Industrial Experience with Test Generation Languages for Processor Verification

Michael Behm, John Ludden
{behm, ludden@us.ibm.com}
IBM Development Center
Austin, Texas

Yossi Lichtenstein, Michal Rimon, Michael Vinov
{yossil, michalr, vinov@il.ibm.com}
IBM Research Laboratory
Haifa, Israel

ABSTRACT

We report on our experience with a new test generation language for processor verification. The verification of two superscalar multiprocessors is described and we show the ease of expressing complex verification tasks. The cost and benefit are demonstrated: training takes up to six months; the simulation time required for a desired level of coverage has decreased by a factor of twenty; the number of escape bugs has been reduced.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Verification, Measurement, Experimentation

Keywords

Functional verification, Processor verification, Test generation

1. INTRODUCTION

Functional verification is widely recognized as the bottleneck of the hardware design cycle. The ever-growing demand for performance and time-to-market, coupled with the exponential increase in hardware size, cause the verification task to become increasingly difficult. This problem is exacerbated by low tolerance for bugs on the finished product. While noticeable progress has been made through the use of formal methods, such as model checking [5] and theorem proving [11], these approaches are still limited to the verification of relatively small design blocks, or to very focused verification goals. Simulation-based techniques still play a major role in functional verification.

In recent years special purpose verification languages have been developed to support automatic stimulus generation. Synposys' Vera [7], Verisity's *e* [10] and Cadence's System-C Verification library [8] are the prime examples. These languages allow engineers to express pseudo random generation in conjunction with intricate event scenarios. This should enable better coverage of the specification and design, especially in corner cases. The expected result is to uncover bugs early and to achieve high quality of the design for the silicon tape-out. However, these powerful languages are conceptually complex and entail a significant learning period and high level of expertise in order to benefit from their potential.

In this paper we present the costs and benefits of a similar verification language in an industrial setting. Our context is relatively narrow — the verification of high-end processors using automatic generation tools. The language is relatively domain specific and the underlying tool may be more powerful in the specific domain. However, the language includes many of the programming constructs supported by the general purpose verification languages, and we suspect that our experience is highly relevant to the general languages. In particular, we have found that learning to fully exploit the language takes several months for an experienced verification engineer. We also discovered that the benefit is considerable — a compact test suite and a twenty fold speed up in simulation time, better coverage and fewer bugs that escape to hardware. We have further learned that the powerful language changed the methodology of the verification team. Previously many verification engineers wrote test templates, but with the new language this task is done by only a few experts; the test templates are now shared between different design projects.

The new language is a result of more than twenty years of test generation for processor verification within IBM. The first tool was developed in the mid eighties and described in the IBM System Journal in 1991 [2]; it allowed limited user control. The second generation of these tools was described in a 1995 DAC paper [3] and it is used in this work as a baseline. The tool, named Genesys, allows users to select processor instructions for random generation and global policies to bias the randomness. The language is represented in the GUI of the tool as shown in Figure 1. The language allows the user to express regular expressions over the instruction set. The third generation of this line of tools is called Genesys-Pro [1] and includes a special purpose programming language that allows the user to express both wide biasing policies and intricate scenarios of events. The tool's GUI includes a syntax directed editor as presented in Figure 2. The new language is as expressive as a probabilistic Turing machine [12].

The two languages are compared by contrasting our experience in verifying the IBM POWER4 processor [9] in the late 1990s with Genesys with the recent verification of the POWER5 design (due in IBM products in 2004) with Genesys-Pro. The processors are comparable superscalar multiprocessor RISC machines with multi-threading only in the POWER5 design. The comparison is done by implementing several verification tasks by both Genesys and Genesys-Pro. The tasks include verifying the coherency of an L2-cache, checking the branch instruction queue, and verifying the functionality of an instruction-cache when multiple copies

DAC 2004, June 7–11, 2004, San Diego, California, USA.

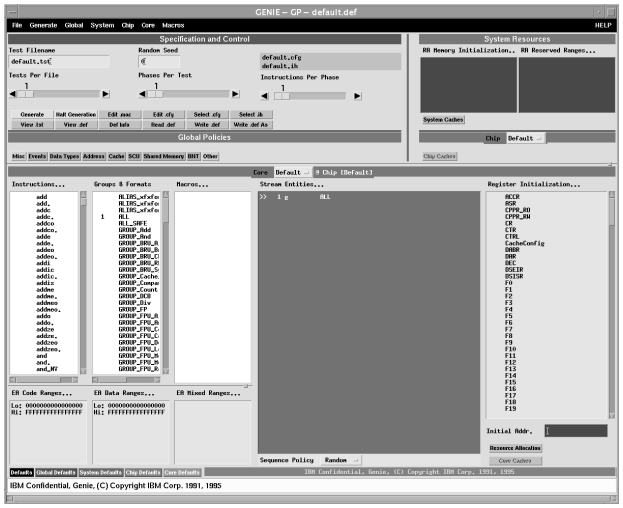

Figure 1: Genesys GUI

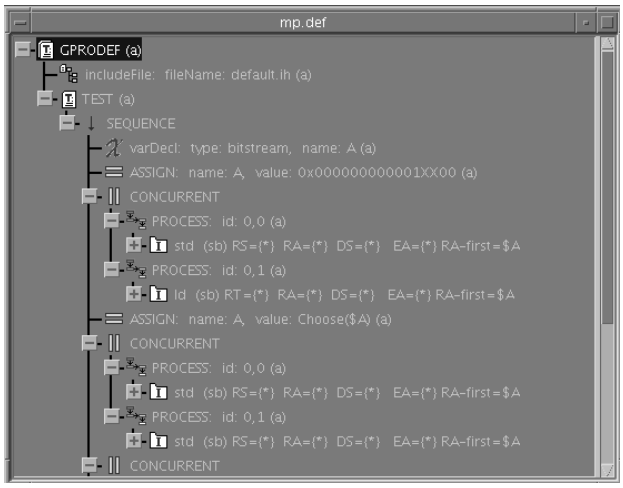

Figure 2: Genesys-Pro syntax-directed editor

of a cache line reside in the cache. The control capabilities of the Genesys-Pro language allowed verification engineers to generate tests that provided excellent coverage of the verification tasks; some of these verification tasks were impossible to express with the Genesys language. The end result has been higher quality first pass hardware for POWER5.

The paper continues with a description of the two languages in Section 2 and the details of our experience in Section 3; a brief discussion section follows.

2. THE TEST GENERATION LANGUAGE

Genesys is a random test generator for processor verification [3]. Users can control the generated tests via a Graphical User Interface that allows the definition of instruction lists and generation constraints. During a test generation, the instructions are selected from the list according to a sequence policy, which may be

Random Instructions are selected randomly from the list. The user is allowed to set relative weights on instructions and is required to set a global `Instructions per test` directive.

Cyclic Instructions are generated in the order they appear in the instruction list while their number is defined by the global `Instructions per test` directive.

Ordered The instruction list defines both instruction order and test length.

For each instruction the user is further allowed to set the address, data, length and other properties of the instruction operands explicitly or via a bias function or to leave their value for random generation. Finally, the language also supports symbols which enables users to state dependencies between operands and instructions.

In Genesys-Pro the user control has evolved into a special purpose programming language; it provides virtually unlimited control over the events to be generated, and at the same time it allows the user to leave non-critical parameters unspecified. The language consists of four types of statements: `basic instruction statements`, `sequencing control statements`, `standard programming constructs`, and `constraint statements`.

`Instruction statements` specify the required properties of an instruction to be placed in the generated test. Users can specify properties of instructions in several ways. They can leave properties completely unspecified, provide them with specific value, or specify properties via user-defined variables and other expressions.

`Sequencing control` consists of the following four statements

Sequence directs Genesys-Pro to generate a list of sub-statements in the order they appear in the sequence.

Permute directs Genesys-Pro to generate a list of sub-statements in some randomly selected order.

Select directs Genesys-Pro to generate a single substatement which is selected randomly from the list of substatements according to the weight attribute.

Repeat causes its child statements to be repeated a given number of times or as long as a boolean expression, defined as the repeat condition, evaluates to True.

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

`Programming constructs` of the Genesys-Pro language include variables, assignment statements, expressions, and a conditional generation control mechanism. These constructs combine with other sub-statements to define the flow of the test template or to give imperative directions to the test program generator.

Genesys-Pro supports typed, scoped variables, including array variables. The scope of a variable is limited to the sequencing statement in which it is declared, or to any of its sub-statements. A variable can be used as the target of an assignment statement or in any place that an expression is expected. Variables are evaluated to sets of objects. These include sets of integers, bit vectors, strings and booleans. These data types are defined in more details in [4]. Variables can be used to provide partial specification of values for specific properties of an instruction statement.

Expressions involve all the standard arithmetical, Boolean, and bit vector operators over constants, variables and several predefined functions useful for template construction (e.g., a random number generator). There are many places where expressions can be used in the language. Expressions generally refer to template variables,

Test Program Template	Test Program

Variable: addr = 0x100
Variable: reg
Bias: register-dependency
 	
Instruction: Store R5 → ?
Repeat (addr < 0x200)
  Instruction: Load reg ← addr
  Select
  Instruction: Add ? ← reg + ?
  Bias: sum-zero
  Instruction: Sub ? ← ? - ?
  addr = addr + 0x10
	Resource Initial Values:
	  `R6=8, R3=-25,..., R17=-16`
	  `100=7, 110=25,..., 1F0=16`
	Instructions:
	  `500: Store R5 → FF0`
	  `504: Load R4 ← 100`
	  `508: Sub R5 ← R6-R4`
	  `50C: Load R4 ← 110`
	  `510: Add R6 ← R4+R3`
	  :
	  `57C: Load R4 ← 1F0`
	  `580: Add R9 ← R4+R17`

Figure 3: Test template and corresponding test

resource values (e.g., the value of a register or a memory segment), test generation information (e.g. the number of registers used in the test so far), and several system constants (e.g., the number of processes, memory size).

The conditional generation control mechanism includes pre- and post-condition expressions that may be attached to any statement in the template. Pre-conditions imply that the corresponding statement should be solved only if the condition holds before the statement is generated, while post-conditions indicate that the corresponding statement should be undone if the condition fails after the statement is generated.

`Constraint statements` influence the validity and quality of generated tests. Validity constraints are always mandatory, in the sense that they *must* hold in all the tests generated from this template. Quality constraints may be mandatory, but it is more common to request them as prioritized non-mandatory constraints.

All these language constructs enable users to control the degree of randomness in the generated tests, ranging from completely random to completely directed. In most cases the template will be a balance of both.

Figure 3 shows an example of a test template expressed in the Genesys-Pro language that defines a table-walk scenario. The scenario loads registers from contiguous addresses in memory. Each `Load` instruction is followed by either an `Add` or a `Sub` instruction. The right part of the figure shows an example of a test generated from this template. The scenario is solved successfully by the generator. The memory locations accessed by the `Load` instructions are indeed contiguous as shown in the Resource Initial Values section of the test (addresses 0x100–0x1F0).

The template includes four basic instruction statements for `Store`, `Load`, `Add`, and `Sub`. Each statement specifies required properties of an instruction to be placed in the generated test. Users can specify properties of instructions in several ways. They can leave properties completely unspecified ('?' in the figure), provide them with specific value (`R5` in the `Store` instruction), or specify properties using user-defined variables (`reg` in `Load` and `Add`).

Variables are used in the example to provide partial specification of values. In particular, the `reg` variable is used to specify that the target register of the `Load` instruction is equal to the source register of the `Add` instruction. Note that variables used in the specification have no corresponding resource in the resulting test. A similar example is the conditional `Repeat` statement — it directs the generator to keep generating the pairs of `Load` and `Add` or `Sub` instructions, as long as the address variable `addr` is less than 0x200. Therefore, there should be 16 pairs of `Load` and `Add` or `Sub` instructions in the test. Note again that there is no actual loop in the corresponding test.

Figure 3 also includes an example of a constraint statement. In particular, the register-dependency bias, which causes the generation engine to select the same registers for adjacent or near adjacent instructions in the test. This bias causes the generation engine to select `R5` as the target register of the `Sub` instruction located in address 0x508.

The language also includes support for Coverage Driven Generation that influences the generation of multiple tests. Users can define a coverage model [6] (a family of related coverage tasks) to direct the generation process. Each choice point in the test template can be marked as a coverage point. The coverage model is defined as an expression over coverage points. For example, the `Select` statement in Figure 3 can be marked as a coverage point and form a coverage model. In this case, Genesys-Pro will generate enough tests until all of the `Select`'s sub-statements (`Add` and `Sub`) are generated.

3. INDUSTRIAL EXPERIENCE WITH THE LANGUAGES

In this section we contrast our experiences in verifying two processor designs using the Genesys and Genesys-Pro verification languages. The examples are the POWER4 and POWER5 processors; both are comparable superscalar, multiprocessor RISC machine. The POWER5 design is somewhat more complex because of its Simultaneous Multithreading (SMT) capability. We present three verification tasks:

- Verifying the coherency of an L2-Cache in a multiprocessing configuration. This is possible with Genesys, however, only 50% of the generated tests satisfy the required microarchitectural event. In contrast, Genesys-Pro is fully efficient and all the generated tests satisfy the desired event.

- Verifying the functionality of a branch instruction queue (BIQ). Tests generated by Genesys cover some of the BIQ functionality, but do not reach boundary cases. Genesys-Pro achieves full coverage of the boundary cases.

- Verifying the functionality of an instruction cache when multiple copies of the same line reside in different cache locations. Genesys is not able to produce the required scenario, while Genesys Pro produces this scenario easily.

Our first example is the verification of the behavior of the L2-Cache in a multiprocessing configuration. The requested event involves two processors. The first processor is required to access 5 to 15 cache rows with 50 load instructions and the second processor is required to generate 50 store instructions that access between 10 and 20 cache lines used by the first processor. In Genesys, we used the global cache biases provided by the tool to express this scenario. In Genesys-Pro we were able to express the desired event more accurately, using shared variables to restrict the addresses accessed by the second processor to the cache lines accessed by the first processor. Figure 4 shows the coverage results. We can see that all 50 tests generated by Genesys-Pro satisfy the desired event in an evenly distributed manner, while the tests generated by Genesys were roughly in the area of the event to be covered, but did not fully hit the event.

The second example is the verification of the Branch-Instruction Queue (BIQ). In both designs, the size of the BIQ determines how many unresolved branches can be in flight at any point in time. The queue size is cut in half when running in simultaneous multithreading mode since the BIQ is shared equally between the threads. A

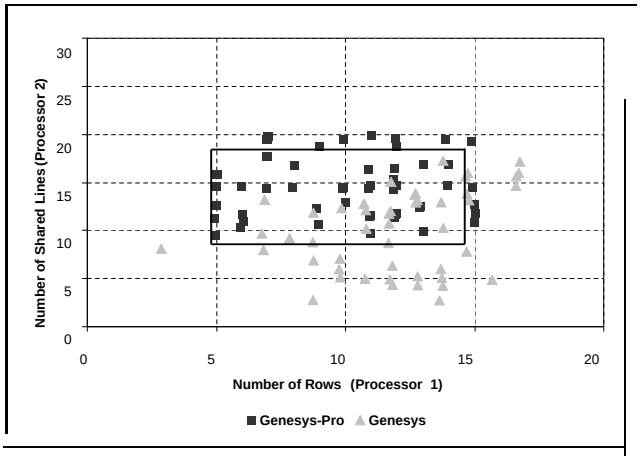

Figure 4: Bias vs. Language Expressiveness

test for verifying BIQ behavior should specify a sequence consisting of a long latency instruction (e.g. a load which misses the data cache) followed by a sequence of branches with particular known properties (e.g. correctly predicted or mis-predicted). Genesys does not provide the means for ensuring certain combinations of predicted and mis-predicted branches. In addition, Genesys requires the BIQ size to be a constant, entailing separate test templates for the single-threaded and multi-threaded configuration. Genesys-Pro, on the other hand, allows to define the BIQ-size as a variable, and to use special directives to enforce the application of the required bias. The following code example illustrates the ease with which such tests can be generated in Genesys-Pro. The expression BIQ_size/NumOfThreads() determines the length of the BIQ per thread dynamically, thus reducing the number of test templates that needs to be maintained. In addition, the predicted_taken-actually_taken bias statement is set to 100%, which causes the bias to become a hard constraint with respect to this particular test template, and ensures that every test generated from this template will satisfy the bias.

```
Variable: BIQ_size = 16;
Repeat times=BIQ_size/NumOfThreads() {
    Permute {
        Instruction-Group: LoadStore
        Instruction-Group: Branch
                           Bias: predicted_taken_actually_taken=100
    }
}
```

In Genesys, very specific test templates were written to verify this area of the design. Consequently, some bugs were not found until the initial POWER4 hardware was available. For POWER5, Genesys-Pro test templates uncovered 3 design bugs prior to first pass silicon. As a result, no bugs escaped the verification phase to hardware despite the added complexity introduced by Simultaneous Multithreading (SMT).

The third example involves testing the functionality of an Instruction-Cache (I-Cache) when multiple copies of the same cache line reside in different locations in the cache. To exercise this, we try to generate tests that repeatedly access the same set of lines in the I-Cache with different Effective to Real address mappings. We do this by specifying a sequence of branch instructions for each selected line in the I-Cache, such that the target address of each branch is restricted to the real address range in the respective cache line. To keep the tests as general as possible the branch target address is selected randomly from this range. Thus, the input test template must specify additional restrictions on the range of the target address, to avoid infinite loops in the generated tests. Since the Genesys language provides no means for expressing these additional restrictions, the scenario could not be described in its general form. Only very directed scenarios could be written, but those did not cover the desired event.

Genesys-Pro, in contrast, provides enough programming control to allow easy implementation of the general scenario, as illustrated by the following code example. We use the array variable `isUsed` to ensure that there is at most one branch instruction to each address location in the selected line. The Effective to Real address mapping is affected by setting the Address-Aliasing bias statement to 50%, which directs the generator to select different effective addresses that map to the same real address location.

```
Variable: i,j=0;
Variable: isUsed[]=0;
Variable: MaxInstructionsPerLine=32;
Variable: MaxNumOfCacheLines=8;
Bias: Address-Aliasing = 50;
Repeat (i < Random(1, MaxNumOfCacheLines)) {
    CacheLineStart[i] = Random(0bXXX...XXX0000000);
    Repeat (j < MaxInstructionsPerLine) {
        k = Random(0, MaxInstructionsPerLine-1);
        Instruction: Branch
                     precondition: (isUsed[i,k] == 0);
                     targetRealAddr = (CacheLineStart[i] + k);
        isUsed[i,k] = 1;
        j++;
    }
    i++
}
```

A summary of the three verification tasks appears in Figure 5.

The expressiveness of the new language is also demonstrated by the size of the verification suite. For Genesys, we used 35,000 test templates while the Genesys-Pro suite includes only 2,000 test templates. The nature of these templates has changed as well. For Genesys we used two types of templates — the majority covered syntactical architecture events (e.g. all pairs of instructions) and were produced from simple scripts that enumerated them. A relatively small number of test templates were written directly in the Genesys language. In total, we have about 33,000 script generated test templates and 1,900 manually written templates. The 2,000 test templates we have for Genesys-Pro have all been written directly in the new language which allows knowledgeable verification engineers to target microarchitectural events. The coverage of both suites is the same, however the 35,000 test templates takes 90 days to simulate, while the 2,000 test templates require only 4 days of simulation with the same hardware resources. This 20 fold speed up is critical in increasing verification productivity because it allows for more rapid breakage testing of the design when late changes are introduced shortly prior to tape out, thereby reducing time to market.

4. DISCUSSION

The introduction of a powerful verification language into IBM's test generator for processor verification had two significant consequences. The first is several changes in the verification methodology. The large number of test templates have been replaced by about two thousand templates which are shared between projects. Also, the team of about ten verification engineers who used to define templates has been replaced by two to four experts who are

	Genesys	Genesys-Pro	Technique
L2-Cache	inefficient, 50% coverage	full coverage, uniform distribution	Directed Random
BIQ	boundary cases not covered, bugs escaped to hardware	full coverage, no bugs escaped	Local constraints
I-Cache	impossible to specify, bugs escaped to hardware	full coverage, no bugs escaped	Utilization of programming constructs

Figure 5: Genesys/Genesys-Pro comparison table

now responsible for coding the new templates. The second consequence of the new language is the performance of the test generation tool. Tests have better quality, the test suite is compact, simulation time is twenty times lower and most importantly, fewer bugs have escaped into silicon despite the added complexity of simultaneous multithreading.

The reasons for these two important consequences is the language's ability to express in minute detail the required micro-architecture scenarios, leave unnecessary detail open to exercise an event in various contexts, and parameterize test templates for various verification tasks and projects.

The cost of better expressiveness and performance is twofold. First, building of a test generator that fully supports the new language. Considerable investment has been required mostly for developing a powerful constraint satisfaction solver. Secondly, the training period of verification engineers has increased; it takes an experienced engineer from two to six months to learn to utilize the full capabilities of the language. Still, novices can exploit the tool with minimal learning time to create basic scenarios.

Our conjecture is that our experience with the Genesys-Pro language is applicable to general verification languages like *e*, Vera and the System-C Verification library. The complexity of systems and designs and the time pressures on teams require full programming languages and powerful generation tools. The costs are considerable, but the benefits are more so.

The main contribution of this paper is in reporting real-world experience with a verification language. The projects and verification tasks we described are high-end real examples; we also supplied some analysis of the costs and benefits of the new language. This type of data is significant when both vendors and customers invest considerably in verification languages. An additional contribution is the description of the verification language.

The main limitation of this study is its narrow context. We describe a language for processor verification while most verification languages are more general. However, the underlying mechanisms are the same — modeling of events, constraints satisfaction, randomness — and we assume that our conclusions hold for a more general scope. Further investigation of actual usage of verification languages is called for.

5. REFERENCES

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-Pro: Innovations in test program generation for functional processor verification. Submitted to IEEE Design&Test of Computers, 2003.

[2] 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 System Journal*, 30(4), April 1991.

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

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

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT-Press, 1999.

[6] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[7] F. Haque, J. Michelson, and K. Khan. *The Art of Verification with Vera*. Verification Central, 2001.

[8] N. Ip and S. Swan. An introduction to the new SystemC verification standard. In *Proceedings of the 2003 Design, Automation and Test in Europe Conference (DATE)*, March 2003.

[9] J. Ludden et. al. Functional verification of the power4 microprocessor and power4 multiprocessor systems. *IBM Journal of Research and Development*, 46(1), 2002.

[10] Samir Palnitkar. *Design Verification with e*. Prentice Hall, 2003.

[11] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[12] M. Spiser. *Introduction to the theory of computation.* Brooks-Cole, 1997.