Simulation Knowledge Extraction and Reuse in Constrained Random Processor Verification

Wen Chen[1], Li-Chung Wang[1], Jay Bhadra[2], Magdy Abadir[2]
[1]University of California - Santa Barbara
[2]Freescale Semiconduction Inc.

ABSTRACT

This work proposes a methodology of knowledge extraction from constrained-random simulation data. Feature-based analysis is employed to extract rules describing the unique properties of novel assembly programs hitting special conditions. The knowledge learned can be reused to guide constrained-random test generation towards uncovered corners. The experiments are conducted based on the verification environment of a commercial processor design, in parallel with the on-going verification efforts. The experimental results show that by leveraging the knowledge extracted from constrained-random simulation, we can improve the test templates to activate the assertions that otherwise are difficult to activate by extensive simulation.

Categories and Subject Descriptors

B6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Verification, Data Mining

Keywords

Functional Verification, Assertion, Coverage, Rule Learning

1. INTRODUCTION

The mainstream practice for functional verification of microprocessors today still heavily relies on extensive simulation. Functional verification starts with a verification plan, specifying the aspects of the design to verify [9]. In constrained-random verification, test cases are generated by constrained-random test generation, in which the test generator instantiates tests using templates written by verification engineers. The tests are simulated with checkers to check the correctness of the design. The completeness of simulation-based verification is measured by various coverage metrics. During the verification process, coverage results are analyzed to guide test generation. A satisfactory level of coverage must be met before tapeout.

This work is supported by Semiconductor Research Corporation, project 2012-TJ-2268.

DAC '13 May 29 - June 07 2013, Austin, TX, USA

In a design cycle, the design evolves over time. Consequently, functional verification is an iterative process in which extensive simulation is run on a few relatively stable versions of the design. When a new version is released with accumulated changes over a period, the verification process restarts with the new version. From one iteration to another, two assets are kept. The first are the test templates refined and accumulated up to the previous iteration. The second are the important tests identified so far. For example, an important test can be the one activating a particular assertion of interest or capturing a bug in the previous design versions. These two assets embed the knowledge accumulated through the iterative verification process.

During the iterative process, it may not be effective to maintain the detailed structural coverage results from one iteration to the next due to major changes in the implementation. Therefore, functional coverage based on assertions (how many times an assertion is activated) is often used as the metric to evaluate the importance of tests and to guide test template refinement. Assertions are relatively stable and do not change as often as the design implementation.

In this work, we propose a novel learning methodology for extracting knowledge from important tests. The extracted knowledge then is reused for two purposes: (1) for producing more tests similar to those important ones and (2) for producing new important tests that, for example, can activate assertions not covered before. To develop such a learning methodology, we need to address three aspects: (1) what knowledge to extract, (2) how to extract and represent knowledge, and (3) how to reuse the extracted knowledge.

In this work, we applied the proposed methodology to verifying a dual-threaded low-power 64-bit Power Architecture-based processor core to be manufactured with a 28nm technology. Our experiments were conducted in parallel with the verification process where the design was not yet stable. The experimental results demonstrate the effectiveness of the methodology for the two intended purposes. More specifically, we show that after applying the extracted knowledge, a refined test template can effectively generate additional tests for activating an assertion that received low coverage before. Moreover, a refined test template can effectively generate tests for activating an assertion that was not covered before. Test template refinement using knowledge learned from simulation is naturally used in real-world verification processes, however, the knowledge is largely acquired by manual effort. In this work, automating the process of knowledge extraction is addressed by feature-based rule learning.

The rest of the paper is outlined as follows: Section 2

briefly reviews related works. Section 3 addresses the first aspect, i.e. what knowledge to extract. A feature-based rule learning methodology is presented in Section 4 to address the knowledge representation aspect. Section 5 discusses the knowledge extraction aspect using subgroup discovery rule learning. Section 6 illustrates how the knowledge can be reused. Experiment results are presented in Section 7. Section 8 concludes the paper.

2. RELATED WORKS

Coverage-directed test generation (CDTG) is an approach to dynamically analyze coverage results and automatically adapt the test generation process to improve coverage. Recent works proposed various techniques to learn from the simulation results. These approaches employ a variety of learning techniques such as Bayesian Networks [8], Markov Models [13], Genetic Algorithms [11] and Inductive Logic Programming [6]. However, automatically modifying the input to the test generator, based on the feedback from simulation, can be very difficult for complex designs. In a recent work [9] the authors proposed to learn test knowledge from micro-architectural behavior and embed the knowledge into test generator to produce more effective tests.

Early identification of the important tests to reduce simulation cost was proposed in [2][4]. Feature-based rule learning has been applied in the context of understanding design-silicon mismatch [1][3]. In contrast, this work studies the feasibility and effectiveness of applying feature-based analysis for extracting knowledge from important tests to improve functional (assertion) coverage.

3. WHAT KNOWLEDGE TO EXTRACT

This work considers knowledge extraction in the context of constrained-random verification for assertion coverage. Figure 1 illustrates a scenario of simulation with tests instantiated from a given test template that had been refined by the verification team up to the time of the experiment. The figure summarizes the statistics of covered assertions for the Load Store Unit (LSU) of the processor in a simulation of 3000 tests. The LSU is among the most complex and difficult-to-verify units in the design. Over 90% of the covered assertions were already activated by 50 or more tests. However, there existed other assertions activated only by 10 tests or fewer. Furthermore, there were assertions with zero coverage (not shown in the figure).

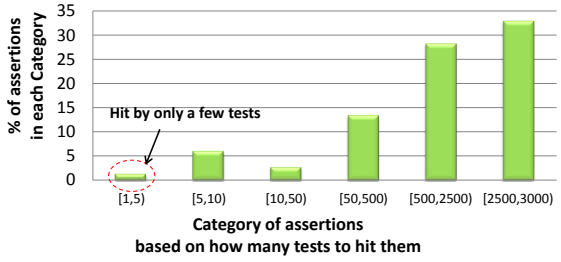

Figure 1: Histogram of covered assertions in LSU based on frequency of being activated

Our interest is in knowledge extraction for activating those assertions with low or zero coverage. The property stated by a complex assertion comprises multiple conditions. Learning the knowledge about the entire assertion directly could be difficult. Hence a divide-and-conquer strategy is employed. The idea is to learn knowledge with respect to each condition and then, the knowledge can be combined for activating the assertion.

Knowledge extraction for a given condition is based on tests activating the condition. We call those tests the *novel* tests. In processor verification, a test is an assembly program. Figure 2 illustrates the learning goal. Suppose a novel assembly program is identified to trigger a special condition in the simulation, for example, a "coreflush" condition concerning the instructions already fetched but not yet committed. Then what we want to learn are descriptive *rules* explaining the properties in the novel tests that trigger the condition, for example, the rule being the existence of a mispredicted branch in the test. Such rules are then used as constraints to refine test templates for hitting the condition.

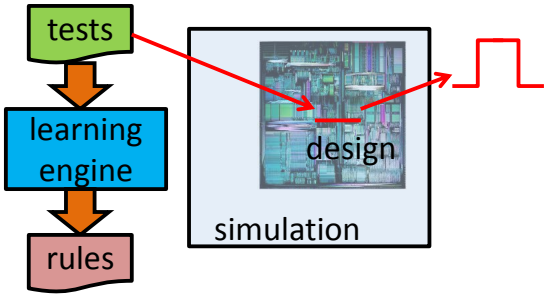

Figure 2: Illustration of the learning goal

To summarize, in our methodology we begin by extracting a list of conditions from the assertions for monitoring. Novel tests with respect to these conditions are identified in the simulation. The extracted knowledge is rules describing the special properties of the novel tests.

4. FEATURE GENERATION

4.1 Snippet-based Vector Representation

To extract knowledge from an assembly program, we first need an approach to automatically convert an assembly program into a representation suitable for applying the feature-based rule learning. A given assembly program may consist of hundreds of instructions. Our representation approach comprises two steps. The first step converts an assembly program into multiple *snippets* of instruction sequence of equal length k where k is a user-supplied input.

Figure 3 illustrates how this step, with a slide window size of 3, works on an example test with 6 instructions. Six snippets are extracted, where the ith snippet ends with the ith instruction in the test. Beginnings of the first two snippets are filled with dummy instructions.

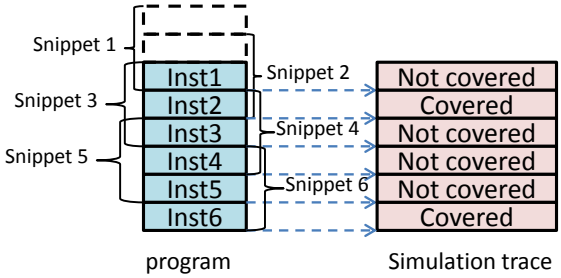

Figure 3: Illustration of the slide window approach

Each snippet is paired with the episode of simulation trace starting from the the commitment of the second-to-last in-

struction and ending at the commitment of the last instruction. In this way, each snippet is paired with a unique simulation episode. For a given condition to be monitored, the episode is used to decide if the condition is covered by the snippet. Figure 3 shows, in the example, the condition is covered by snippets 2 and 6.

The second step of the representation approach is to convert each snippet into a feature vector. A feature vector encodes a sequence of instructions based on a set of features. A feature can be an *occurrence* feature or a *descriptive* feature. An occurrence feature has a value of 0 or 1. A descriptive feature has a numerical value.

occurrence				*descriptive*				
o_1	o_2	...	o_n	d_1	d_2	...	d_m	$Class$
1	0	...	1	231	6	...	54	-1
1	1	...	0	78900	654	...	37	$+1$
0	1	...	0	256	800	...	24	-1
0	0	...	0	3	60	...	4096	-1
0	0	...	1	701	9754	...	7	-1
1	1	...	1	1	570	...	0	$+1$

Figure 4: Illustration of the transformed dataset

Figure 4 illustrates the look of a feature-encoded dataset based on the six snippets for a condition. The illustration shows n occurrence features and m descriptive features. The set of snippets are divided into two classes, the positive class with the condition being activated and the negative class without the condition being activated.

It is important to note that in the analysis, the negative class will also contain snippets obtained from non-novel tests. Hence, the size of the negative class is usually much larger than the size of the positive class. It is also important to note that such a feature-encoded dataset is constructed for each condition to be monitored.

4.2 Defining a Set of Features

Features are ISA dependent. A feature set defined for PowerPC ISA can be different from that for x86 architecture. Defining the feature set is treated as a one-time cost, although the feature set may be manually expanded during the verification process. The development of a good feature set is a process of incorporating domain knowledge into learning, thus it requires delicate design efforts. However, because the feature set only depends on the ISA, it can be reused for generations of design compatible to the ISA.

In this work, the feature set is defined based on the Power ISA. In the experiment presented later, we consider three categories of features:

- **State-based features:**
  - The contents of a set of special registers such as machine state register (MSR), exception syndrome register (XER), L1 cache control and status register (L1CSR), and etc.
- **Instruction-based features:**
  - Instruction types and data patterns of associated operands, the result of execution, and etc.
  - Information associated with load/store addresses, such as the virtual addresses and physical addresses, the attributes of the page which the addresses lie on, and etc.
- **Sequence-based features:**
  - Data dependency in a sequence of instructions, the distance between the dependent instructions, and etc.
  - Address collision in a sequence of instructions, the distance between the collided instructions, and etc.

```
stdx 4,28,15
(EA=0x00000000ee308888,RA=0x0000000020edb888)
ldx 22,22,22
(EA=0x00000000fff1d908,RA=0x0000000020edb908)
ldx 21,22,3
(EA=0x00000000fff1d888,RA=0x0000000020edb888)
```

Figure 5: Illustration of a test program snippet

Figure 5 illustrates an example showing a simplified view of a snippet from a novel test. The feature vector extracted from the third instruction is illustrated in Table 1. The subscript 3 denotes the features of the third instruction. EA_3 specifies the effective address, while RA_3 is the real address. op_type_3 refers to the instruction type. $collided_3$ is an occurrence feature indicating whether the instruction has address collision with any of previous instructions. $collision\_dist_3 = 1$ means there is one instruction between the third instruction and the closest previous collided instruction.

Feature	...	EA_3	RA_3
Value	...	0x00000000fff1d888	0x0000000020edb888

...	op_type_3	$collided_3$	$collision\_distance_3$	...
...	ldx	1	1	...

Table 1: Illustration of portion of a feature vector

4.3 Feature Discretizatition

In rule learning, a descriptive feature with numerical values is first partitioned into multiple bins to facilitate the rule search. For example, RA is a descriptive feature whose value can be partitioned into bins based on the cache line size or page size. In general, an entropy minimization heuristic developed by [7], can be employed for the partitioning such that a small range of feature values with rare occurrence is considered important and identified as a separate bin. A large range of feature values commonly-appearing in many samples are considered less important and grouped into the same bin. We use a discretization scheme based on the entropy minimization heuristic with additional constraints based on the known design features such as cache line and page sizes, etc.

5. KNOWLEDGE EXTRACTION BY RULE LEARNING

Given two classes of snippets, $S_{covered}$ (*positive* samples) and $S_{not-occurred}$ (*negative* samples), we are interested in finding the rules to describe the properties of positive samples $S_{covered}$. A rule is in the form of $Ante \Rightarrow S_{covered}$, where the class $S_{covered}$ appears in the rule consequent, and the rule antecedent $Ante$ is a conjunction of clauses $c_1 \wedge c_2 \ldots \wedge c_n$. Each clause involves a single feature. For an occurrence feature f, a clause can be either $f = 0$ or $f = 1$. For a descriptive feature f', a clause can be $f' = bin$ where bin is a bin number after the discretization described above. The $Ante$ is essentially a combination of important features selected to describe the properties on the positive samples. In principle, the $Ante$ should appear in zero or only very few negative samples. Moreover, an $Ante$ with a smaller number of clauses is preferred because such an $Ante$ is more general. An example rule based on features discussed in Table 1 is shown as follows:

$$op\_type_1 = stdx \quad \wedge \quad op\_type_3 = ldx \quad \wedge \quad collided_3 = 1 \quad \wedge \quad collision\_dist_3 = [1,2) \quad \Rightarrow \quad S_{covered} \tag{1}$$

There are two classes of rule learning algorithms: classification rule learning and association rule learning. Classification rule learning is an approach for *predictive induction* (supervised learning), aimed at constructing a set of rules to be used for classification. Association rule learning is a form of *descriptive induction* (unsupervised learning), aimed at the discovery of rules which define interesting patterns in data. Subgroup discovery aims to address a task at the intersection of predictive and descriptive induction. For descriptive induction, it identifies groups of similar samples that should be analyzed collectively. Then, for a group of multiple similar samples, predictive induction is applied to extract rules. The search iterates between descriptive induction and predictive induction to find the optimal group boundaries and rules to describe each group.

Compared to classification rule learning, subgroup discovery is more suitable for the application. A class of positive samples hitting a particular condition can be due to multiple reasons. In classification rule learning, the positive samples are analyzed collectively. But because one subset of samples may be due to one reason and another subset may be to due a different reason, it becomes difficult to find a single rule to explain most of the samples, i.e. a single rule with high accuracy. This problem is resolved in subgroup discovery by grouping similar samples and searching rules to describe each group individually.

In this work, we implement a rule search engine similar to the CN2-SD algorithm proposed by [10], which adapted the classification rule learning CN2 algorithm [5] to subgroup discovery learning in order to achieve both predictive and descriptive induction.

The rule search engine performs a breadth-first search where the depth is characterized by the number of clauses. The evaluation metric of a rule is based on a weighted relative accuracy [12] as described below.

For a rule $Ante \Rightarrow S_{covered}$, the weighted relative accuracy $WRAcc$ is defined as follows:

$$WRAcc(Ante \Rightarrow S_{covered}) = p(Ante) \cdot (p(S_{covered}|Ante) - p(S_{covered})) \tag{2}$$

$p(Ante)$ is the frequency of the total samples satisfying the $Ante$. $p(S_{covered}|Ante)$ is the frequency of the positive samples satisfying the $Ante$. $p(S_{covered})$ is the frequency of the positive samples. The weighted relative accuracy consists of two components. The *relative accuracy* component ($p(S_{covered}|Ante) - p(S_{covered})$) and the *generality* component ($p(Ante)$). Therefore, the weighted accuracy provides a tradeoff between the generality of the rule (rule coverage) and the relative accuracy.

In classification rule learning, covered samples are dropped to avoid finding the same rule again. However, a single sample may attribute to two reasons for hitting the condition. If such a sample is dropped after uncovering one reason, its information is lost for uncovering the other reason. To address this problem, the rule search engine uses a weighed covering heuristic. Instead of dropping a covered sample, it stores the covered sample with a weight indicating how many times the sample has been covered, i.e. how many rules have been produced based on the sample. Then, in Equation (2) the frequencies are adjusted based on these weights. The output of the search is a ranked list of rules where the ranking can be based on several metrics [10].

6. KNOWLEDGE REUSE

6.1 Rule Validation and Refinement

From a ranked list of rules, a rule can be selected and validated by creating a test template macro satisfying the rule. A macro is a parameterized building block of a template, which specifies how instruction sequences are instantiated. For example, the rule in Equation (1) can be encoded into a macro illustrated in Figure 6, which will generate a pair of *stdx*-*ldx* collision with a random instruction between them.

```
sequence:
    var a = random()
    gen_inst(optype=stdx, addr=a)
    gen_inst()
    gen_inst(optype=ldx, addr=a)
```

Figure 6: Illustration of a test template macro

A rule is evaluated based on the frequency of the produced tests hitting the desired condition. A rule is considered to be meaningful if the frequency is higher than the ratio of the number of positive samples over the total number of samples in the original dataset. The larger the difference is, the more meaningful the rule is. In the learning process, a rule can be further refined based on additional positive samples produced in the rule validation process.

6.2 Rule Reuse

Rules and macros are reused to improve the coverage of complex assertions. A database is built to store the rules and macros for each condition to be monitored. When we want to produce tests to activate an assertion comprising multiple conditions, the corresponding macros for the conditions are retrieved from the database. These macros are combined to create more complex macros for activating the assertion.

In our methodology, combining macros follows a predefined set of built-in procedures that can be selected by the user. For example, one procedure combines macros by enumerating all the orderings without interleaving instructions from two macros. Another procedure combines macros based on a given fixed ordering by interleaving the instructions from two consecutive macros in the ordering. There are variants of interleaving schemes in the procedure to decide how instructions from two macros can be interleaved.

When creating compound macros, the constraints specified by individual macros should be preserved. For example, if we combine the macro in Figure 6 with another macro, the *stdx* instruction should still proceed the *ldx* instruction. While we can interleave instructions from another macro between the *stdx* and *ldx*, the number of intercepted instructions should not exceed one.

7. EXPERIMENT RESULTS

7.1 Experiment Environment

In this work, the experiments were conducted based on a dual-threaded low-power 64-bit Power Architecture-based processor core. It is targeted to be manufactured in a 28 nm technology. The processor core supports dual-thread capability that enables each core to act as two virtual cores. Each

thread is two-way superscalar and maintains up to 16 out-of-order instructions in-flight through 10 parallel execution pipelines. The core is designed with a memory subsystem supporting up to a twelve-core SoC implementation.

The in-house simulation-based verification environment conforms to a state-of-the-art constrained-random verification flow. An in-house test generator is used to generate constrained-random test programs based on user-supplied test templates. During the test generation, architectural simulation is also performed and the simulation results are embedded in test programs. During the RTL simulation, the RTL simulation results are compared with the architectural simulation results for checking correctness.

The experimental results shown below focus on the assertion coverage of the Load Store Unit (LSU). LSU is one of the most complex and difficult-to-verify units in the design. Since the experiment was conducted in parallel with the ongoing verification efforts, the experiment started with the best test templates that had been refined by the verification team up to the time of the experiment. In the following, we describe three results in detail to demonstrate the effectiveness of the proposed learning methodology.

7.2 The First Result

The first result demonstrates the following: The learning began with novel tests activating an assertion A comprising a single condition c_1. Learning was to extract rules for hitting c_1. After the learning, two things were accomplished by the tests instantiated from the refined test template. First, the frequency of activating assertion A was substantially improved. Moreover, two additional assertions B, C (with zero coverage before) were covered.

The assertion B comprises a single condition c_2 that is highly correlated to the condition c_1. The assertion C comprises both conditions c_1 and c_2. The result demonstrates that learning from tests activating one assertion can lead to fortuitous coverage of other correlated assertions.

In the simulation run, 1000 tests were instantiated from a test template based on 114 types of memory instructions. Each test consists of 50 instructions. The simulation time for each test on a single machine took several minutes. When simulating using a server farm, 20-30 tests could be simulated simultaneously. The assertion A was covered by merely three tests. This assertion refers to the special condition c_1 concerning how certain queues in LSU are filled up.

We applied the learning methodology to extract rules from the three novel tests. One interesting rule we found can be interpreted as follows:

- There is a *lmw* (load multiple word) instruction.
- The page on which the real address of *lmw* lies, is not cache inhibited.
- The destination register of the *lmw* is before $G20$.

The rule is converted into a test template macro and used to generate another 200 tests, each comprising 50 instructions. Table 2 shows the comparison of coverage between the original 1000 tests and 200 new tests. As shown by the 4th row, the number of tests activating the assertion A increases from 3 to 33 in the new test set, thus boosting the frequency from 0.3% to 15.5%. Moreover, assertions B and C which were not activated by the original 1000 tests could be activated by 9 tests and 5 tests from the 200 new tests (the 5th and 6th rows).

	# of tests		% of tests	
test set	original	new	original	new
size	1000	200	1000	200
assertion A	3	33	0.3%	15.5%
assertion B	0	9	0	4.5%
assertion C	0	5	0	2.5%

Table 2: Comparison of assertion coverage between original 1000 tests and 200 new tests

7.3 The Second Result

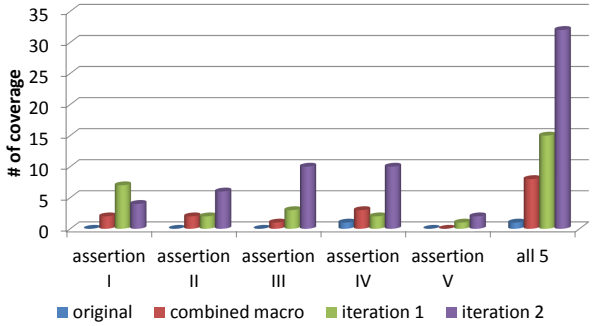

Figure 7: Assertion coverage improvement

Figure 7 summarizes the second result. In the original simulation run, 2000 tests were instantiated from a template based on 44 types of memory instructions and over 200 types of non-memory instructions. Each test consists of 100 instructions. In the simulation, only assertion IV was activated by one test. Assertion I, II, III, and V had zero coverage.

Assertion IV comprises two conditions c_3 and c_4. Other assertions comprise the same two conditions. However, the temporal constraints between the two conditions are different across the five assertions. c_3 and c_4 last only one or two clock cycles when asserted. The temporal constraints require them to occur within a small time window, which makes the assertions difficult to activate.

Learning was carried out based on the novel tests hitting conditions c_3 and c_4. Note that there were multiple tests hitting c_3 and c_4 individually. After the learning, rules were extracted for hitting c_3 and c_4, resulting in multiple macros for each condition.

Two macros m_1 and m_2 (for c_3 and c_4, respectively) were identified to be consistent with two respective segments of instructions in the one test activating assertion IV. The corresponding rules for these two macros are illustrated in Table 3. Hence, macros m_1 and m_2 were combined to produce a new template macro. Because in the test, instructions from m_1 was followed by instructions from m_2 without interleaving, in the combined macro, m_1 was followed by m_2 without instruction interleaving.

Rule for m_1	There is a *mulld* instruction and the two multiplicands are larger than 2^{32}
Rule for m_2	There is a lfd instruction and the instructions prior to the lfd are not memory instructions whose addresses collide with the lfd

Table 3: Rules for macros m_1 and m_2

The combined macro was used to produce 100 new tests. These new 100 tests led to higher coverage for assertions I to IV as shown with the legend "combined macro" in Figure 7. But assertion V remained at zero coverage.

The learning was re-applied with the additional 100 tests and new rules/macros were obtained for hitting c_3 and c_4.

Again, 100 new tests were produced. The result was denoted as "iteration 1" in Figure 7. Observe in "iteration one" that coverage for assertion I to IV was improved further. More importantly, assertion V could be covered. The process repeated in the "iteration 2" and we can observe further coverage improvement for assertions II to V.

7.4 The Third Result

The original simulation was based on 2000 tests instantiated from a given test template. For the 10 assertions of interest, none of them was covered. The simulation was expanded with tests instantiated from multiple other test templates. In total, over 30k tests were simulated without activating any of the 10 assertions. Then, the learning was carried out based on the 2000 tests in the first simulation run.

The 10 assertions comprise the same two conditions, c_5 and c_6, and the temporal constraint between c_5 and c_6 varies across the 10 assertions. Also, c_5 and c_6 are evanescent and are required to happen with a small time window. While none of the tests activate an assertion, multiple tests could hit each condition individually. Table 4 illustrates some of the rules learned for c_5 and c_6:

Rules for condition c_5	
Rule R_1	There is an *isync* instruction.
Rule R_2	There is a *mcrxr* instruction followed by a *mullwo* instruction. The *mullwo* instruction causes an overflow in XER.
...	...
Rules for condition c_6	
Rule Q_1	A *stmw* instruction, then a *stdx* instruction followed by a *ldx* with address collision.
...	...

Table 4: Illustration of rules for c_5 and c_6

A macro from a rule for c_5 was combined with instruction interleaving with a macro from a rule for c_6 to produce a combined macro. In total, 12 combined macros were produced based on 4 rules for c_5 and 3 rules for c_6. Then, 1200 tests were generated, 100 tests based on each combined macro. The third column of Table 5 shows that 6 out of the 10 assertions were covered by these 1200 tests.

test set	original	combined macros	refined macro
# of tests	>30k	1200	100
cycles($\times 10^4$)	>1000	40.8	4.23
assertion 1	0	1	2
assertion 2	0	0	1
assertion 3	0	0	1
assertion 4	0	16	56
assertion 5	0	0	1
assertion 6	0	1	2
assertion 7	0	0	1
assertion 8	0	16	56
assertion 9	0	15	61
assertion 10	0	26	77

Table 5: Comparison of assertion coverage

Analyzing the tests activating the assertion led to the conclusion that a combination of particular two rules (R_2 and Q_1) with a particular instruction interleaving scheme is especially effective. A new macro (illustrated in Figure 8) was created based on the two rules by fixing the particular interleaving scheme. Additional 100 tests were produced using the new macro and the result was shown in the 4th column of Table 5 where all 10 assertions were covered.

```
sequence:
    var a = random()
    gen_inst(optype=mcrxr)
    gen_inst(optype=stmw)
    gen_inst(optype=stdx, addr=a)
    gen_inst(optype=mullwo, result=overflow)
    gen_inst(optype=ldx, addr=a)
```

Figure 8: Illustration of the effective macro for activating the assertions

8. CONCLUSION

This work proposes a learning methodology to extract knowledge from simulation in constrained-random processor verification. A feature-based rule learning approach is developed for the knowledge extraction. The extracted knowledge is reused for test template refinement to improve assertion coverage. Experimental results demonstrate the effectiveness of the proposed learning methodology in various scenarios where assertion coverage could be further improved after a substantial verification effort had been spent.

9. REFERENCES

[1] P. Bastani and et al. Diagnosis of design-silicon timing mismatch with feature encoding and importance ranking - the methodology explained. In *IEEE International Test Conference*, pages 1 –10, oct. 2008.

[2] P.-H. Chang and et al. Online selection of effective functional test programs based on novelty detection. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 762 –769, nov. 2010.

[3] J. Chen and et al. Mining ac delay measurements for understanding speed-limiting paths. In *Test Conference (ITC), 2010 IEEE International*, pages 1 –10, nov. 2010.

[4] W. Chen and et al. Novel test detection to improve simulation efficiency–a commercial experiment. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, nov. 2012.

[5] P. Clark and T. Niblett. The cn2 induction algorithm. *Mach. Learn.*, 3(4):261–283, Mar. 1989.

[6] K. Eder, P. Flach, and H.-W. Hsueh. Inductive logic programming. chapter Towards Automating Simulation-Based Design Verification Using ILP, pages 154–168. Springer-Verlag, Berlin, Heidelberg, 2007.

[7] U. M. Fayyad and K. B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029, 1993.

[8] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Design Automation Conference, 2003. Proceedings*, pages 286 – 291, june 2003.

[9] Y. Katz, M. Rimon, A. Ziv, and G. Shaked. Learning microarchitectural behaviors to improve stimuli generation quality. In *Design Automation Conference (DAC), 48th ACM/EDAC/IEEE*, pages 848 –853, 2011.

[10] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski. Subgroup discovery with cn2-sd. *J. Mach. Learn. Res.*, 5:153–188, Dec. 2004.

[11] G. Squillero. Microgp-an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, Sept. 2005.

[12] L. Todorovski, P. A. Flach, and N. Lavrac. Predictive performance of weghted relative accuracy. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '00, pages 255–264, London, UK, UK, 2000. Springer-Verlag.

[13] I. Wagner, V. Bertacco, and T. Austin. Microprocessor verification via feedback-adjusted markov models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(6):1126 –1138, june 2007.