Harvey and P. J. Stuckey, "Constraint Representation for Propagation," *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming* (CP'98), *Lecture Notes in Computer Science* series, Volume 1520, Springer-Verlag, New York (1998), pp. 235–249.
30. F. Bacchus and P. van Beek, "On the Conversion Between Non-Binary and Binary Constraint Satisfaction Problems," *Proceedings of the 15th National Conference on Artificial Intelligence* (AAAI-98) *and of the 10th Conference on Innovative Applications of Artificial Intelligence* (IAAI-98), AAAI Press, Menlo Park, CA (1998), pp. 311–318.
31. C. Bessière and J.-C. Régin, "Arc Consistency for General Constraint Networks: Preliminary Results," *Proceedings of the IJCAI*, Morgan Kaufmann Publishers, San Francisco, CA (1997), pp. 398–404.
32. E. C. Freuder, "A Sufficient Condition for Backtrack-Free Search," *Journal of the Association for Computer Machinery* **29**, No. 1 (1982), 24–32.
33. R. Dechter and J. Pearl, "Network-Based Heuristics for Constraint-Satisfaction Problems," *Artificial Intelligence* **34**, No. 1 (1988), 1–38.
34. I. Wegener, *Branching Programs and Binary Decision Diagrams: Theory and Applications*, Siam Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, Philadelphia, PA (2000).
35. *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 745-1985, American National Standards Institute, Washington, DC (1985).
36. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, San Francisco, CA (1997).
37. E. Tsang, C. Wang, A. Davenport, C. Voudouris, and T. Lau, "A Family of Stochastic Methods for Constraint Satisfaction and Optimization," *Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming* (PACLP'99), London, April 1999, The Practical Applications Company Ltd. (1999).
38. B. N. Freeman-Benson, J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Communications of the ACM* **33**, No. 1 (1990), 54–63.
39. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, "The Constraint Logic Programming Language CHIP," *FGCS-88: Proceedings of the International Conference on Fifth-Generation Computer Systems*, Tokyo, December 1988, Springer-Verlag, New York (1988), pp. 693–702.

Accepted for publication March 25, 2002.

Eyal Bin *IBM Research Division, Haifa Research Laboratory, Haifa 31905, Israel (electronic mail: bin@il.ibm.com).* Mr. Bin is a graduate of the Technion, Israel Institute of Technology, where he received a B.Sc. degree in computer engineering in 1988, and an M.Sc. degree in electrical engineering in 1991. Mr. Bin joined the Haifa Research Lab in 1991 as a research staff member, and was involved in physical design technologies, including automatic layout generation and waveform planing. Since joining the Verification Technologies Department in 1999, Mr. Bin has led the micro-architecture test generator activity for microprocessors in IBM. His other research interests include constraint satisfaction problems and self-stabilizing distributed systems.

Roy Emek *IBM Research Division, Haifa Research Laboratory, Haifa 31905, Israel (electronic mail: emek@il.ibm.com).* Mr. Emek joined the Haifa Research Lab as a research staff member in 1998. Currently, he is the technical lead for X-Gen, a system-oriented random test program generator. His main research areas include hardware verification and constraint satisfaction techniques.

Gil Shurek *IBM Research Division, Haifa Research Laboratory, Haifa 31905, Israel (electronic mail: shurek@il.ibm.com).* Mr. Shurek has been a research staff member at the Haifa Research Laboratory since 1991. He received his B.Sc. degree in computer engineering and his M.Sc. in computer science from the Technion, Israel Institute of Technology, in 1987 and 1991, respectively. His research interests include test program generation, shared memory models, constraint satisfaction, object-oriented modeling, and formal verification techniques.

Avi Ziv *IBM Research Division, Haifa Research Laboratory, Haifa 31905, Israel (electronic mail: aziv@il.ibm.com).* Dr. Ziv received his B.Sc. degree in computer engineering from the Technion, Israel Institute of Technology, in 1990, and his M.Sc. and Ph.D. degrees in electrical engineering from Stanford University, in 1992 and 1995, respectively. Since joining the Verification Technologies Department at the IBM Haifa Research Lab as a research staff member in 1996, Dr. Ziv has been involved in several activities in the fields of testing and reliability of software and hardware systems, and the high-level modeling of hardware systems. His other research interests include fault-tolerant computing and parallel and distributed systems.