Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification

Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv
IBM Research Lab, Haifa

Editor's note:
Historically, IBM's random test-program generation (RTPG) methodology has tightly coupled architectural information with the TPG tool. Model-based TPG removes this architecture dependency and provides a generic solution to functional testbench generation. Genesys-Pro, the second-generation model-based TPG tool, has many improvements over its predecessor, Genesys, including greater expressive power in the test template language and more constraint-solving processing power.
—Li-C. Wang, University of California, Santa Barbara

FUNCTIONAL VERIFICATION is widely recognized as the bottleneck of the hardware design cycle. With the ever-growing demand for greater performance and faster time to market, coupled with the exponential growth in hardware size, verification has become increasingly difficult. Although formal methods such as model checking[1] and theorem proving have resulted in noticeable progress, these approaches apply only to the verification of relatively small design blocks or to very focused verification goals. In current industrial practice, simulation-based techniques play a major role in the functional verification of microprocessors.[2,3]

The recent emergence of hardware verification languages and comprehensive environments designed to automate the functional verification process[4] has significantly affected simulation-based technology. Engineers typically use these environments to verify ASICs, SoCs, and unit-level components in a processor. Using such environments to verify large processors (x86, PowerPC, and so on) still requires significant effort.

Current industry practice is to use separate, automatic, random stimuli generators for processor- and multiprocessor-level verification. The generated stimuli, usually in the form of test programs, trigger architecture and microarchitecture events defined by a verification plan.[5] In general, test programs must meet the validity requirement (the tests' embedded behavior should conform to the targeted design's specification) and the quality requirement (the tests should expand the targeted design's coverage and increase the probability of bug discovery). The generator can produce many distinct, well-distributed test program instances that comply with user requests. Numerous random selections made during generation achieve the variation among different instances.

The first generation of test generators at IBM was developed in the mid-1980s. These generators incorporated a biased, pseudorandom, dynamic generation scheme.[6] The need for a generic solution, applicable to any architecture, led to the development of a model-based test generation scheme that partitions the test generator into two main components: a generic, architecture-independent engine and a model that describes the targeted architecture.[7] In 1991, IBM developed the first model-based pseudorandom test program generator, Genesys.[7] Widely used during the past decade, both

0740-7475/04/$20.00 © 2004 IEEE Copublished by the IEEE CS and the IEEE CASS

inside and outside IBM, Genesys has proved the model-based approach's strength and versatility on different architectures and designs.

This article describes Genesys-Pro, IBM's third-generation test generator for the functional verification of microprocessors. Genesys-Pro relies on the same underlying model-based approach as Genesys but has three significant advancements. First, the language that describes the template of the test to be generated has the expressiveness of a programming language. It lets users define any desired verification scenario while leaving noncritical parameters unspecified. This allows virtually unlimited control over the events to be generated. The second advantage is Genesys-Pro's framework for modeling processor architectures. In addition to the standard modeling constructs present in other modeling environments, Genesys-Pro provides high-level building blocks specifically suited for describing processors. This combination offers the flexibility to model the architecture's complex and procedural aspects, such as very large instruction words (VLIWs) and various address translation mechanisms.

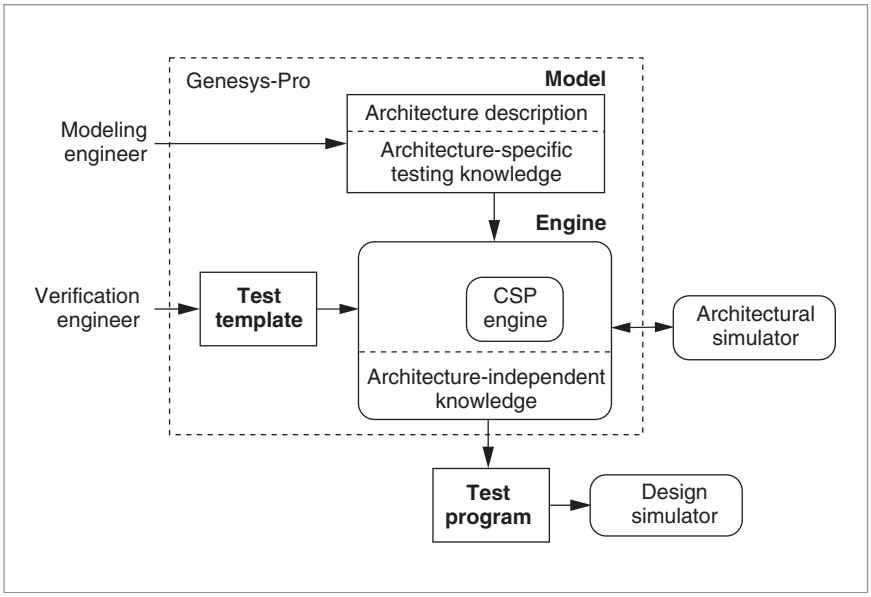

Figure 1. General structure of the Genesys-Pro random test generator.

The third major advancement is the powerful generation engine. The generator translates the test generation problem into a constraint satisfaction problem (CSP) and uses a generic CSP solver customized for pseudorandom test generation[8] to increase the probability of generation success and to improve test program quality. The CSP framework is the backbone of all recently developed IBM test generators.

These three components—the test template language, the modeling framework, and the generation engine—have all been specifically tuned for verifying processors. This is a major difference between Genesys-Pro and other commercial testing environments, such as Vera, *e*, and the System-C Verification Library. These environments provide powerful high-level languages that incorporate multiple programming paradigms. However, because they are not specific to the processor verification domain, verifying processor architectures demands greater effort than with Genesys-Pro.

Genesys-Pro overview

Genesys-Pro's architecture is based on a clear delineation of three types of knowledge relevant to test program generation for processor verification. First, there's the generic engine with knowledge about general processor architectures and general test generation techniques. Then there's an architectural model that contains processor-specific information. This principally includes a declarative description of a specific processor architecture and a database of testing knowledge relevant to that processor. The testing knowledge describes aspects of the architecture and is used to create high-quality tests. Following the model-based approach, this architecture-specific model remains separate from the generic generation engine. Finally, test templates describe specific scenarios meant to cover the tested processor's verification plan.

Figure 1 is an overview of Genesys-Pro. A verification engineer provides a test template of the scenario that should occur in the test program. Genesys-Pro generates a test by formulating and then solving a separate constraint problem for each test instruction. The constraint problem is based on constraints that originate from both the architectural description and the testing knowledge, and Genesys-Pro annotates the constraints according to directives in the test template. Constraints can be either mandatory (hard) or nonmandatory (soft). Constraints originating from the architectural description are typically mandatory. Nonmandatory constraints are those that the CSP engine tries to solve but might give up on if it fails to find a solution. Users can set testing-knowledge constraints and constraints originating from the test template as mandatory or nonmandatory.

Test program template

```
Variable: addr = 0x100
Variable: reg
Bias: Resource-Dependency(GPR) = 30
Bias: Alignment(4) = 50

Instruction: Load R5 ← ?
     Bias: Alignment(16) = 100
Repeat (addr < 0x200)
   Instruction: Store reg → addr
   Select
      Instruction: Add ? ← reg + ?
         Bias: SumZero
    Instruction: Sub ? ← ? - ?
   addr = addr + 0x10
```

Test program

```
Resource Initial Values:
   R6 = 8, R3 = - 25,..., R17 = - 16
   100 = 7, 110 = 25,..., 1F0 = 16

Instructions:
   500:   Load R5 ← FF0
   :
   504:   Store R4 → 100
   508:   Sub R5 ← R6 - R4
   50C:   Store R4 → 110
   510:   Add R6 ← R4 + R3
   :
   57C:   Store R4 → 1F0
   580:   Add R9 ← R4 + R17
```

Figure 2. Test program template and the corresponding test.

After formulating the CSP, the generation engine solves it by using a dedicated CSP engine appropriate for the type of problems common in test program generation.[8] After generating each instruction, Genesys-Pro sends it to an architectural simulator for simulation. Thus, the generator can maintain an accurate view of architectural resources, which is essential for resolving subsequent constraints and for producing expected results for each resource involved in the test.

The generated test program passes to a design simulation environment, which runs the program and looks for mismatches between the results specified in the test and the actual results produced by the design simulator. The design simulation environment also uses other means of detecting violations, such as assertions and coherency monitors. Coverage data collected during simulation helps to monitor the progress of the verification process.[4]

Test template language

The test generator lets users describe scenarios required for the verification test plan. Genesys-Pro has a highly expressive input language for writing such scenarios. The language consists of four types of statements: basic instruction statements, sequencing-control statements, standard programming constructs, and constraint statements. Users combine these statements to compose complex test templates that, in varying detail, describe the test scenarios. The language lets users control the degree of randomness in the generated tests, from completely random to completely directed. In most cases, the template balances both modes, explicitly formulating the essential parts of what is to be tested while leaving the rest unspecified. The generation engine translates these verification scenarios into complete test programs.

Figure 2 shows an example of a test template and a corresponding generated test. The test template describes a table-walk scenario that stores the contents of a randomly selected register into memory addresses ranging from 0x100 to 0x200, in increments of 16 bytes. An Add or a Sub instruction follows each Store. The first source register used for each Add instruction is the same as the source register of the previous Store. Additionally, the template requests several bias constraints controlling such characteristics as interdependency between instructions and the alignment of addresses.

Basic instruction statements

Instruction statements specify which instructions to place in the generated test. For example, the template in Figure 2 includes four basic instruction statements for Load, Store, Add, and Sub. Users can also control various properties of the instruction, such as which resources to use. These properties can be randomly selected (indicated by a "?" in the template), equal to some specific value (for example, R5), or dependent on a previous value according to user-defined variables.

Sequencing-control statements

Sequencing control consists of the following five statements:

- *Sequence* directs Genesys-Pro to generate a list of ordered substatements.
- *Select* directs Genesys-Pro to generate a single substatement randomly selected from a list of substatements according to a weight attribute.
- *Repeat* causes the generator to repeat substatements a given number of times or as long as a specified repeat condition is satisfied. In the example, the **`Repeat`** statement directs the generator to keep generating the pairs of Store and Add or Sub instructions for as long as the address variable **`addr`** is less than 0x200.
- *Permute* directs Genesys-Pro to generate a list of substatements in a randomly selected order.

- *Concurrent* directs Genesys-Pro to generate instruction streams for each processor or thread in a multiprocessor configuration. The instruction streams will execute concurrently.

Standard programming constructs

Standard programming constructs include variable definitions, assignment statements, expressions, and assertions. Genesys-Pro supports typed, scoped variables, including arrays. A variable's scope is limited to the sequencing statement declaring the variable and to any of the sequencing statement's substatements. Users can assign each variable a value through an assignment statement and reference it within expressions. In the example in Figure 2, the **`addr`** variable initially has the assigned value 0x100. Variable **`addr`** appears later in the condition of the **`Repeat`** sequencing statement that directs the generator to generate pairs of instructions for as long as **`addr`** is less than 0x200. The same variable serves to specify the Store instruction's target address. Variables can provide partial specification of values to particular properties of an instruction statement. For example, the **`reg`** variable can specify that the Store instruction's source register is the same as the Add instruction's source register. The generation engine randomly chooses the actual value for the **`reg`** variable.

Expressions involve all the standard arithmetical, Boolean, and bit vector operators over constants, variables, and several predefined functions useful for template construction (for example, a random number generator). Expressions generally refer to template variables, resource values (say, the value of a register or a memory segment), test generation information (such as the number of registers used in the test so far), and several system constants (for example, the number of processes or memory size).

Genesys-Pro supports pre- and post-assertions for attachment to any statement in the template. Preassertions mean that the following statement should be part of the test only if the assertion holds before the statement's generation. Post-assertions indicate that the previous statement should be part of the test only if the assertion holds after the statement's generation. Procedurally, Genesys-Pro undoes the corresponding statement's generation if the post-assertion fails. Post-assertions are unusual in programming languages, but they are useful in a test template language because it is often difficult to preassess a statement's condition, as instructions have complex, externally defined semantics.

Constraint statements

Constraint statements influence the quality of the generated tests by biasing the generator's random decisions toward interesting areas. Users can activate or deactivate constraints, or mark them as mandatory or as prioritized nonmandatory.

The list of available constraints originates from the database of design-specific testing-knowledge constraints in the model (discussed later) or from a general list of generic biasing constraints applicable to any processor and provided as part of the generic generation engine. The following are examples of generic biasing constraints:

- *Alignment bias* causes the generation of addresses aligned at, near, or across some specified address boundaries, such as a word, a cache line size, or a page.
- *Cache bias* causes the generation of memory access patterns that in turn cause specified cache events such as hits, misses, and line replacements. Specified initializations of cache lines placed in the generated test (called *cache warm-loading*) can also help to generate some of these events.
- *Translation bias* is responsible for triggering various address translation mechanisms. Interesting translation scenarios include translation table events, translation path protection events, translation path reuse, and sharing.
- *Resource dependency bias* controls interdependencies among resources used by instructions that are close to each other in the test stream. Users can ask for source-source, source-target, target-source, and target-target dependencies, or for independence of resources (that is, they can cause close instructions to use different registers).

Constraint statements can also adjust the meaning of constraints. In Figure 2, the template sets the alignment size to 4 bytes and the activation rate of the alignment generic biasing constraint to 50%. This parameter setting holds for all instructions in the test except Load, which changes the alignment size to 16 bytes and the activation rate to 100%. This bias causes the generation engine to select a 16-byte aligned address for the Load instruction, as shown in the test. In addition, the request for the resource dependency bias causes the generation engine to select R5 as the target register of the Sub instruction in address 0x508.

The features just described significantly advance the

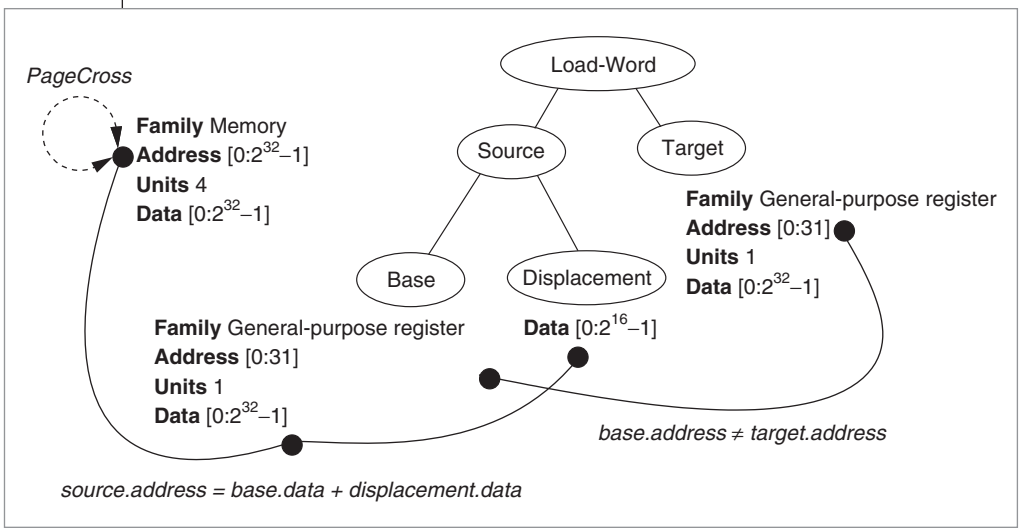

Figure 3. Model of a Load-Word instruction.

Genesys-Pro template language's expressiveness. In its predecessor, Genesys, the approach was based on controlling the test structure through a dedicated graphical user interface that supported limited test patterns, without the flexibility of a full programming language. A detailed description of the industrial experience with the Genesys and Genesys-Pro languages is available in the literature.[9]

Modeling framework

The model-based approach permits rapid development and easy maintenance of a test generation environment. Separating the model from the engine gives users the flexibility to maintain tool versions for several different designs and follow-ons. This separation also supports ongoing design changes—without depending on tool developers.

Genesys-Pro offers the following major advancements in the modeling framework:

- The modeling framework provides high-level building blocks designed specifically for modeling processors. This makes the model easier to write and to understand.
- The constraint-based representation of the modeled components gives the flexibility required to model many different and complex architectures.
- Genesys-Pro emphasizes the delineation between architecture-generic information (contained in the engine) and architecture-specific information (included in the model). The earlier Genesys system kept parts of the architecture-specific information in the engine because of that information's complex procedural nature. The more advanced Genesys-Pro can now model this information and keep it separate from the Genesys-Pro engine, providing the benefits of a model-based approach. This capability is especially useful in modeling VLIW architectures and complex address translation mechanisms.[10]

The model's architecture-dependent knowledge includes the following:

- a declarative description of the processor, including instructions, design resources (such as registers and caches), and high-level mechanisms (such as address translation); and
- design-specific testing knowledge to increase the test's coverage of verification events that randomly generated tests are likely to miss.

Instructions, described as constraint problems (attributes and relations), form the bulk of the Genesys-Pro model. Instruction attributes include an opcode and attributes that relate to the resources the instructions use, such as data, the resource family (floating-point register or memory), address (say, register number 5 or memory address 100), and number of units (for example, four bytes for a Load-Word memory operand).

Figure 3 shows a model of a Load-Word instruction that loads four bytes from memory. The operands form a tree structure, with the attribute names (in bold) and value domains. The Load instruction model has two operands: the source memory and the target register. The source memory operand, in turn, has two suboperands: the base register and the displacement (immediate field). The arcs in the figure denote relations between attributes and correspond to constraints. The instruction's architectural definition imposes these relations. For example, the source memory address is the sum of the value of the base register and the displacement (*source.address* = *base.data* + *displacement.data* in Figure 3). Relations such as this, which we

can express as equations with arithmetic and Boolean operators, are stated directly in the model. The modeling of relations with more-complex semantics can refer to an external implementation in C++, which the modeling engineer would have to provide.

We can also model instruction-specific testing knowledge as part of the instruction model. As an example, consider the instruction Add Rt ← Ra + Rb. The probability of randomly generating an instance of the Add instruction in which *data*(Ra) + *data*(Rb) = 0 is fairly low. Verification engineers might decide to give this combination a higher probability of appearing in tests. In this case, they can model a corresponding testing-knowledge relation between attributes *data*(Ra) and *data*(Rb). Figure 3 includes a testing-knowledge constraint (PageCross, represented by the dashed arc) that constrains the **`address`** attribute to cross a page boundary. Experience with Genesys-Pro shows that users often accumulate this type of testing knowledge during design verification and then pass it on to the models of follow-on designs. In addition, all designs that comply with the IEEE floating-point standard can use a testing-knowledge database based on FPgen.[11] This testing knowledge focuses on verifying the data path of floating-point designs. Genesys-Pro uses it for generating a rich variety of floating-point events.

Test generation engine

The Genesys-Pro generation engine takes a test template and a declarative model of the architecture as input and produces a test program that complies with the scenario requested by the template. Because the template typically represents a large set of verification events, there are several different ways to satisfy it. The generator's pseudorandom nature, coupled with its generic biasing constraints, ensures that each engine invocation exercises the requested scenario in different contexts by varying all parameters that aren't critical to the task itself. Consider the Load instruction in the table-walk example. Because the input template doesn't restrict the source memory address, different engine invocations might satisfy the Load request in different and interesting ways. For example, the selected address might cause a cache event or a translation event.

Test program generation occurs on two levels: stream and single instruction. Stream-level generation determines which instructions appear in the test and in what order, while instruction-level generation creates specific instruction instances. The two levels use different techniques to deliver their results: Stream-level generation is a recursive process; instruction-level generation uses constraint satisfaction techniques. The hybrid test-generation approach implemented by Genesys-Pro enables the generation of long, high-quality tests (tens of thousands of instructions). This increases the chance of hitting events such as filling large buffers and exploring machine states reachable only by nontrivial execution paths.[12] An alternative approach would be to solve the entire test problem as a single CSP. This way, users could express tighter dependencies at the stream level,[13] but, because the CSP is NP-hard, the approach severely limits the size of generated tests.

Stream generation

Sequencing-control statements in the user's test template are the primary drivers of stream generation. For each control statement, Genesys-Pro associates a separate stream solver, which recursively invokes solvers for its substatements. Each recursive path terminates with an invocation of the single-instruction solver. For an illustration, consider the **`select`** statement in Figure 2. The stream solver associated with this statement randomly chooses one of its substatements—say, the Add instruction—and invokes its solver.

In the test template, the user can express these dependencies between different instructions in the stream through shared variables (for example, the variable **`reg`** in Figure 2) and generic biases (such as the resource dependency bias in Figure 2).

When generating a particular statement, the engine considers only previously generated instructions but does not consider requests that follow the statement in the template. This can increase the possibility of failing to generate statements later on. We adopted two strategies to reduce this problem. The first involves detecting failures before they occur and injecting special instructions into the test to prevent them.[14,15] For example, when all the resources from a bounded resource family (such as floating-point registers) are initialized and can no longer be set to a requested value, Genesys-Pro applies a register reloading technique[15] to inject load instructions into the test stream, thereby setting registers with the desired values. In addition, each stream solver has rollback capabilities that let it retry substatement generation in case of failure. For example, in Figure 2, if the Add instruction fails to generate, the **`select`** solver will roll back to the state before the failed generation attempt and select another substatement (say, the Sub instruction) to generate.

The stream solver also allows controlled generation of

reentrant instructions that will execute more than once in a single test run. Examples are procedure calls, recurring interrupts, user-defined loops, and self-modifying code. Prohibiting randomly generated loops protects against generating infinite loops. When generating tests for multiprocessor or multithreaded designs, the stream solver creates separate instruction sequences for each processor or thread. This can involve additional synchronization mechanisms, described in the literature.[16]

Instruction generation

We decided to handle the problem of generating a single instruction by using the well-researched class of CSPs.[8] A CSP consists of a finite set of variables and a set of constraints between these variables. 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 CSP solution assigns each variable a value from its domain, such that these values satisfy all constraints. A constraint graph represents each CSP; its nodes are the CSP's variables, and its arcs are the CSP's constraints.

Generating a single instruction is a three-phase process. First, the generation engine formulates the instruction request as a CSP. Next, the instruction solver finds a solution to the CSP and produces an instance of the instruction that satisfies the constraints. Finally, the instruction solver invokes the architecture reference model for simulating the generated instruction.

To formulate the CSP, we first transform the instruction's architectural model into a CSP graph by associating a CSP variable with each instruction attribute (for example, *base.address* and *target.address* in Figure 3). We also associate a CSP arc with each architectural and testing-knowledge constraint; for example, we convert the relation *base.address* ≠ *target.address* in Figure 3 into an arc between these attributes. The constraints specified in the test template are then superimposed on the CSP graph. For example, the target address of the first Load in the test template reduces to the single element 5, according to the request for R5 in the template.

CSP solvers rely on search algorithms to find an assignment to each variable from its domain such that the assigned value satisfies all constraints. The test program generation domain typically has CSPs with large domains for their variables, resulting in very large search spaces. For example, the data attribute for a 32-bit register resource and the `address` attribute for a 32-bit machine both have a domain size of 2^{32}. A CSP solver's challenge is to produce uniformly distributed random solutions that satisfy mandatory and nonmandatory constraints when the problem's search space is very large. In general, CSP solution techniques that use strong filtering mechanisms to prune the search space are well suited to this type of problem.

Maintaining arc consistency (MAC)[8] is a family of algorithms that uses arc consistency as its filtering mechanism. We call an arc consistent if for any value in the domain of any variable in the arc there exists a valid assignment to the other variables in the arc that satisfies the constraint. The procedure for reaching arc consistency removes from the domains of variables those values that cannot participate in any solution.

Genesys-Pro uses an external, generic solver that implements a customized MAC algorithm for the pseudorandom test-program generation domain. This algorithm produces uniformly distributed random solutions that satisfy all the mandatory constraints and as many of the nonmandatory constraints as possible.[8] The solution proceeds as follows:

1. The algorithm first applies the arc consistency procedure over all mandatory constraints. If a constraint fails to reach arc consistency, the solution process fails; otherwise the process repeatedly reduces the domains of each of the arc's variables.
2. The algorithm then applies arc consistency over all nonmandatory constraints. It invokes constraints in priority order and ignores failures. Then it reconsiders the mandatory constraints, and the whole process repeats until it reaches a fixed point—that is, the process does not further reduce any variable's domain.
3. The algorithm then randomly selects one variable and reduces its domain to a single element. This retriggers the arc consistency procedure of steps 1 and 2.
4. After reducing the domain of each variable to a single element, the algorithm reaches a solution.

Table 1 shows one possible application of the algorithm for the Load instruction. Beginning with the second row, each line in the row shows one constraint and the domain reduction it causes. The constraints appear in the order of their application by the arc consistency procedure. From the table we see that after the first fixed point (fixed point 0), the solver eliminates the value 5 from the *base.address* variable's domain by applying the constraint *base.address* ≠ *target.address*.

Similarly, the source operand's address has been reduced to quad-word aligned addresses. This appears as the hexadecimal mask 0xXXXXXXX0, where each X represents a 4-bit don't-care.

MAC-based algorithms are well suited for the test program generation domain because they postpone heuristic decisions until after consideration of all architectural and testing-knowledge constraints. Thus, these algorithms can find solutions that satisfy all the architectural constraints and as many of the non-mandatory constraints as possible. Many other CSP algorithms[8] rely on local heuristic methods for early pruning of the search space. This weakens their ability to solve several constraints simultaneously and to allow a uniform exploration of the search space. To demonstrate this, we compared Genesys-Pro's solving capabilities and its MAC-based solution with the capabilities of its predecessor, Genesys, which uses a local heuristic search method. In both cases, the test template requests load instructions with 16-byte (quad-word) alignment and a read-after-write register dependency. We computed the percentage of instructions that satisfy both of these constraints as a function of the test length.

Table 1. Load instruction generation.

Fixed point	Constraint/selection	Domain reduction
Initial states	NA	base.address = 0..31 source.address = 0xXXXXXXXX target.address = 5 ...
Fixed point 0	*base.address ≠ target.address* *Alignment(16)* *source.address = base.data + displacement.data*	base.address = 0..4, 6..31 source.address = 0xXXXXXXX0 None
Fixed point 1	Random selection *source.address = base.data + displacement.data*	base.data = 0x0 displacement.data = 0xXXX0 source.address = 0x0000XXX0
Fixed point 2	Random selection	base.address = 4
Fixed point 3	Random selection *source.address = base.data + displacement.data*	displacement.data = 0x0FF0 source.address = 0x00000FF0
Fixed point 4	Random selection	source.data = 0x0F
Fixed point 5	Random selection	target.data = 0xFFFFFFFF

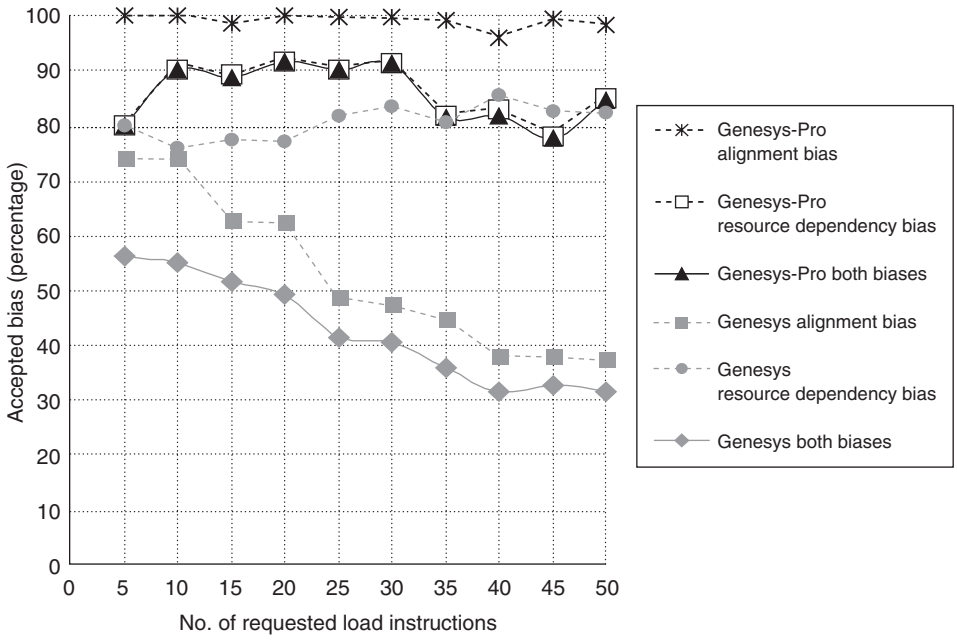

Figure 4. Ability of Genesys and Genesys-Pro to find acceptable solutions to constraint problems with biases for alignment and resource dependency. The graph also shows the rate of simultaneous satisfaction for both biases.

Figure 4 shows that Genesys-Pro's generation engine maintained a high level of success regardless of test length (the *x*-axis). Genesys' success level is significantly lower and decreases with test length. Furthermore, in Genesys-Pro, the rate for simultaneously satisfying two types of bias—alignment and resource dependency—is the same as that of satisfying the resource dependen-

cy bias alone. In Genesys, the simultaneous satisfaction rate is lower than the rate when satisfying either of the two biases separately.

GENESYS-PRO IS CURRENTLY the main test generation tool for functional verification of IBM processors, including several complex processors. Although it requires a high level of expertise to model architectures and testing knowledge to use the full power of test templates, Genesys-Pro's benefits are already apparent. Generated tests are higher in quality, different types of knowledge are much easier to maintain, full coverage of complex verification plans is possible, and there are very few or no escape bugs. We've found that the new language considerably reduces the effort needed to define and maintain knowledge specific to an implementation and verification plan. ■

■ References

1. E.M. Clarke, O. Grumberg, and D.A. Peleg, *Model Checking*, MIT Press, 1999.
2. B. Bentley, "Validating the Intel Pentium 4 Microprocessor," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, 2001, pp. 244-248.
3. S. Taylor et al., "Functional Verification of a Multiple-Issue, Out-of-Order, Superscalar Alpha Processor—the DEC Alpha 21264 Microprocessor," *Proc. 35th Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 638-643.
4. F. Haque, J. Michelson, and K. Khan, *The Art of Verification with Vera*, Verification Central, 2001.
5. L. Fournier, Y. Arbetman, and M. Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator: Application to the x86 Microprocessors Family," *Proc. Design Automation and Test in Europe* (DATE 99), IEEE CS Press, 1999, pp. 434-441.
6. A. Aharon et al., "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator," *IBM System J.*, vol. 30, no. 4, Apr. 1991, pp. 527-538.
7. A. Aharon, Y. Lichtenstein, and Y. Malka, "Model-Based Test Generator for Processor Design Verification," *Proc. 7th Innovative Applications of Artificial Intelligence Conf.* (IAAI 94), AAAI Press, 1994, pp. 83-94.
8. E. Bin et al., "Using Constraint Satisfaction Formulations and Solution Techniques for Random Test Program Generation," *IBM Systems J.*, vol. 41, no. 3, Aug. 2002, pp. 386-402.
9. M. Behm et al., "Industrial Experience with Test Generation Languages for Processor Verification," submitted to DAC 04, 2004.
10. A. Adir et al., "Deeptrans—A Model-Based Approach to Functional Verification of Address Translation Mechanisms," *Proc. 4th Int'l Workshop Microprocessor Test and Verification: Common Challenges and Solutions*, IEEE CS Press, 2003, pp. 3-7.
11. M. Aharoni et al., "FPgen—A Test Generation Framework for Datapath Floating-Point Verification," *Proc. 8th Ann. IEEE Int'l Workshop High-Level Design Validation and Test* (HLDVT 03), IEEE Press, 2003, pp. 17-22.
12. A. Hartman, S. Ur, and A. Ziv, "Short vs. Long—Size Does Make a Difference," *Proc. High-Level Design Validation and Test Workshop* (HLDVT 99), IEEE Press, 1999, pp. 23-28.
13. A. Adir et al., "Piparazzi: A Test Program Generator for Micro-Architecture Flow Verification," *Proc. 8th Ann. IEEE Int'l Workshop High-Level Design Validation and Test* (HLDVT 03), IEEE Press, 2003, pp. 23-28.
14. A. Adir, R. Emek, and E. Marcus, "Adaptive Test Program Generation: Planning for the Unplanned," *Proc. 7th Ann. IEEE Int'l Workshop High-Level Design Validation and Test* (HLDVT 02), IEEE Press, 2002, pp. 83-87.
15. A. Adir et al., "Improving Test Quality through Resource Reallocation," *Proc. 6th IEEE Int'l Workshop High-Level Design Validation and Test* (HLDVT 01), IEEE CS Press, 2001, pp. 64-69.
16. A. Adir and G. Shurek, "Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification," *Proc. 7th Ann. IEEE Int'l Workshop High-Level Design Validation and Test* (HLDVT 02), IEEE Press, 2002, pp. 77-82.

Allon Adir is a research staff member at the IBM Research Laboratory in Haifa. His research interests include test program generation, multiprocessor verification, languages for shared memory, and distributed programming. Adir has a BS and an MS in computer science from the Technion, Israel Institute of Technology.

Eli Almog is a research staff member at the IBM Research Laboratory in Haifa. His research interests include test program generation, processor verification, and scheduling algorithms. Almog has a BS in mechanical engineering and an MS in computer science from the Technion, Israel Institute of Technology.

Laurent Fournier is a research staff member at the IBM Research Laboratory in Haifa. His research interests include test program generation, verification methodology, and functional coverage. Fournier has a BS and an MS in computer science from the Technion, Israel Institute of Technology.

Eitan Marcus is a research staff member in the Verification Technologies Department at the IBM Research Laboratory in Haifa. His research interests include test program generation, functional coverage, and constraint-based modeling languages. Marcus has a BS from Columbia University and an MS from Carnegie Mellon University, both in computer science.

Michal Rimon is a research staff member at the IBM Research Laboratory in Haifa. Her research interests include knowledge-based systems, test program generation, planning, and constraint satisfaction. Rimon 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.

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 the Moscow Institute of Radio Technique, Electronics and Automation; and an MS in computer engineering from the Technion, Israel Institute of Technology.

Avi Ziv is a research staff member in the Verification Technologies Department at the IBM Research Laboratory in Haifa, where he works on simulation-based verification. His research interests include functional coverage, coverage-directed test generation, and high-level modeling for hardware systems. Ziv has a BS in computer engineering from the Technion, Israel Institute of Technology, and an MS and a PhD in electrical engineering from Stanford University.

■ Direct questions and comments about this article to Michael Vinov, IBM Research Laboratory, Haifa University Campus, Haifa 31905, Israel; vinov@il.ibm.com.