UNIVERSITY OF CALIFORNIA

Santa Barbara

Data Learning Methodologies for Improving the Efficiency of Constrained Random Verification

A dissertation submitted in partial satisfaction of the

requirements for the degree

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Wen Chen

Committee in charge:

Professor Li-C. Wang, Chair

Professor Forrest Brewer

Professor Malgorzata Marek-Sadowska

Dr. Jayanta Bhadra

September 2014

The dissertation of Wen Chen is approved.

Professor Forrest Brewer

Professor Malgorzata Marek-Sadowska

Dr. Jayanta Bhadra

Professor Li-C. Wang, Committee Chair

June 2014

Data Learning Methodologies for Improving the Efficiency of Constrained Random Verification

Wen Chen

Dedicated to my mom and dad

Acknowledgements

First of all, I express my deepest appreciation and gratitude to my advisor Professor Li-C. Wang for his continuous guidance and tremendous patience through the past four years. Not only did he teach me how to conduct solid research, but he also guided me to becoming a more mature person in life. The mindset and philosophy I developed while working with Li will be definitely beneficial for a lifetime.

I would like to thank my colleagues in Freescale Semiconductor Inc. for their help during my research internships there. I owe special thanks to Dr. Jay Bhadra, my manager and PhD committee member, for his mentorship and enormous help with my research and internships. I am also thankful to Dr. Magdy Abadir and Dr. Shaun Feng for their mentorship and support for my projects. Also, I would like to thank Sanjay Gupta, Robert Page, Daniel Pinero, Prashant Bansal and Larry McConville for their technical help with setting the verification environment.

I would like to thank the faculty at the University of California, Santa Barbara. Specially, I am grateful to the members of my PhD committee, Professor Forrest Brewer and Professor Margaret Marek-Sadowska, for their guidance and valuable feedback.

I feel very fortunate to meet my adorable labmates in Li's group. Particularly, I would like to acknowledge Po-Hsieh Chang and Nik Sumikawa for their help when I started as a junior graduate student. I would like to thank Jeff Tikkanen, Gagi Drmanic, Vinayak Kamath, Sebastian Siatkowski, Kuo-Kai Hsieh and Chia-Ling Chang for discussions and collaborations.

I really appreciate the love and support from all my friends. I would like to thank Yang Li, Yang Lin, Liming Chen and many other friends here at UCSB for their company and comfort during the past four years. I also owe thanks to Gang Li, Qiang Xu, Tao Mao, Cheng Zhuo, Jia Zeng and many other friends who are not here, for their spiritual support for my PhD endeavor.

Last but certainly not the least, I dedicate this dissertation to my parents, with my gratitude for their unconditional support and care in my lifetime.

Curriculum Vitae

Wen Chen

EDUCATION

2010 − 2014	PhD in Electrical and Computer Engineering, University of California, Santa Barbara.
2008 − 2010	M.S. in Computer Science and Engineering, University of Michigan, Ann Arbor.
2004 − 2008	B.S. in Electrical Engineering, Zhejiang University, Hangzhou, China.

PUBLICATIONS

1. Wen Chen, Li-C. Wang, Jay Bhadra, Madgy S. Abadir, *Simulation Knowledge Extraction and Reuse in Constrained Random Verification,* ACM/IEEE Design Automation Conference (DAC), June 2013

2. Wen Chen, Li-C. Wang, Jay Bhadra, Madgy S. Abadir, *Novel Test Analysis to Improve Structural Coverage A Commercial Experiment,* in Proc. IEEE/ACM International Symposium on VLSI Design, Automation and Test (VLSI-DAT), April 2013

3. Vinayak Kamath, Wen Chen, Nik Sumikawa, Li-C. Wang, *Functional test content optimization for peak-power validation - An experimental study,* IEEE International Test Conference (ITC), Nov 2012

4. Wen Chen, Nik Sumikawa, Li-C. Wang, Jayanta Bhadra, Xiushan Feng, Magdy S. Abadir, *Novel test detection to improve simulation efficiency - A commercial experiment,* IEEE/ACM International Conference on Computer Aided Design (ICCAD), Nov 2012

FIELD OF STUDY

Electrical and Computer Engineering Professor Li-C. Wang

Abstract

Data Learning Methodologies for Improving the Efficiency of Constrained Random Verification

by Wen Chen

Functional verification continues to be one of the most time-consuming steps in the chip design cycle. Simulation-based verification is well practised in industry thanks to its flexibility and scalability. The completeness of the verification is measured by coverage metrics. Generating effective tests to achieve a satisfactory coverage level is a difficult task in verification. Constrained random verification is commonly used to alleviate the manual efforts for producing direct tests. However, there are yet many situations where unnecessary verification efforts in terms of simulation cycles and man hours are spent. Also, it is observed that lots of data generated in existing constrained random verification process are barely analysed, and then discarded after simplistic correctness checking. Based on our previous research on data mining and exposure to the industrial verification process, we identify that there are opportunities in extracting knowledge from the constrained random verification data and use it to improve the verification efficiency.

In constrained random verification, when a simulation run of tests instantiated by a test template cannot reach the coverage goal, there are two possible reasons: insufficient simulation, and improper constraints and/or biases. There are three actions that a verification engineer can usually do to address the problem: to simulate more tests, to refine the test template, or to change to a new test template. Accordingly, we propose three data learning methodologies

to help the engineers make more informed decisions in these three application scenarios and thus improve the verification efficiency.

The first methodology identifies important ("novel") tests before simulation based on what have been already simulated. By only simulating those novel tests and filtering out redundant tests, tremendous resources such as simulation cycles and licenses can be saved. The second methodology extracts the unique properties from those novel tests identified in simulation and uses them to refine the test template. By leveraging the extracted knowledge, more tests similar to the novel ones are generated. And thus the new tests are more likely to activate coverage events that are otherwise difficult to hit by extensive simulation. The third methodology analyses a collection of existing test items (test templates) and identifies feasible augmentation to the test plan. By automatically adding new test items based on the data analysis, it alleviates the manual efforts for closing coverage holes.

The proposed data learning methodologies were developed and applied in the setting of verifying commercial microprocessor and SoC platform designs. The experiments in this dissertation were conducted in the verification environment of a commercial microprocessor and a SoC platform in Freescale Semiconductor Inc. and were in parallel with the on-going verification efforts. The experiment results demonstrate the feasibility and effectiveness of building learning frameworks to improve verification efficiency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Functional verification is acknowledged as a key bottleneck in the chip design cycle and industry has witnessed soaring sizes of verification teams [1]. The increasing sizes and complexities of emerging multi-core System-on-Chip (SoC) have placed greater challenges on verification [2][3][4]. Numerous efforts and resources have been dedicated to verifying the increasingly complicated microprocessors and the integration of heterogeneous cores at the SoC level. Hence, there is an enormous need for the development of advanced verification technologies.

Although the application of formal methods in verification has made remarkable advances, extensive simulation is still the most applicable for full-chip verification due to its scalability and flexibility. A typical verification flow includes a process of stimulus generation, simulation, result checking and coverage collection. One prevalent approach to simulation-based verification is

Constrained Random Verification (CRV), where the verification engineers encode constraints and biases as test templates and instantiate them into tests. The completeness of the verification is measured by coverage metrics. A satisfactory coverage level must be met before tape-out.

One difficult task in CRV is to generate effective tests to achieve high coverage. The quality of test generation usually improves along the verification process, as the verification engineers learn more in-depth knowledge about the design. However, this effort is conducted manually, usually in a trial-and-error manner. In the meanwhile, tons of data are generated in CRV, most of which are discarded during verification iterations. It would be helpful to analyse those data and extract useful information that can aid in improving the test generation. In recent years, the advances of data mining techniques have made it possible to analyse large volumes of data in various application fields [5]. This motivates us to investigate the feasibility and effectiveness of applying data mining techniques in functional verification. More specifically, we aim to develop learning-based methodologies to improve the efficiency of constrained random verification.

1.2 Motivation

Suppose we were verifying a complex design with a constrained random verification framework. There was a particular coverage event we tried to hit. We developed a test template and let the framework to instantiate a number of tests and simulate them. After a long simulation run, we decided that it could not hit the event. We modified the test template and tried a new run. After a couple of trials, we still could not hit the event. At that point, we would

be hoping that there could be a tool that could guide us to more effectively produce a test template that could generate a test to hit the coverage event.

This is a typical scenario in constrained random verification. When we say that "my test template cannot hit the coverage hole," it can be actually due to two reasons:

Insufficient simulation How do we know that the test template is incapable of generating a test to hit the hole? Usually, we reach this conclusion by generating N tests. How do we know it will not hit the hole if we continue the simulation by generating $100 \times N$ tests?

Improper input constraints and biases If we use a test template to produce 1M tests and still none of them can hit the hole, we probably would confidently conclude that we need to change the test template. Do we start from the scratch or do we start by modifying the existing input constraints and biases?

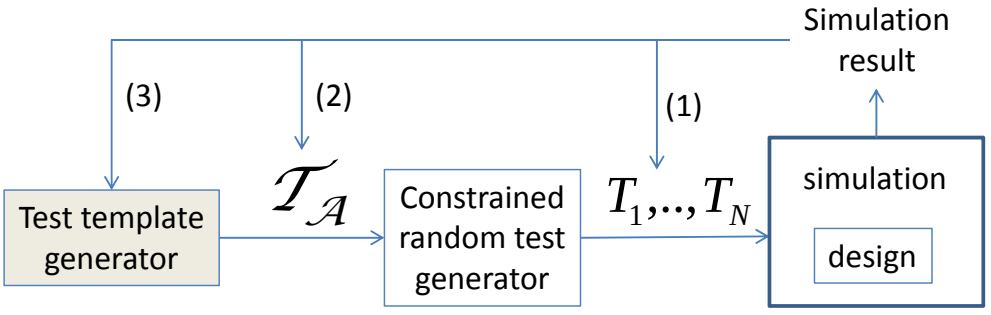

FIGURE 1.1: When result is not satisfactory, there can be three ways to improve

These two reasons imply three ways to improve the constrained random verification process when we encounter coverage holes. Figure 1.1 illustrates them. In the figure, we assume that a test template $\mathcal{T}_{\mathcal{A}}$ is instantiated into N tests

$T_1, \ldots, T_N$. Moreover, the test template is likely to be manually crafted by a verification engineer.

The first way to improve is to remove "redundancy" in $T_1, \ldots, T_N$.

For example, suppose we can afford (or decide) to simulate 2000 tests per test template. Currently, we simply instantiate 2000 tests and simulate them. Alternatively, we can instantiate 100K tests and identify the top 2000 novel ("non-redundant") tests. Intuitively, the alternative approach would be more effective.

The effectiveness can be viewed in two ways: (1) the 2000 novel tests achieve a higher coverage (or have a higher chance to hit the hole) than the original 2000 tests. (2) When we decide to stop using $\mathcal{T}_{\mathcal{A}}$, such a decision can be made with higher confidence.

The second way is to refine the test template $\mathcal{T}_{\mathcal{A}}$.

Refinement may mean to constrain and/or bias the template in such a way that the resulting tests have much higher probability for hitting a desired area (or a state) of the design. For example, a refinement increases the probability of satisfying some local conditions that we know would help to hit the hole.

The third way is to produce a new test template.

If we would like to have a tool that can automatically produce a test template for a desired target, in essence, this becomes solving the test generation problem. And we know that this would be a difficult problem. In practice, producing a new test template means someone writes the test template manually. However, when there is a collection of test templates, it is possible to

identify the insufficiency of the current collection and thus produce new test templates to augment the collection.

1.3 The Proposed Methodologies

Accordingly, we developed three learning-based methodologies to help improve the efficiency in the three application scenarios mentioned above.

The first methodology is a novel test detection framework to reduce simulation cost. A novel test is dissimilar to those tests already simulated thus is likely to provide additional coverage. A kernel is defined to represent the similarity between two tests. By using the kernel-based novelty detection model, we can efficiently capture the covered space and filter out redundant tests. The idea of the novel test detection is not new, however, designing practical kernels is at the core of the novelty detection framework. We investigate the practical implementation of the kernel-based novel test detection framework and design a coverage-based kernel that is easy to implement.

The second methodology is a feature-based rule learning framework for extracting knowledge from novel tests. The novel tests embed valuable knowledge about how to activate special conditions in simulation. By analysing novel tests against a large population of non-novel tests based on a feature set, we can extract rules that explain the specialty of the novel tests. The learned rules can be used to refine the test templates and thus produce tests that are likely to hit the functional events that had low or zero coverage.

The first two methodologies are developed in the context of microprocessor verification. They can be used in combination to solve the problem of generating effective tests for a coverage goal. Even though we do not solve the test template generation problem directly, with these two components, novel test identification and test template refinement (based on learning from novel tests), we can build an iterative flow that implements a test template search process. Figure 1.2 illustrates this idea.

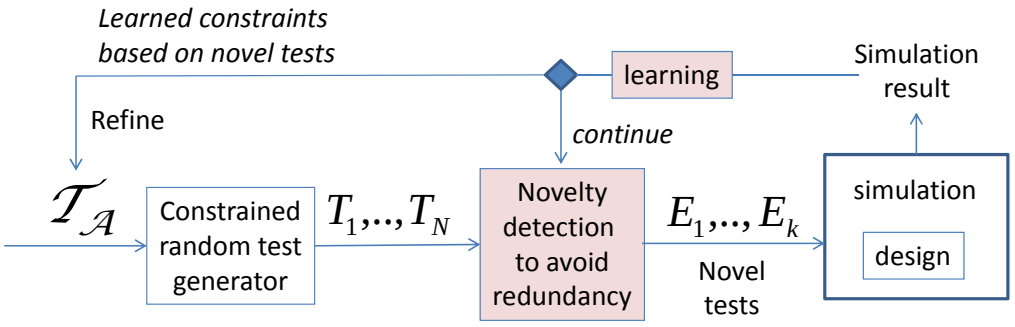

FIGURE 1.2: With non-redundant test identification and effective test template refinement, we are implementing a test template search process that can help us to find tests for hitting coverage holes ($k \ll N$)

The third methodology is a test plan augmentation approach in the context of SoC platform verification. In platform verification, the constraints and biases are encoded as test items (like test templates in processor verification). By analysing a collection of test items, we can extract knowledge that captures the underlying verification intent and thus implies feasible augmentation to the test plan. By adding new test items based on this information, we can make the test plan more complete and thus ease the manual efforts for reaching coverage closure.

The first two methodologies have been applied on top of the verification environment of a high-performance low-power microprocessor within Freescale. The third methodology has been applied in the verification of a commercial

SoC platform within Freescale. The experimental results prove the effectiveness and efficiency of the proposed methodologies and show promises of making practical tools based on them.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 provides necessary background information and reviews previous works for simulation-based verification and test generation. Chapter 3 reports the study of practical implementation of the novel test detection framework with the emphasis of kernel design. The feature-based rule learning framework is presented in Chapter 4, where the learned knowledge is used for test template refinement. The differences of processor verification and platform verification are discussed in Chapter 5, and a test plan augmentation approach for platform verification is presented. Chapter 6 concludes the dissertation and discusses future research opportunities to extend the materials addressed in this dissertation.

Chapter 2

Background and Related Works

2.1 Simulation-based Verification

Simulation-based verification is the predominant methodology for full-chip verification in industry. Today's state-of-the-art verification flow includes a highly automated flow of test generation, simulation, correctness checking and coverage collection, with islands of manual labor [6]. The verification usually starts with the creation of a verification plan, which specifies the important functionalities and aspects of the design to verify. Then verification engineers prepare stimulus/tests via various approaches. The tests are simulated using RTL models and the correctness checking is done by comparing results with those produced by reference models or by checking with built-in checkers such as assertions. The completeness of the verification is measured by various coverage metrics [7]. There are two types of coverage metrics based on the way that they are defined: those that can be automatically extracted from the design code, such as toggle coverage, and those that are user-specified in order

to tie the verification environment to the design intent or functionality [8]. The former is referred to as structural coverage and the latter is referred to as functional coverage. Generating effective tests to achieve satisfactory coverage within bounded simulation cycles is crucial in meeting the verification budget requirement. In practice, a mixture of several test generation schemes are used, which we will discuss in Section 2.2

2.2 Verification Test Generation

Direct tests are tests manually drafted by verification engineers for verifying specific scenarios. They are focus tests delicately designed for exercising particular mechanisms and thus are very effective in hitting the events of interest. However, the creation of direct tests requires a lot of manual efforts and in-depth knowledge into the design. Thus, their primary use is to hit those corner cases that cannot be well exercised in extensive simulation. In addition, scenarios not considered by the designer might be overlooked by the test developers.

Random testing is an approach to overcome the costly manual labor and biases in direct tests. However, pure random testing proves ineffective due to two reasons: (1) It generates a lot of invalid tests (2) It cannot target effectively on the events of interest since the sampling space is enormous. Constrained random verification [9] is a method that combines the verification knowledge with the power of random test generation. Constraints are used to restrict the test sampling space to ensure the tests are valid and focused on certain verification subspace. The unconstrained aspects are randomized with certain biases to increase the chances of hitting targets of interest while preserving the

potential of exposing bugs in scenarios overlooked by designers. As for microprocessor verification, a methodology called Random Test Program Generation (RTPG), originated in IBM, is designed to facilitate the constrained random verification for different processor models [10] [11]. In RTPG, a descriptive language is provided to describe the constraints and biases based on the architectural specifications and the test knowledge of the verification engineers [12]. The constraints and biases are encoded in test templates and are then fed to a test program generator. The generator converts the test generation problem into a Constraint Satisfaction Problem (CSP) and leverages CSP solvers to generate test programs satisfying the constraints [13].

Constrained random verification and direct tests are used as two prevalent approaches in industry. However, it might still be difficult to achieve a certain coverage level using both approaches. Coverage Directed Test Generation (CDTG) is a technique to generate stimulus to hit specific coverage targets. There are two approaches to CDTG: one is model-based and the other is learned-based or feedback-based [14].

In model-based CDTG, an abstract model such as Finite-State-Machine (FSM) of the design is built, and algorithms traversing the model are used to search for a path from the initial state/node to the state/node corresponding to the coverage point of interest. Commonly used models are FSM [15] [16], graph model [17], and Extracted Control Flow Machine [18]. The abstract models are either manually constructed from high-level specifications [19] [17] [15] or automatically extracted from the design. Formal methods such as model checking [15] [16], symbolic simulation [20], bounded model checking [21] are used to search the path. For the method to be applicable, the model must be

abstract enough, which affects its ability to be accurate; otherwise, the method can only be applied to relatively small designs.

The main idea behind feedback-based CDTG is to create a system that captures the relationships between coverage and inputs in a simpler but less precise manner and to combine this knowledge with the power of the random stimuli generator to generate inputs that improve the possibility of hitting the target coverage points [14]. Recent works proposed various techniques to learn from the simulation results. These approaches employ a variety of learning techniques such as Bayesian Networks [22], Markov Models [23], Genetic Algorithms [24] and Inductive Logic Programming [25]. However, automatically modifying the input to the test generator, based on the feedback from simulation, can be very difficult for complex designs. The feedback-based CDTG has been an active research area. A recent work in [6] proposed to learn test knowledge from micro-architectural behavior and embed the knowledge into the test generator to produce more effective tests.

Chapter 3

Kernel-Based Novelty Detection for Simulation Cost Reduction

3.1 Overview

Novel test detection is an approach to improve simulation efficiency by selecting novel tests before their application [26]. Techniques have been proposed to apply the approach in the context of processor verification [27]. This chapter reports our experience in applying the approach to verifying a commercial processor. Our objectives are threefold: to implement the approach in a practical setting, to assess its effectiveness and to understand its challenges in practical application. The experiments are conducted based on a simulation environment for verifying a commercial dual-thread low-power processor core. By focusing on the complex fixed-point unit, the results show up to 96% saving in simulation time. The main limitation of the implementation is discussed based

on the load-store unit with initial promising results to show how to overcome the limitation.

3.2 Introduction

In a practical simulation-based verification environment, one of the most challenging tasks is to produce the tests that lead to the desired coverage level. One common practice is to manually produce direct tests targeting on specific coverage items. For processor verification, another common approach is constrained random test program generation in which users provide constraints and biases in the form of test templates and directives to the test generator [9]. The input to the test generator specifies the sampling scheme for various dimensions in the test space such as address selection, register dependencies, arithmetic data selection, etc.

Coverage-directed test generation (CDTG) is an emerging approach to overcome the test generation problem. CDTG techniques dynamically analyse coverage results and automatically adapt the test generation process to improve the coverage. Recent works proposed various techniques to learn from the simulation results and improve the test generation. These techniques employ a variety of learning techniques such as Bayesian Networks [22], Markov Models [23], Genetic Algorithms [24] and Inductive Logic Programming [25]. In [28] the authors proposed an automatic target constraint generation technique to alleviate the burden of constraint generation.

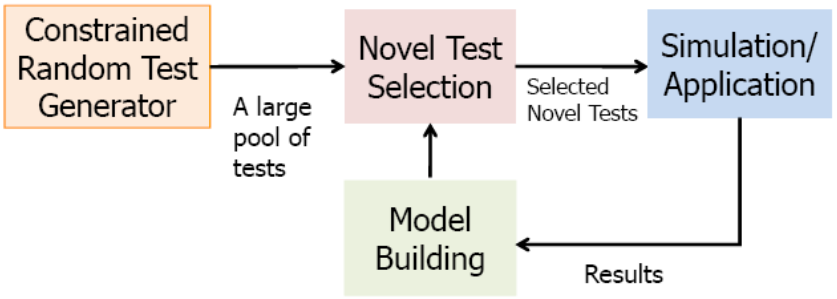

FIGURE 3.1: Illustration of novel test detection

Novel test detection tries to tackle a problem much more restricted than CDTG. Figure 3.1 illustrates the approach. In a novel test detection framework, the assumption is that there is a constrained random test generator that can instantiate the test template to generate a large number of functional tests. The idea is to learn a novel test detection model based on the results from tests that have already been applied. This model is used to select novel tests from the large pool of tests before their application. Hence, only the selected novel tests are applied, which reduces the simulation cost.

The authors in [26, 29] proposed a novel test detection framework where Support Vector Machine (SVM) one-class algorithm [30] is used to build models. The framework is limited to analysing fixed-cycle functional tests. The authors in [27] extended the application to build novel test detection models where tests are assembly programs and the context is for processor verification. The experiments were conducted based on a rather simple Plasma/MIPS processor design.

The objective of the work in this chapter is not to claim that novel test detection is better than the existing approaches for improving constrained random test generation. In fact, novel test detection can be viewed as complementary to constrained random test generation and to CDTG. Our objective instead is to assess the applicability and effectiveness of novel test detection in a practical

setting. We began by implementing the approach proposed in [27] in a company's in-house simulation environment for a dual-thread low-power processor. The experiments were conducted parallel to the ongoing verification efforts.

In this chapter, we explain the main findings based on the commercial experiment. These findings are organized into the remaining flow of the chapter as the following:

- Section 3.3 presents simulation results to illustrate the existence of novel tests in the particular simulation-based processor verification environment.
- Section 3.4 reviews the approach proposed in [27], in particular the *graph-based kernel* method used to measure similarity on a pair of assembly programs. Applying the approach to the complex fixed-point unit demonstrates up to 80% potential saving in simulation time.
- Section 3.5 discusses the major challenge of applying the graph-based kernel approach in practice. To implement the graph-based kernel demands a user to manually implement a *cost table* defining the similarity between every pair of instructions in consideration. To overcome this challenge, an alternative approach based on estimating coverage of an assembly program is proposed. This alternative approach implements a flow that requires minimal user involvement. The implementation is also easier. We demonstrate that this alternative approach can be as effective as the graph-based kernel approach and delivers up to 96% potential saving in simulation time.
- Section 3.6 discusses the main limitation of the alternative approach and proposes an extension to overcome the limitation. The effectiveness of

this extension is shown based on an experiment on a module in the load-store unit with a potential 96% saving in simulation time.

- Section 3.7 summarizes this chapter.

3.3 The Experimental Framework and Novel Tests

The experiments in this chapter and next chapter were conducted based on a dual-thread low-power 64-bit Power Architecture-based processor core. It was 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 has dedicated Fetch, Decode, Issue, and Completion resources. Each thread also has a dedicated Branch Unit, Load Store Unit, and Simple Fixed Point unit. The Complex Fixed Point unit as well as the Floating Point Unit and the vector engine are shared between threads. The core is designed with a memory subsystem supporting up to an eight-core implementation in a multiprocessing system.

The in-house simulation-based verification environment conforms to a state-of-the-art coverage-driven 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. The RTL simulation results are compared with the architectural simulation results for checking correctness. The coverage information is recorded and reported using a commercial coverage analysis tool. The verification coverage space is divided into subspaces.

A subspace can be a part of the design, e.g., a particular unit or a specific mechanism such as memory collision, etc. In our experiments, we focused on the toggle coverage of the Complex Fixed Point unit (CFX) and the Load Store Unit (LSU). Test templates targeting on these units are provided by the verification team and are used in constrained random test generation.

3.3.1 Existence of Novel Tests in Practice

Figure 3.2 shows three plots for three simulation runs, two on CFX and one on LSU. The x-axis shows the number of tests simulated, incremented by 30 at a time. The y-axis shows the normalized coverage based on the maximum coverage achieved for the respective unit in all experiments.

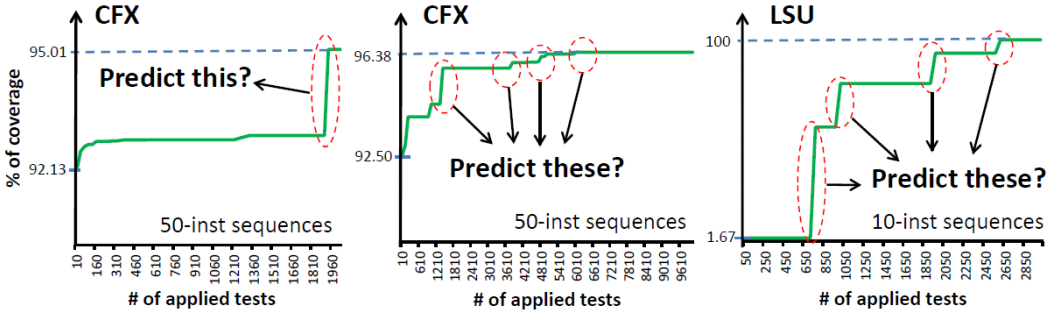

FIGURE 3.2: Three simulation runs to illustrate the existence of novel tests

For the CFX, the first run consists of 2000 test programs each with 50 instructions and an initial machine state. The test programs are instantiated from a template based on 33 instructions targeting on the unit. The second run is similar, consisting of 10K test programs each also with 50 instructions and an initial state. For the LSU, the run consists of 3000 test programs each with 10 instructions and an initial state. The template is based on 6 instructions targeting on the unit.

In all three plots, we observe jumps in the coverage curves. These jumps are due to special tests that provide relatively significant coverage at the given simulation point. These special tests are the novel tests that we are looking for. If they can be identified before simulation, they can be applied earlier in the simulation run. As a result, the respective coverage can be achieved much faster.

Take the first plot as an example. We see that the jump occurs after simulating 1900 tests. We also see that the coverage curve is flat from 1300 to 1900. Suppose an engineer uses the template to instantiate 1600 tests and observes the flat curve. It is likely that the engineer would decide it is not effective to continue. Then, the coverage jump would have been missed. If we have the ability to predict novel tests before simulation, we can generate a much larger number of tests to begin with and consequently reduce the chance of missing a test capable of producing a significant coverage increase.

The three plots in Figure 3.2 show the existence of novel tests in practical simulation-based verification scenarios. This gives a clear motivation to apply novel test detection to identify those tests before simulation.

3.4 The Graph-based Kernel Approach

3.4.1 Kernel Based Learning with SVM One-class

Support Vector Machine (SVM) one-class algorithm, such as the ν-SVM algorithm [30] is an unsupervised learning method that builds a model to identify outliers in a given set of samples. The parameter ν is a user-supplied input

that represents an upper bound on the number of outliers and low bound on the number of *support vectors*. In application with n samples, we typically set ν to be $\frac{1}{n}$ meaning that we want to build a model to incorporate at least $n-1$ samples, i.e. with at most one outlier.

In applying ν-SVM in novel test selection, the samples are tests that have been simulated up to the point of simulation. Suppose they are $t_1, ..., t_m$. A SVM model when applying to an un-simulated test T takes the following form:

$$M(T) = \sum_{i=1}^{m} \alpha_i K(T, t_i) - \rho$$

Conceptually, one can consider each α_i as a weight denoting the importance of test t_i in the calculation of the model. A test t_i is a *support vector* if $|\alpha_i| > 0$. Otherwise, it is a non-support vector, meaning that it is not used in the calculation. The ρ is a constant denoting the boundary of the measured outlier value for a test T. If $M(T) < 0$, T is deemed dissimilar to the simulated tests $t_1, ..., t_m$. The more negative the $M(T)$ is, the more dissimilar the test T is to $t_1, ..., t_m$. Given a set of un-simulated tests $T_1, \ldots, T_n$, let $M(T_j)$ be the most negative value computed by the model. Then, test T_j is the most novel test selected by the model.

The function $K(T, t_i)$ is called a *kernel* function used to measure similarity between T and t_i, i.e. a pair of tests. The novel tests selected by a SVM model highly depend on the definition of the kernel function. The kernel function dictates the perspective of what novelty means.

Suppose our objective is to cover a set S of coverage items. Suppose test T covers the subset S_T. Suppose test t_i covers the subset S_{t_i}. Intuitively, the

similarity between T and t_i can be measured as $\frac{|S_T \cap S_{t_i}|}{|S_T \cup S_{t_i}|}$. For t_i, S_{t_i} is known. However for T, S_T is unknown because it has not yet been simulated.

The SVM one-class is a kernel-based learning method [31]. Such a method consists of two components, a kernel function used to measure similarity between a pair of samples and an optimization engine used to build the model. Figure 3.3 illustrates the learning approach.

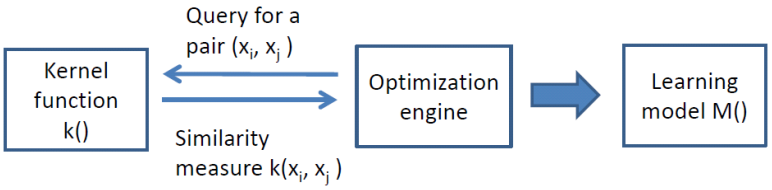

FIGURE 3.3: Illustration of kernel-based learning

The SVM one-class algorithm concerns how to find the best values for $\alpha_1, \ldots, \alpha_m$ and ρ, based on a given kernel function. As shown in Figure 3.3, such an algorithm access the kernel function by querying the similarity between a pair of samples x_i, x_j. In application, one can alter the kernel definition without changing the SVM algorithm in order to influence the model building process.

3.4.2 The Coverage-independent Graph-based Kernel

Developing an appropriate kernel is at the core of applying the kernel-based learning algorithm. In our application, tests are assembly programs. Hence, the kernel function $K()$ needs to measure similarity between a pair of assembly programs. The work in [27] proposes a graph-based kernel that computes a similarity measure by analysing two assembly programs. It is important to note that such a graph-based kernel does not rely on any coverage information by a test in the calculation. Hence, it is a coverage-independent kernel.

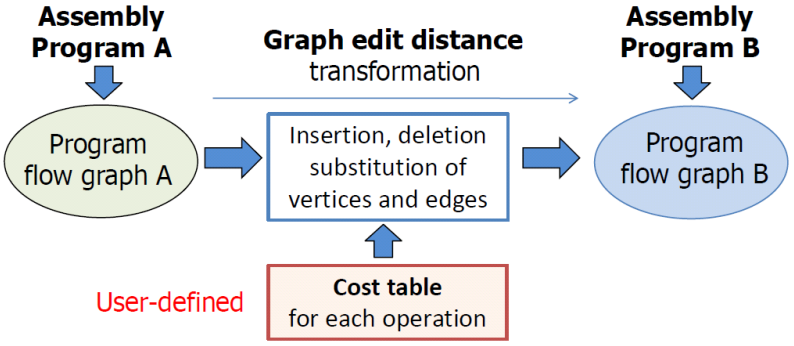

FIGURE 3.4: The framework of computing graph-based kernel

Figure 3.4 illustrates the graph-based kernel. Each assembly program is first converted into a *program flow graph*, a directed graph capturing the possible execution flows of the program. Then, the kernel calculates the similarity between two programs based on the *graph edit distance* (GED) of the two graphs. The larger the distance is the more dissimilar the two programs are. The GED is measured as the minimal cost of using a number of operations to transform one graph to the other. These operations include insertion, deletion, and substitution of vertices and edges. Each operation when performed has a cost value. The cost is defined in a *cost table*. For example, the cost of substitution of an addition instruction to a subtraction is smaller than the cost of substitution of an addition to a load/store instruction. This is because both addition and subtraction utilize the same execution unit while the load/store instruction utilizes the load-store unit.

Because the graph-based kernel is coverage independent, for a given cost table the process of building the model is fixed and consequently the novel tests detected by the model are fixed. This means that in order to apply the graph-based kernel to a given scenario, it is important to have a proper cost table. This cost table can be design dependent, unit dependent and coverage metric dependent. While this provides the flexibility to tackle a variety of scenarios,

it can also be a challenge for its user to develop a proper cost table in practice.

3.4.3 Model Building and Novelty detection

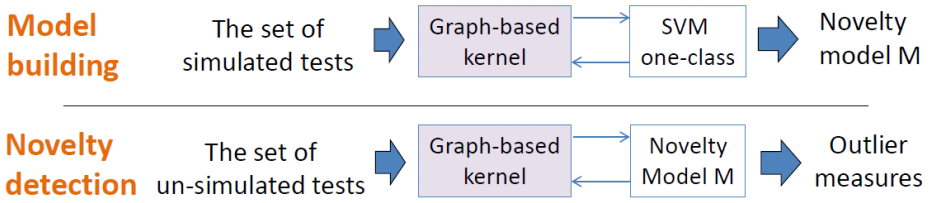

FIGURE 3.5: The framework of graph-based kernel

Figure 3.5 illustrates the model building and novelty detection processes. In model building, a model is built on a set of simulated tests. In novelty detection, the model is applied to a set of un-simulated tests to calculate an outlier measure for each test. These measures are used to rank tests. The most outlying k tests are selected and simulated. For example, the process is iterative as shown in Figure 3.1 where in each iteration the most outlying k tests are selected for simulation.

3.4.4 Experiment Results

The novel test detection framework using the graph-based kernel approach is implemented and integrated with the in-house simulation environment. Discussions in this section focus on the example shown in the first plot in Figure 3.2, i.e. the case with 2000 test programs for the CFX unit.

The novel test detection is applied iteratively where each iteration selects 30 tests to simulate from the pool of un-simulated tests. Figure 3.6 compares the coverage curves achieved with and without the novelty detection. The curve without is the same as that shown in Figure 3.2.

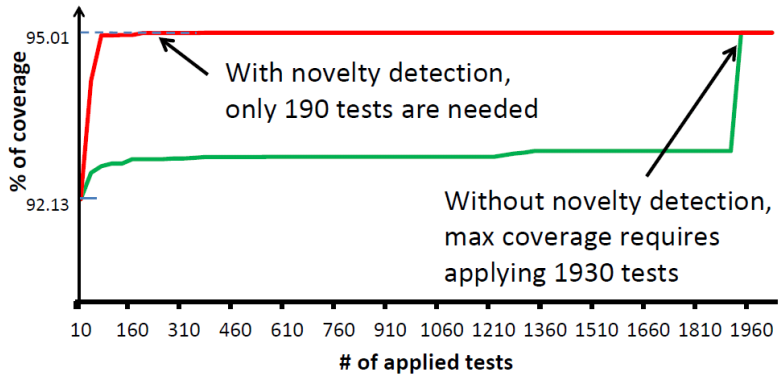

FIGURE 3.6: Comparison of coverage curves with and without novelty detection

Without the novel test detection, the original simulation achieves a maximal coverage with 1930 tests. With the novel test detection, the same coverage is achieved using 190 tests, a 90% saving (i.e. 1-$\frac{190}{1930}$). The simulation time of 2000 tests is more than a day (using a single machine). This means that with the novel test detection, a day of single-machine simulation time can be reduced to less than two hours.

One may notice the huge coverage jump in the original simulation at around the 1930th test. This indicates a special test whose characteristic is quite different from that of others, i.e. involving a dramatically different sequence of instructions. This might make the novel test detection problem easier. To assess the impact of this special test on the novel test detection, we conduct a different experiment by removing this test from consideration. In this revised experiment, we consider only the first 1800 tests.

Figure 3.7 shows the results with and without novel test detection based on the 1800 tests. Observe that in this case, the novel test detection can still provide a 60% saving (i.e. 1-$\frac{520}{1300}$). The figure also confirms that the existence of the special test does make the novel test detection more effective.

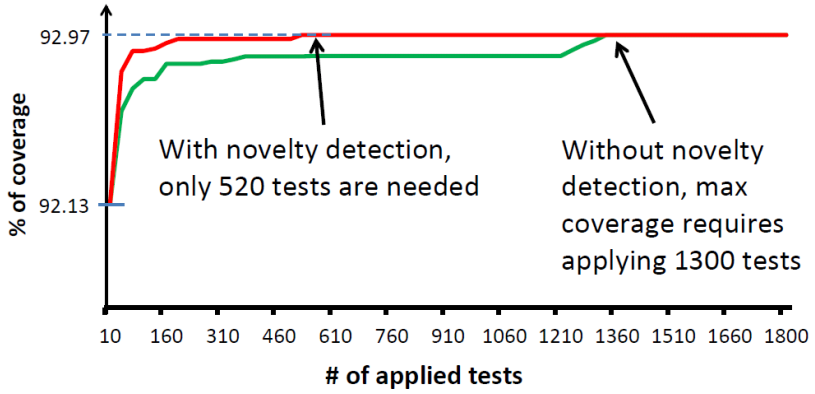

FIGURE 3.7: Comparison of coverage curves with and without novelty detection based on only the first 1800 tests in Figure 3.6

3.5 Kernel Based on Estimated Coverage

3.5.1 Disadvantage of the Graph-kernel Approach

As discussed in Section 3.4.2, the major disadvantage with the graph-based kernel approach is in the manual implementation of the cost table. Figure 3.6 and Figure 3.7 show promising results. However, these results were not obtained without noticeable effort to develop the cost table for verifying the unit. Such a development may take days or weeks to understand the behavior of each instruction with respect to the intended coverage space based on the target unit and/or design. Although one may argue that the development effort can be seen as a one-time cost, in practice, it represents a major obstacle for the acceptance of the approach.

3.5.2 Coverage-based Kernel

To ease the use of the novel test detection approach, what we need is a new way to compute the similarity with minimal manual involvement. This motivated us to develop an alternative kernel method based on estimated coverage.

Recall from the discussion in Section 3.4.1 that a novelty detection model is of the form: $M(T)=\sum_{i=1}^{m} \alpha_i K(T,t_i)-\rho$ where $t_1,\ldots,t_m$ are simulated tests. Such a model is learned based on $t_1,\ldots,t_m$ to decide the values on $\alpha_1,\ldots,\alpha_m$ and ρ. To calculate the similarity between a pair of simulated tests t_i, t_j, i.e. the kernel denoted as $K_c(t_i,t_j)$, we can simply let $K_c(t_i,t_j) = \frac{|S_{t_i} \cap S_{t_j}|}{|S_{t_i} \cup S_{t_j}|}$, where S_{t_i} and S_{t_j} are subsets of covered items by t_i and t_j, respectively. Note that t_i, t_j are simulated tests and hence, S_{t_i} and S_{t_j} are known. Such a calculation can be based on a given set S of items to cover in the simulation. Hence, the kernel calculation only depends on the selection of S that is much easier to obtain than the cost table. For example, S can be the toggled lines in a specific module of interest. As another example, S can be a set of hard-to-cover toggled lines after some initial simulation.

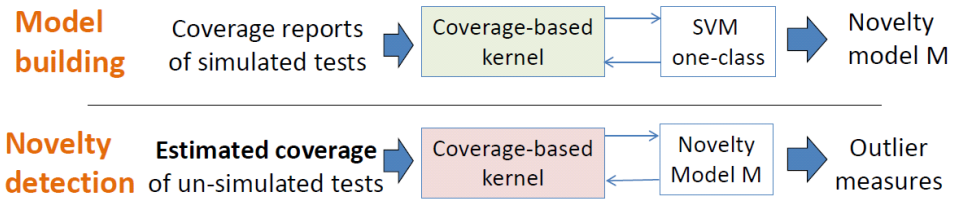

FIGURE 3.8: The framework of coverage-based kernel

Figure 3.8 illustrates the framework using the coverage-based kernel. In model building, a coverage-based kernel works well because the true coverage of each simulated test is available. In novel test detection, the model M is applied to compute an outlier measure for each un-simulated test T. This requires computing $K_c(T,t_i)$ for each support vector test t_i where the true coverage of T is not yet known. Hence, to enable the approach, we require a method to estimate coverage for an un-simulated test T.

3.5.3 Estimating Coverage Before Simulation

The idea to estimate the coverage of an un-simulated test is simple. Figure 3.9 illustrates the idea.

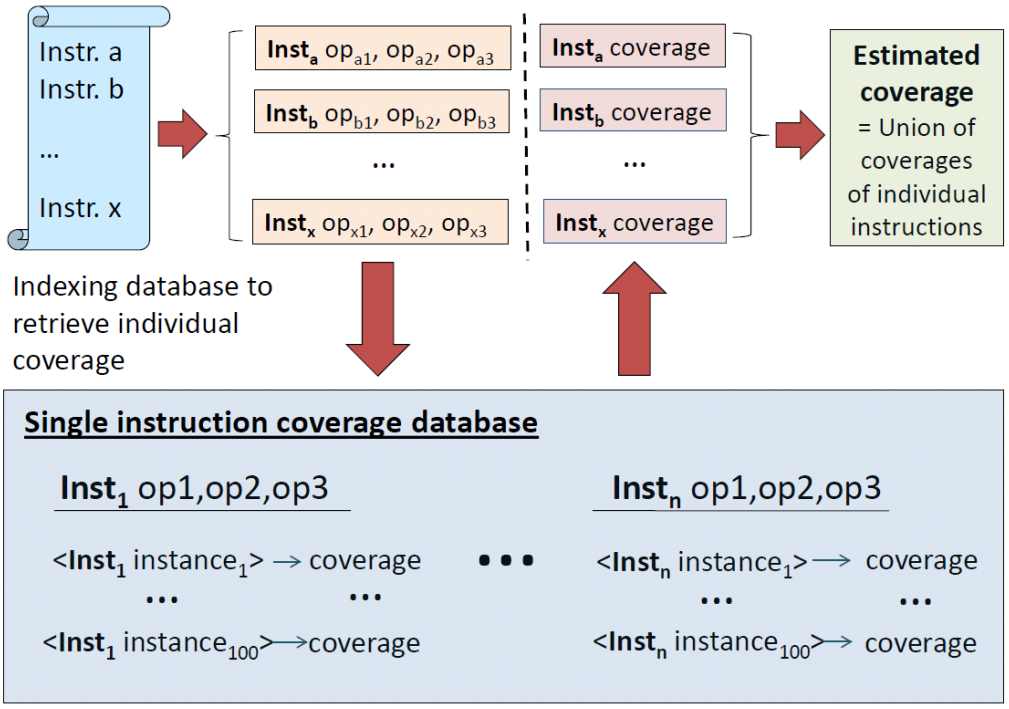

FIGURE 3.9: Illustration of coverage estimation flow

For each single instruction, we randomly instantiated h instances using the constrained random test generation framework. In the experiments, we had $h = 100$. These 100 instances were simulated and their coverages were recorded in a database. There are 600+ instructions defined by the PowerPC ISA. It took about 250 hours to build the entire single instruction coverage database. The storage requirement is about 480GB. The simulation time represents a one-time cost for the approach.

For a given un-simulated program T consisting of a sequence of instructions, for each instruction I we retrieve the coverage from the database based on

the instruction instance that is closest to the instruction I. This closeness is decided based on an *indexing function*. We implemented the indexing function to look for the closest instruction instance based on Hamming-distance calculation between the operand values of the instruction I in T and the operand values of the instruction instances stored in the database. For each instruction I in T, the indexing function decides the closest instruction instance in the database. Then, the corresponding coverage is retrieved and used for I. To estimate the coverage of T, we simply take the union of all the retrieved coverages.

It is important to note that using the union operation to estimate the coverage presents a major limitation to the approach. This limitation will be discussed in Section 3.7 later.

3.5.4 The Accuracy of Coverage Estimation

To give an idea on the accuracy of the coverage estimation method, Figure 3.10 shows a result based on the 2000 test programs used in the experiment in Figure 3.6. The x-axis shows the accuracy measured in terms of the percentage of overlap between the estimated coverage and the true coverage of a test program. The average estimation accuracy is around 75% and is far from being perfect. Later in the experimental section 3.5.6, we will show that this accuracy is sufficient for novel test detection to be effective.

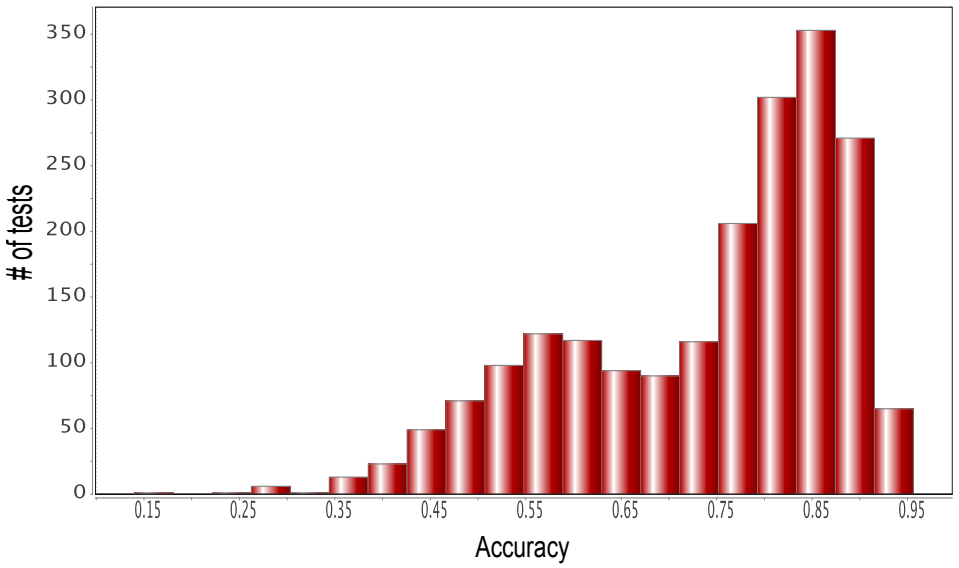

FIGURE 3.10: Histogram of estimation accuracy of 2000 tests

3.5.5 Dynamically Adjusting the Coverage Base Set S

In Section 3.5.2, we discuss the flexibility of the coverage-based kernel method. The coverage is estimated based on a set S of coverage items where this set can be flexibly defined. We call such a set the *coverage base set*.

Recall that novel test detection is an iterative process. Hence, ideally in each iteration the perspective of novelty should be defined with respect to the uncovered items. In other words, the novelty of a test should be evaluated based on its chance to provide coverage on the uncovered items.

Figure 3.11 illustrates the iterative process. Initially, a set of tests $T_1, ..., T_n$ are simulated. Then a novel test detection model M_0 is learned from the coverage results of $T_1, ..., T_n$. When applying M_0 to select the next n novel tests $T_{n+1}, ..., T_{2n}$, we would like to cover the uncovered area in the design. To achieve this effect, we can perform the following adjustment on the coverage base set S.

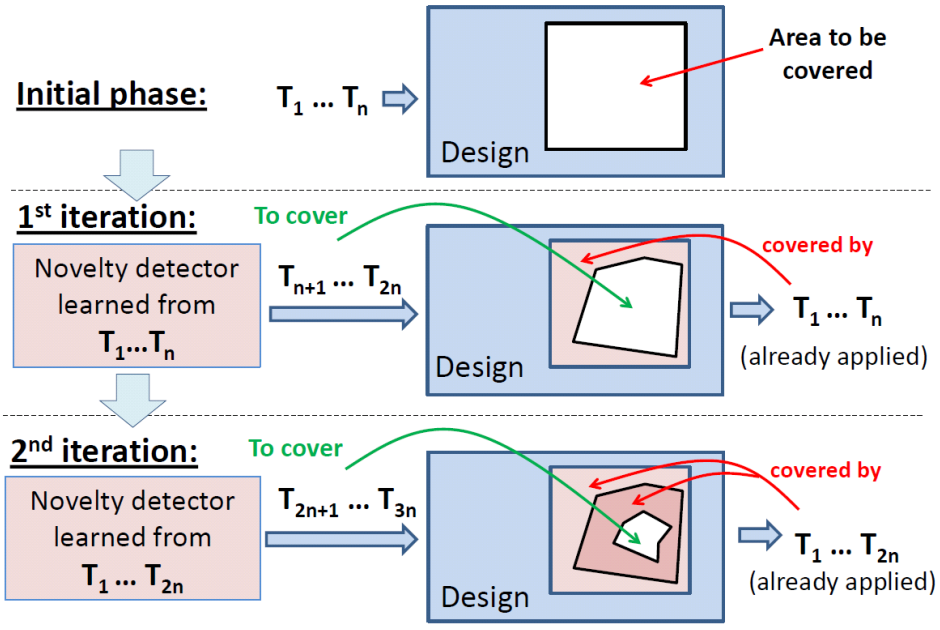

FIGURE 3.11: An ideal iterative process with novel test detection

Initially, suppose the set S contains p items $c_1, \ldots, c_p$. Let each item c_i be associated with a weight w_i initialized as 1. We calculate the coverage as $\sum w_i$ for all i such that c_i is covered by a test. Every time c_i is covered, w_i is adjusted to w_i/a where a is a constant such as $a = 2$. Such a weight adjustment scheme depreciates the importance of a covered item gradually.

Similarly, after the first iteration, a novel test detection model M_1 is learned based on all the simulated tests $T_1, ..., T_{2n}$. This model M_1 is used to select the next n novel tests $T_{2n+1}, ..., T_{3n}$ for hitting the uncovered area.

It is important to note in model building, those uncovered items do not participate in the coverage-based kernel calculation. This is because in model building, the true coverage of simulated tests is used and an uncovered item is skipped in the coverage calculation. When the model is applied to estimated coverage for an un-simulated test, an uncovered item may participate in the kernel calculation. This is because it is possible that the instruction instances retrieved from the database can hit the uncovered item.

3.5.6 Results Compared to the Graph-based Kernel Method

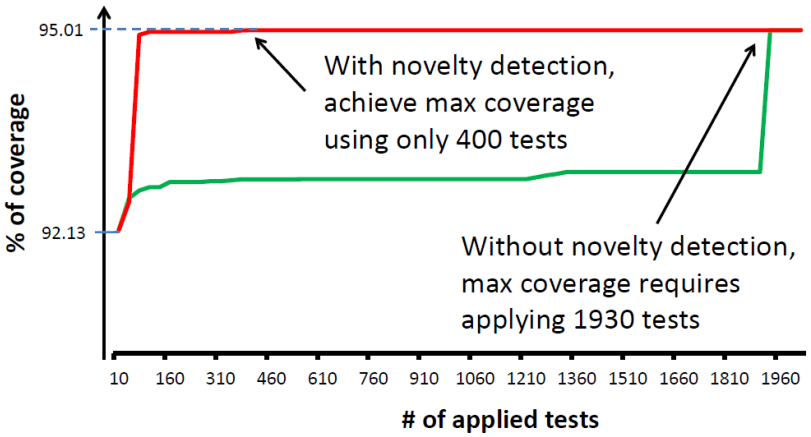

FIGURE 3.12: Comparison of coverage curves with and without novelty detection using the coverage-based kernel; The same example shown in Figure 3.6

Figure 3.12 shows the result based on the same example shown in Figure 3.6. Again, in each iteration the top 30 novel tests are selected for simulation. We see that with the novelty detection, only 400 tests are required to achieve the same coverage of using 1930 tests in the original simulation run, an 80% saving. Comparing this result to that shown in Figure 3.6, we observe the effectiveness is not as good as before. However, 80% remains a significant saving.

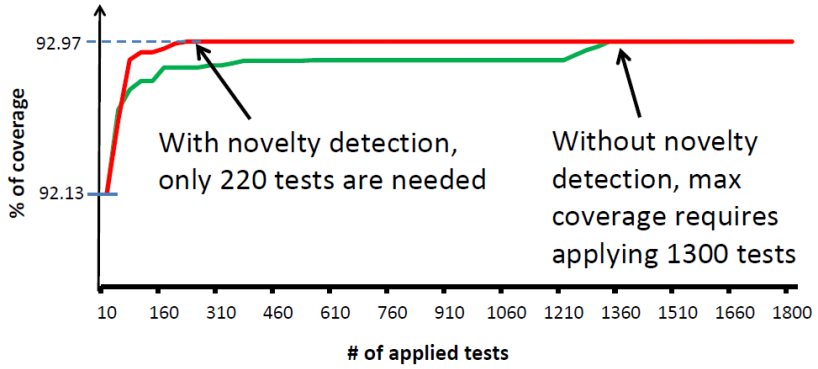

FIGURE 3.13: Comparison of coverage curves with and without novelty detection using the coverage-based kernel; The same example shown in Figure 3.7

Figure 3.13 shows the result based on the same example shown in Figure 3.7 before, i.e. using only the first 1800 tests by removing the one special test giving the big coverage jump at the 1930th test in the original simulation run. We see that with the novelty detection, only 220 tests are required to achieve the same coverage of using 1300 tests in the original simulation run, an 83% saving. Comparing this result to the 60% saving shown in Figure 3.7, the effectiveness is better than before.

3.5.7 Result on Simulation of 10K Tests

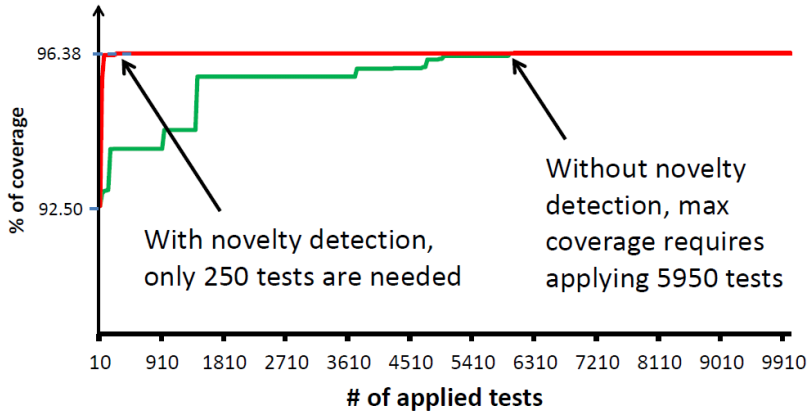

FIGURE 3.14: Comparison of coverage curves with and without novelty detection based on the middle plot example shown in Figure 3.2 before

Figure 3.14 demonstrates the effectiveness of novel test selection using the coverage-based kernel for the 10k tests simulation example shown in Figure 3.2. Without novelty detection, the maximal coverage of the original simulation run is achieved with 5950 tests. With novelty detection, the same coverage is achieved using only 250 tests, or roughly a 96% saving. Simulation of the 5950 tests would have taken more than 4 days of single-machine simulation time. With the novelty detection, this time is reduced to less than 6 hours.

3.5.8 Two Additional Results

To show that the novelty detection approach can work well on tests based on a focused instruction base, we conducted an experiment using a test template based on only 6 CFX instructions. 2000 test programs were instantiated each with 50 instructions and an initial state. Figure 3.15 shows the results with and without novelty detection. Without the novelty detection, the original simulation achieves the maximal coverage with 1720 tests. With the novelty detection, the same coverage is achieved with only 100 tests, i.e. a 94% saving.

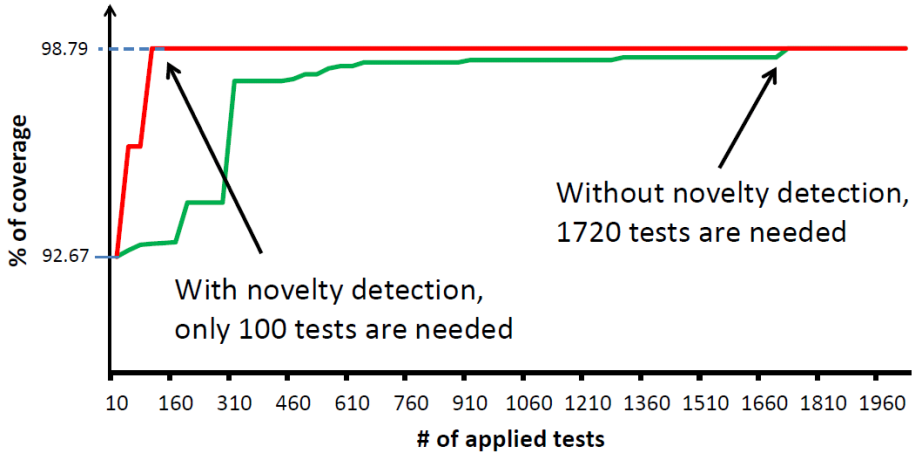

FIGURE 3.15: Results based on 2000 tests instantiated from 6 CFX instructions

To show that the novelty detection can also work well on selected coverage points, we conducted an experiment by focusing on the 200 hard-to-cover points in the CFX unit. 2000 tests of 50 instructions were simulated in the original run. Without the novelty detection, the original simulation achieves the maximal coverage with 1930 tests. With the novelty detection, the same coverage is achieved with only 100 tests, i.e. a roughly 95% saving.

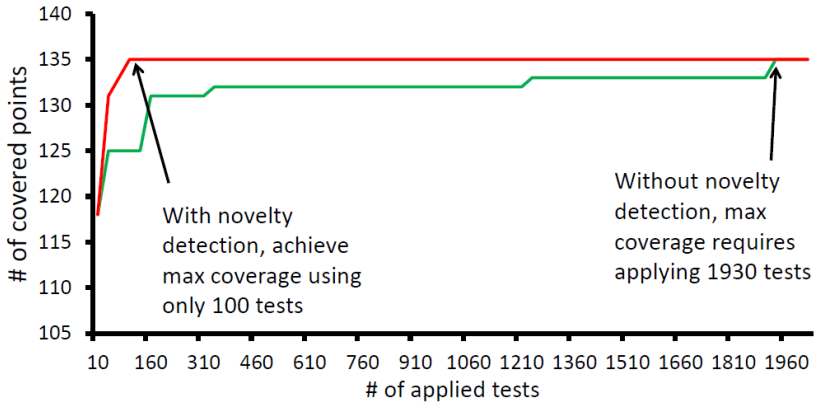

FIGURE 3.16: Results based on 200 hard-to-cover points in CFX

3.6 Limitation of the Single-Instruction Database

Section 3.5.3 discusses the method to estimate coverage for an un-simulated test program and points out its major limitation is in the use of the union operation to compute the coverage (also see Figure 3.9 for this union operation). Because the estimated coverage of a test program is the union of individual estimated coverages of all the instructions in the test program, such an estimated coverage does not consider coverage contributed by multiple instructions collectively. This limits the application of novelty detection to, for example, the load-store unit consisting of multiple finite-state machines, arrays and register files. For example, a data-forwarding event occurs when Read-After-Write hazards are present. Using the single-instruction database would be unable to properly estimate the coverage given by a test containing such hazards.

The LSU is one of the most complex units in the design. It is responsible for scheduling and managing the out-of-order memory operations. To illustrate the idea for overcoming the limitation we focus on an experiment based on the data-forwarding module used in the store queue. The result of the original simulation run is shown in the third plot in Figure 3.2. Below we discuss how

to refine the novelty detection implementation to capture those novel tests shown in the plot.

The idea is simple. To overcome the limitation of using the single-instruction database, we build a database with a large number of test program instances each consisting of three instructions. Then, we use the coverage information stored in this 3-instruction database to estimate the coverage of test programs with a longer length. The indexing function in Figure 3.9 needs to be modified. In other words, the estimated coverage of a 10-instruction test becomes the union of coverages of several 3-instruction instances retrieved from the database.

Figure 3.17 shows the result of applying this extension to the particular example. Without novelty detection, the original simulation achieves the maximal coverage with 2590 tests. With novelty detection, the same coverage is achieved with only 100 tests, a 96% saving. Again, the coverage shown on y-axis is normalized based on the coverage achieved in the particular example and hence, it is shown as 100%.

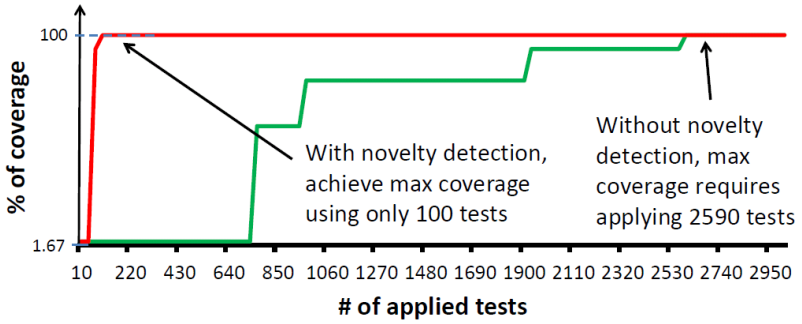

FIGURE 3.17: Comparison of coverage curves with and without novelty detection using extended coverage-based kernel based on the third example plot shown in Figure 3.2

3.7 Summary

In this chapter, we report the experience of applying novel test detection in a company in-house constrained random test generation and simulation environment for a Power architecture-compliant processor core. The first implementation is based on the graph-based kernel method. While this implementation can demonstrate 60-90% saving of simulation time, its practical applicability is limited because of the requirement to manually construct the cost table. To overcome this limitation, a second implementation is proposed. This alternative approach is based on a coverage-based kernel method. The effectiveness of this approach is comparable to the graph-based kernel approach. The alternative approach demands minimal user involvement and hence is much more acceptable in practice. With the second implementation, we demonstrate 80-96% simulation cost reduction in various experiments. In one case, more than four days of single-machine simulation time can be reduced to less than six hours.

We discuss an extension based on the second implementation. The extension overcomes the limitation of using the single-instruction database to estimate coverage. A new database of 3-instruction instances is added to capture coverage depending on multiple instructions collectively. The effectiveness of this extension is demonstrated on the data-forwarding module in the LSU with a potential 96% saving in simulation time.

Chapter 4

Knowledge Extraction Framework to Improve Functional Verification Coverage

4.1 Overview

This chapter proposes a methodology of knowledge extraction from constrained random verification 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 hit the functional events that otherwise are difficult to hit by extensive simulation.

4.2 Introduction

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 "novel" tests (as described in Chapter 3) identified so far. For example, a novel test can be the one hitting a particular block and/or event of interest or capturing a bug in the previous design versions. These two assets embed the knowledge accumulated through the iterative verification process.

In this chapter, we propose a novel learning methodology for extracting knowledge from novel tests. The extracted knowledge then is reused for two purposes: (1) for producing more tests similar to those novel ones and (2) for producing new novel tests that, for example, can hit blocks and/or events 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.

We applied the proposed methodology to verifying a dual-thread low-power 64-bit Power Architecture-based processor core to be manufactured with a 28nm

technology. Our experiments were conducted parallel to 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 hitting a block and/or functional event that received low coverage before. Moreover, a refined test template can effectively generate tests for hitting an event that was not covered before.

The rest of the chapter is outlined as follows: Section 4.3.1 presents a motivational example for this work. Section 4.3.2 addresses the first aspect, i.e. what knowledge to extract. Section 4.3.3 briefly reviews the related works. A feature-based rule learning methodology is presented in Section 4.4 to address the knowledge representation aspect. Section 4.5 discusses the knowledge extraction aspect using subgroup discovery rule learning. Section 4.6 illustrates how the knowledge can be reused. Section 4.7 discusses the possible adaptations to enhance the methodology. Experimental results are presented in Section 4.8. Section 4.9 summarizes the chapter.

4.3 Motivation and Related Works

4.3.1 The Benefits of Understanding Novel Tests

Figure 3.12 in Chapter 3 shows that the special test causing a coverage jump at about the 1930th test in the original simulation is captured by novel test detection within the first 100 tests. It is interesting to understand why the special test can cause such a coverage jump. We analysed the special test

in Figure 3.12 based on a set of features such as instruction types, operand values and the changes of those values in a program. A property we learned is that there is an exception in the novel test that does not occur in non-novel tests. Also, there are move-to-special-register instructions in the novel test, which don't appear elsewhere. We examined the novel test and found the exception triggers an interrupt routine and the special instructions are part of the interrupt routine. Then we modified the test templates to produce more tests satisfying the property and observed the coverage impact.

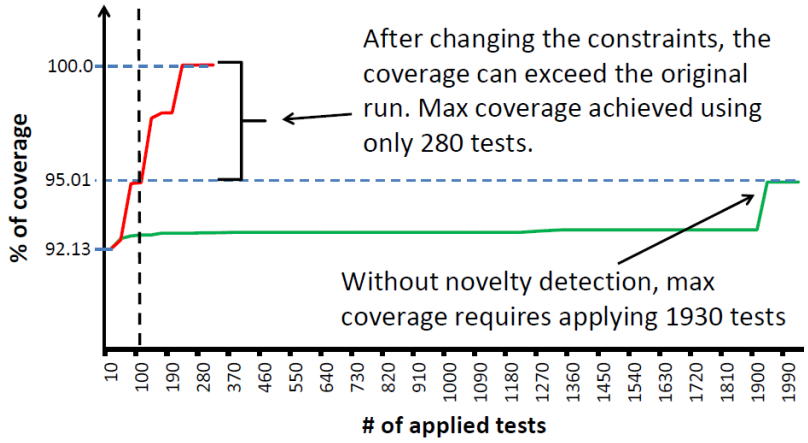

FIGURE 4.1: Improving coverage by test template refinement

Figure 4.1 shows the result of this test template refinement. After simulation of the first 100 tests, the special test that results in a coverage jump is identified. After understanding the unique properties of the special test, the test template is manually modified to produce additional tests. Observe that the additional 180 tests are able to improve the coverage to exceed that achieved by the original 2000 tests. The y-axis is normalized based on the maximal coverage achieved, and that is why the best coverage shown is 100%. Note that this maximal coverage is the best coverage achieved across all experiments on the CFX unit in this dissertation and all coverages for CFX shown in Chapter 3 are normalized based on this best coverage. The result shows us that we can

achieve additional coverage benefits beyond novel test detection by extracting knowledge from the novel tests to refine the test template.

4.3.2 What Knowledge to Extract

The example in Section 4.3.1 illustrates a scenario where structural coverage such as toggle coverage is concerned. In more occasions, functional coverage is more of a concern. During the design iteration, 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 is often used as the metric to evaluate the importance of tests and to guide test template refinement. The definitions of the functional events are relatively stable and do not change as often as the design implementation. Hence, the majority of the purpose for knowledge extraction in this chapter is to improve functional coverage, although as we will demonstrate by experimental results, it can also improve structural coverage.

Figure 4.2 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 functional events 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 events were already hit by 50 or more tests. However, there existed other events activated only by 10 tests or fewer. Furthermore, there were events with zero coverage (not shown in the figure).

Our interest is in knowledge extraction for hitting those events with low or zero coverage. The property stated by a complex event comprises multiple

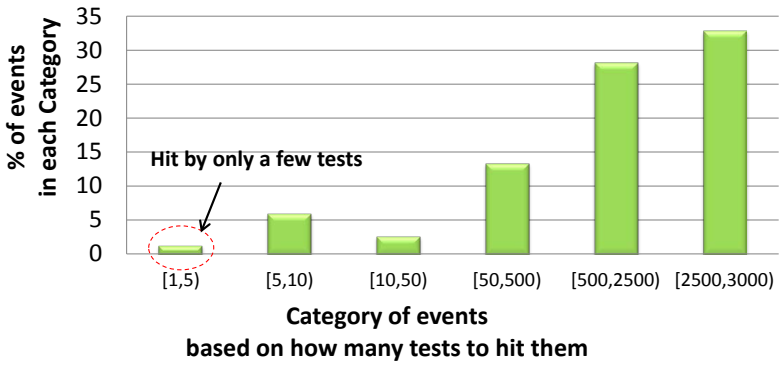

FIGURE 4.2: Histogram of covered events in LSU based on the frequency of being hit

conditions. Learning the knowledge about the entire event 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 hitting the event. The similar thinking applies to improving the structural coverage of a block since hitting a block usually depends on the activation of certain conditions.

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 4.3 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 mis-predicted branch in the test. Such rules are then used as constraints to refine test templates for hitting the condition.

To summarize, in our methodology we begin by monitoring a set of conditions. Novel tests with respect to these conditions are identified and recorded in the

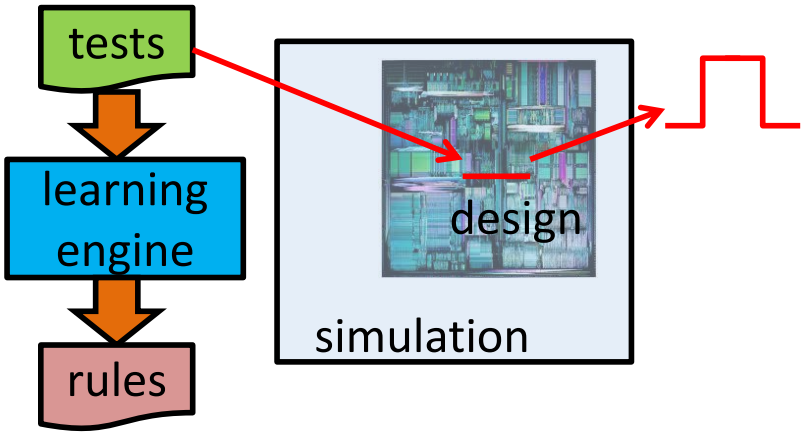

FIGURE 4.3: Illustration of the learning goal

simulation. The extracted knowledge is rules describing the special properties of the novel tests.

4.3.3 Related Works

In a feature-based diagnosis approach, a set of *features* are used to encode the characteristics of a sample (in our case a sample is a test). This encoding transforms each sample into a feature vector. Then, by analysing the feature vector of a special sample against other non-special samples, we can extract rules to explain the unique property of the special sample, e.g. the special sample satisfies the rule and all other samples do not. For the rule extraction analysis, one can use a decision tree algorithm [32] or the subgroup discovery algorithm [33]. Feature-based rule learning has been applied in the context of understanding design-silicon mismatch [34]. In our work, we apply the approach to analyse the special test to understand its specialty. In contrast, this chapter studies the feasibility and effectiveness of applying feature-based analysis for extracting knowledge from novel tests to improve verification coverage.

4.4 Feature Generation

4.4.1 Snippet-based Vector Representation

To extract knowledge from an assembly program, we first need an approach to 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 4.4 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.

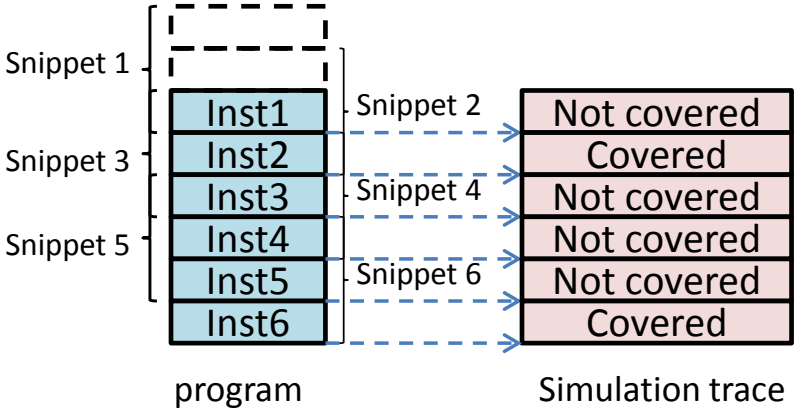

FIGURE 4.4: Illustration of the slide window approach

Each snippet is paired with the episode of simulation trace starting from the commitment of the second-to-last instruction 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 4.4 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 4.5: Illustration of the transformed dataset

Figure 4.5 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 hit and the negative class without the condition being hit.

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.

4.4.2 Defining a Set of Features at ISA level

Features are ISA dependent. A feature set defined for PowerPC ISA can be different from that for x86 architecture. In the proposed methodology, defining a feature set is treated as a one-time cost. During the verification process, the feature set may be manually expanded. However, because the feature set only depends on the ISA, it can be reused for generations of design compatible to the ISA.

In this chapter, the feature set is defined based on the Power ISA [35]. 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 4.6: Illustration of a test program snippet

Figure 4.6 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 4.1. 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 4.1: Illustration of portion of a feature vector

4.4.3 Feature Discretization

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 [36], 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.

4.5 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 of 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 4.1 is shown as follows:

$$\begin{aligned} op\_type_1 = stdx \quad &\wedge \quad op\_type_3 = ldx \quad \wedge \quad collided_3 = 1 \quad \wedge \\ collision\_dist_3 = [1,2) \quad &\Rightarrow \quad S_{covered} \end{aligned} \tag{4.1}$$

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 analysed 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 analysed 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.

We implement a rule search engine similar to the CN2-SD algorithm proposed by [33], which adapted the classification rule learning CN2 algorithm [37] 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 [38] as described below.

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

$$WRAcc(Ante \Rightarrow S_{covered}) = p(Ante) \cdot (p(S_{covered}|Ante) - p(S_{covered})) \quad (4.2)$$

$p(Ante)$ 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 (4.2) 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 [33].

4.6 Knowledge Reuse

4.6.1 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 (4.1) can be encoded into a macro illustrated in Figure 4.7, 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 4.7: 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.

4.6.2 Rule Reuse

Rules and macros are reused to improve the coverage of complex events. A database is built to store the rules and macros for each condition to be monitored. When we want to produce tests to hit an event comprising multiple conditions, the corresponding macros for the conditions are retrieved from the

database. These macros are combined to create more complex macros for hitting the event.

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 4.7 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.

4.7 Learning with Microarchitecture Features

4.7.1 Limitations of Learning at ISA Level

In the learning methodology proposed earlier in this chapter, learning is performed with instruction-level features based on a slide window approach. This might be a limitation when complexity increases. Let us consider an example scenario as illustrated in Figure 4.8. Suppose the 100th instruction of a program invalidates the Translation Lookaside Buffer (TLB). Then following 99 non-memory instructions, the 200th instruction is a load instruction. This

would cause a TLB fault. Apparently, the approach proposed earlier is not capable of learning the rules for the TLB fault event here. This example shows there exist a category of events that cannot be learned efficiently by merely looking at the instruction interactions locally without consideration of micro-architecture states. Hence, when the instruction-level learning proves to be ineffective, adaptations to the proposed learning methodology are needed to deal with more complexities.

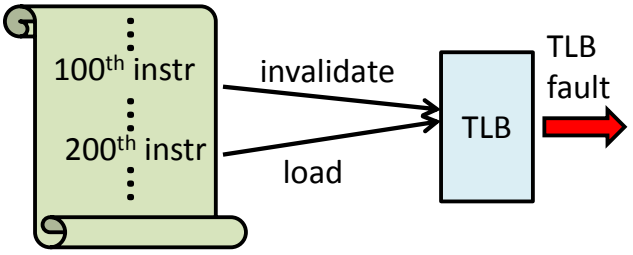

FIGURE 4.8: Illustration of an example scenario in which rules cannot be efficiently learned by the approach discussed in previous sections

There are two key components in a learning methodology: feature representation and learning algorithms. When applying learning to a more complex system, one might wonder which direction to pursue: more sophisticated learning algorithms or better feature representations? At first glance, it might appear that more advanced learning algorithms are required to solve more complex problems. However, our experiences show that a better feature representation will give us more insightful perspectives and thus result in better learning results than going after advanced algorithms. From the analysis of the limitation of learning only at the ISA level, we adapt the learning framework to being applied at two levels. We first learn rules of the important combinations of microarchitecture states present when a special condition occurs. Then a learning scheme similar to that proposed in previous sections in this chapter is applied at the ISA level. The adaptation includes two levels of features: features describing the microarchitecture states, and features describing the

characteristics of instructions. The application of the learning scheme relies on the proper selection of microarchitecture states, hence simple learning algorithms are preferred to search the rules based on different sets of features. A hypothesis pruning and ranking scheme is a good candidate for its simplicity and efficiency. We will briefly review the hypothesis pruning and ranking scheme in Section 4.7.2 and present the adapted learning scheme in 4.7.3.

4.7.2 Hypothesis Pruning and Ranking

All rule learning algorithms essentially try to find good hypotheses that can explain why samples fall into a particular category. A hypothesis space is formed based on features. Given a set of n occurrence features $F = \{f_1, ..., f_n\}$, the hypothesis space is the power set 2^F. Each hypothesis is a combination of features. The hypothesis space formed from all features can be huge. However, when forming hypotheses to explain a special sample, we only need to consider the features that appear in the sample. This number is usually much smaller than the number of all features. However, even for the number of features in the order of tens, the space is still too large to enumerate explicitly. Hence, the hypothesis space usually exists implicitly in the analysis. In the hypothesis space, a lot of hypotheses appear in non-special samples, which means they cannot be used to distinguish the special samples from non-special ones. We call them *inconsistent* hypotheses. When inconsistent hypotheses are pruned, only *consistent* hypotheses that are more creditable for explaining the special samples will be left.

We illustrate the concept of hypothesis pruning through an example data set as shown in Table 4.2. It consists of 2 special samples S_1, S_5 and 4 other

non-special samples. Each sample is characterized by 5 features F_1 through F_5. The hypothesis space formed by features appearing in the special samples is illustrated as a *concept lattice* in Figure 4.9. Each node represents a hypothesis annotated with the number of appearances in non-special samples. For example, the 2-order hypothesis $\{F_1, F_4\}$ denotes the combination of F_1 and F_4, and annotated with 2 since it appears in S_2 and S_4.

TABLE 4.2: Example Data Set

	F_1	F_2	F_3	F_4	F_5	Class
$\mathbf{S_1}$	**1**	**0**	**1**	**1**	**1**	**1**
S_2	1	0	1	1	0	0
S_3	0	1	1	1	1	0
S_4	1	1	0	1	0	0
$\mathbf{S_5}$	**1**	**0**	**1**	**0**	**1**	**1**
S_6	1	0	0	0	1	0

In the example concept lattice, $\{F_1, F_3, F_5\}$, $\{F_1, F_4, F_5\}$ and $\{F_1, F_3, F_4, F_5\}$ are annotated with 0, which means they never appear in non-special samples. In other words, they are consistent hypotheses. The goal of hypothesis pruning is to find all the consistent hypotheses that are not descendants of other hypotheses. In this example, $\{F_1, F_3, F_5\}$, $\{F_1, F_4, F_5\}$ are the results of hypothesis pruning. The reason for finding low-order consistent hypotheses is that high-order consistent hypotheses ($\{F_1, F_3, F_4, F_5\}$ in this example) are included in their ancestors and tend to "overfit" the data set. Also, the low-order hypotheses are easier to understand.

The problem of hypothesis pruning can be formulated as finding a *cut* in the concept lattice. The dashed curve in Figure 4.9 is such a cut separating the consistent hypotheses from the others. Various algorithms can be used to search for the cut [39]. In practice, the pruning is not that strict and can tolerate certain degrees of inconsistency. The consistency checking is replaced by support-confidence evaluation. Let P be the set of all special samples, and

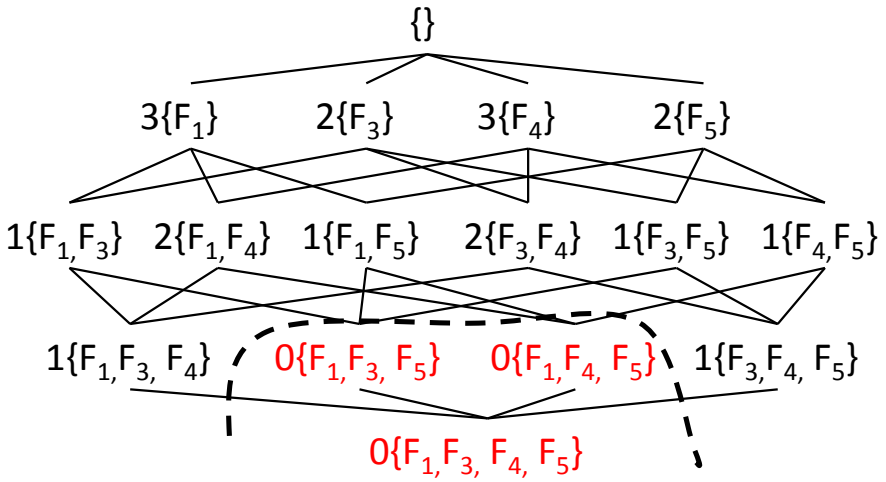

FIGURE 4.9: Illustration of the concept lattice based on the example data set

N be the set of all non-special samples. For a hypothesis h, let P_h be the subset of P satisfying the hypothesis h and N_h be the subset of N satisfying h. Then the support of h is $\frac{|P_h|}{|P|}$ and the confidence of h is $\frac{|P_h|}{|P_h|+|N_h|}$. For example, the hypothesis $\{F_1, F_4, F_5\}$ appears in one special sample S_1 but not in the other special sample. Then its support is 50%. Its confidence is 100% as it never appears in non-special samples.

Rule learning algorithms based on hypothesis pruning usually start with the null hypothesis and generate lowest-order hypotheses first in a breadth-first-search manner. For each hypothesis, support-confidence evaluation is performed with user-specified thresholds. Hypotheses without sufficient support or confidence will be pruned. If a hypothesis passes the evaluation, its descendants will not be explored. If a hypothesis is pruned due to insufficient support, its descendants will not be explored as well (since they have lower support than their ancestors). The search will stop after all lowest-order hypotheses are found or the search depth exceeds a threshold. The hypotheses found are ranked based on support and/or confidence. In the example, $\{F_1, F_3, F_5\}$ is ranked as the top hypothesis as it has higher support (100%) than $\{F_1, F_4, F_5\}$ while they have the same confidence.

4.7.3 Adaptation of the Learning Methodology

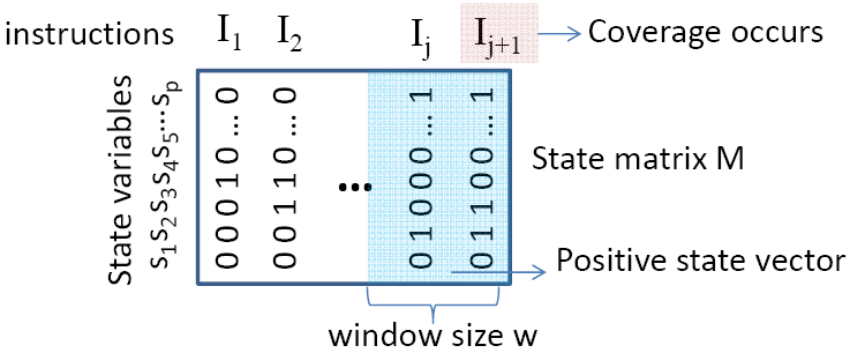

FIGURE 4.10: State matrix view of a test

Figure 4.10 illustrates the learning scheme. We assume that there is a set of relevant microarchitecture states. For each analysis, a subset of p states are selected, $S = \{s_1, \ldots, s_p\}$. Given an instruction sequence, simulation provides the state vector based on S achieved at the end of each instruction. For simplicity, we assume that in this state vector, "1" means the state is present and "0" means the state is not present.

Suppose for a test we observe that instruction I_{j+1} hits a coverage point of interest. Our first learning goal is to uncover what state configuration is causing the instruction to hit the coverage point.

The learning can proceed by assuming a small window size w and concatenating the last w columns in the state matrix in Figure 4.10 to form a *positive state vector*. Then, in all tests where an instruction of the same type as I_{j+1} appears and the coverage point is not hit, we extract a *negative state vector*. There can be many negative state vectors.

FIGURE 4.11: A positive state vector and its hypotheses

Given a positive state vector, a set of hypotheses can be formed. For example, Figure 4.11 shows a positive state vector of 10 states. Since only states 3, 4, 6 and 8 are present, hypotheses are formed based on these four states. Others are ignored because we assume that hitting the coverage point depends only on state presence. Using four states, there are $2^4 - 1 = 15$ hypotheses. Each hypothesis can then be checked against all the negative state vectors. For example, one can set the rule that if a hypothesis appears in any negative state vector, it will be removed. Or one can set the rule that if a hypothesis appears in much more positive state vectors than negative state vectors (when combining analysis of multiple tests), then it can be accepted.

The learning scheme is quite simple. Its power of course lies in the proper selection of S. Also because the learning algorithm is simple, its effectiveness decreases as the number of states p and the window size w increases, i.e. the dimensionality of a state vector increases. In practical use, we want to keep p and w small. For example, w can be kept at 2 while p is kept at less than 10.

Because p is kept small, it may require multiple runs to try out different sets of state variables. In practice a user makes the selection of the set S from a larger set of known state variables built into the tool. Even so the simple learning scheme still provides a useful way to help the user quickly explore a large number of hypotheses.

Suppose a hypothesis is accepted. Then, finding the instructions causing the states is a relatively easy problem. One can trace back in time from the state vector to identify the corresponding instruction where a state is first present. Then, we apply a second level of learning to learn the characteristics of the instruction for causing the state presence. In this second level of learning, a positive example is an instruction causing the state of interest. A negative

example is an instruction of the same type that does not cause the state to be present. The learning scheme can be designed in a similar fashion as described above.

In summary, the two levels of learning schemes first try to uncover the combination of states causing the coverage hit and then uncover the specific instructions and instruction features.

4.8 Experiment Results

4.8.1 Experiment Environment

In this chapter, the experiments were conducted on a dual-thread low-power 64-bit Power Architecture-based processor core. The design is a newer version of the processor core mentioned in Chapter 3. 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. In this version, 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. The RTL simulation results are compared with the architectural simulation results for checking correctness.

The experimental results shown below focus on the Load Store Unit (LSU). LSU is one of the most complex and difficult-to-verify units in the design. The LSU in this version supports up to eight outstanding load operations and eight outstanding store operations in-flight. The design leverages features such as store queueing, L1 load miss queueing, store gathering, and critical-word-first service to support the speculative memory access in order to achieve high performance.

Since the experiment was conducted in parallel to the on-going verification efforts, the experiment started with test templates that had been refined by the verification team up to the time of the experiment. In the following, we describe five results in detail to demonstrate the effectiveness of the proposed learning methodology. The first result is based on toggle coverage and the rest are focused on the more important coverage type– functional coverage.

4.8.2 The First Illustrative Result Based on Structural Coverage

The first experiment was conducted on a test set instantiated by a test template based on 6 load/store instructions. It contains 3000 test programs, each of which has 10 instructions. Figure 4.12 shows the accumulated toggle coverage curve of the data forwarding block in LSU. Out of 3000 tests, only 4 contribute to coverage increase for that block. After simulating the whole test set, the coverage only reaches 46.31%.

Instruction-level rule learning were conducted to extract rules. One rule we learned can be interpreted as follows:

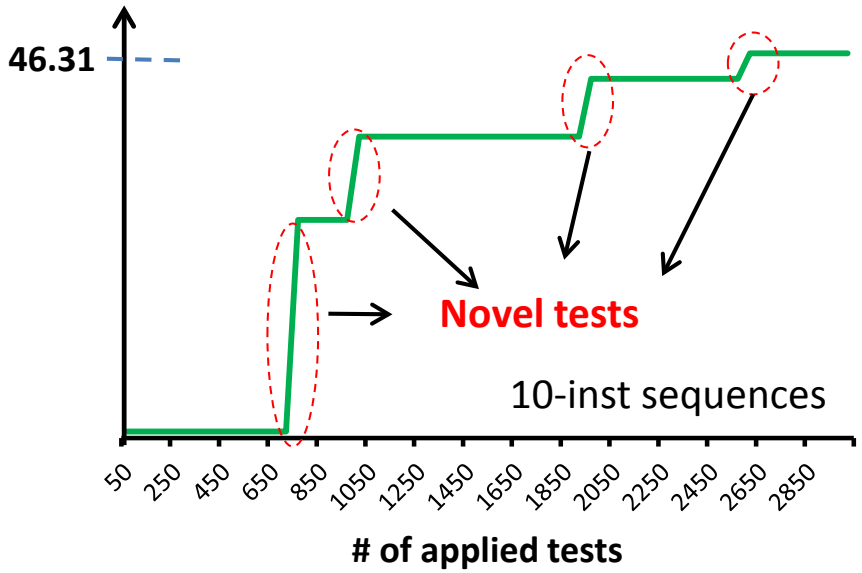

FIGURE 4.12: Toggle coverage on a block in LSU of the original simulation run

- There is a store instruction followed by a load instruction.
- The data width of the two instructions are the same.
- The real addresses of the two instructions can only differ in the last 3 bits.
- There are no more than 2 instructions between these two instructions.

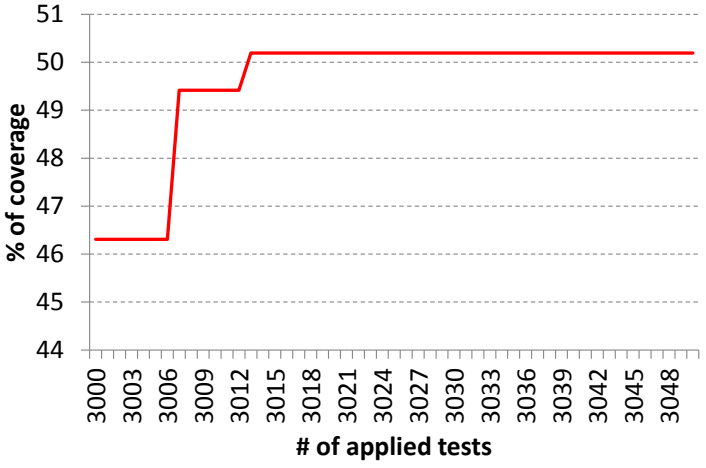

FIGURE 4.13: Coverage improvement in the first iteration

We modified the test template toward producing tests featuring sequences that satisfy this rule and generated additional 50 tests. After running the 50 tests, coverage was increased from 46.31% to 50.22%, as shown in Figure 4.13. This was not the endpoint. Since we had more samples of novel tests, we could improve the effectiveness of learning by extracting rules based on the newly generated tests. In this iteration, we discovered that if the qualified sequences occur more than once in a test, the test is more likely to contribute to the coverage increase in the block.

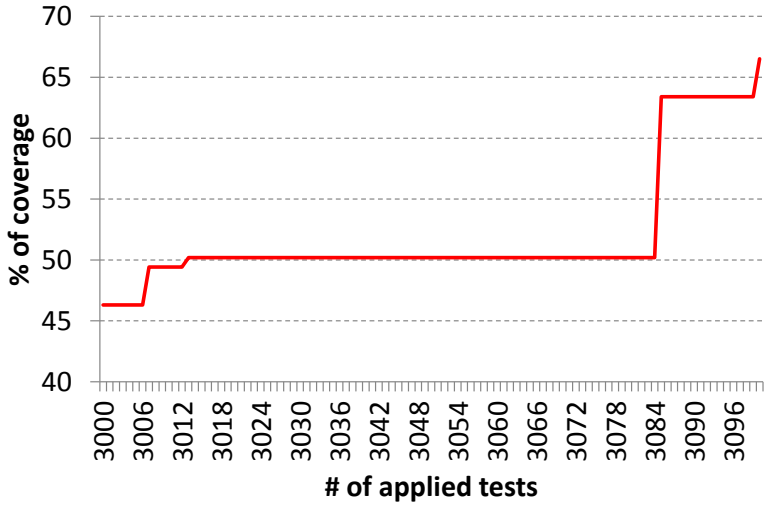

FIGURE 4.14: Coverage improvement in the second iteration

Then we applied this new rule to direct the test generation and instantiated another 50 tests. In this iteration, the coverage was increased to 66%, as shown in Figure 4.14. Again, we iterated the learning process based on the refined tests. We discovered it can be interpreted as that if the Hamming distance between the data of interest in two sequences is large, it's more likely to contribute to the coverage increase of the block. Thus, we applied this constraint and generated another 5 tests. As we can see from Figure 4.15, after applying the extra 5 tests, the coverage of the block ramped up to 100%.

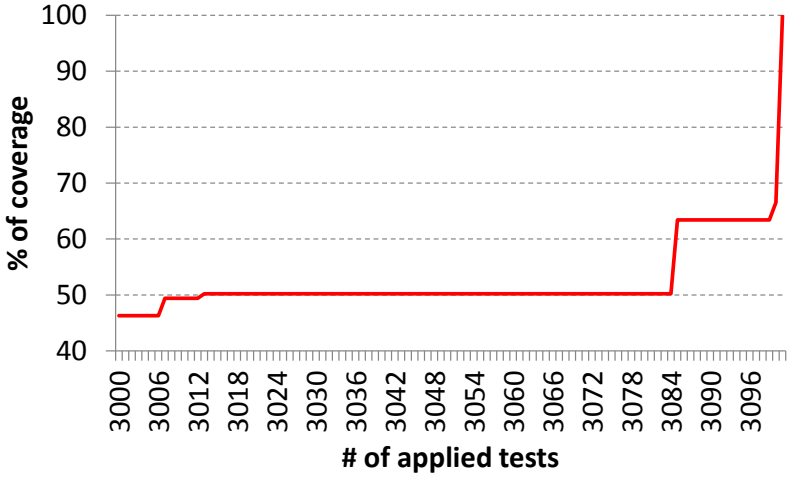

FIGURE 4.15: Coverage improvement in the last iteration

What would we get if we continued simulation using tests generated by the original test template? The comparison is shown in Figure 4.16. We instantiated another 4000 tests and simulated them, which took over 15 hours. However, the coverage only increased to 50% and levelled off in the long simulation. It's very unlikely to increase coverage if we continue simulation without learning.

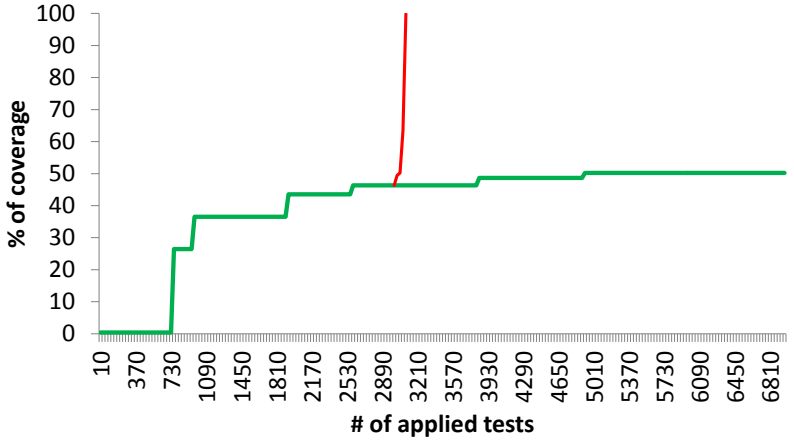

FIGURE 4.16: Comparision between coverage w/ and w/o learning

4.8.3 The Second Result

The second result demonstrates the following: The learning began with novel tests hitting an event α 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 hitting event α was substantially improved. Moreover, two additional events β, γ (with zero coverage before) were covered.

The event β comprises a single condition c_2 that is highly correlated to the condition c_1. The event γ comprises both conditions c_1 and c_2. The result demonstrates that learning from tests hitting one event can lead to fortuitous coverage of other correlated events.

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 event A was covered by merely three tests. This event 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 4.3 shows the comparison of coverage between the original 1000 tests and 200 new tests. As shown by the 4th row, the number of tests hitting the event α increases from 3 to 33 in the new test set, thus boosting the frequency from 0.3% to 15.5%. Moreover,

events β and γ which were not hit by the original 1000 tests could be hit 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
event α	3	33	0.3%	15.5%
event β	0	9	0	4.5%
event γ	0	5	0	2.5%

TABLE 4.3: Comparison of event coverage between original 1000 tests and 200 new tests

4.8.4 The Third Result

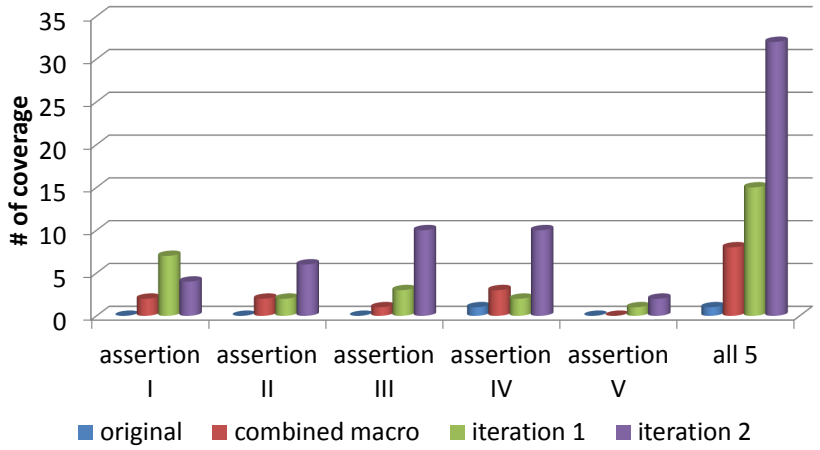

FIGURE 4.17: Functional coverage improvement

Figure 4.17 summarizes the third 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 event IV was hit by one test. event I, II, III, and V had zero coverage.

Event IV comprises two conditions c_3 and c_4. Other events comprise the same two conditions. However, the temporal constraints between the two conditions are different across the five events.

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 hitting event IV. The corresponding rules for these two macros are illustrated in Table 4.4. 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 4.4: 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 events I to IV as shown with the legend "combined macro" in Figure 4.17. But event 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 4.17. Observe in "iteration one" that coverage for event I to IV was improved further. More importantly, event V could be covered. The process repeated in the "iteration 2" and we can observe further coverage improvement for events II to V.

4.8.5 The Fourth and Fifth Results

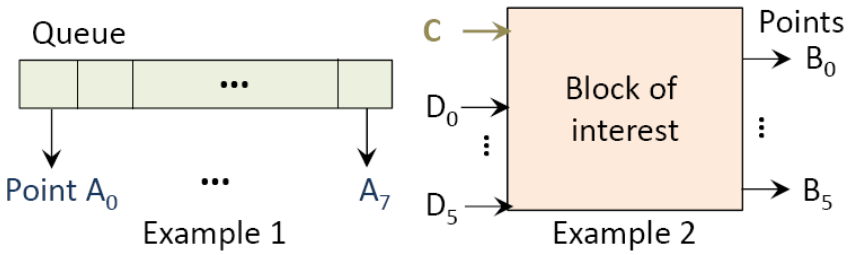

FIGURE 4.18: 2 examples, coverage point sets A and B

Figure 4.18 illustrates the fourth and fifth examples to illustrate the effectiveness of two-level learning. The first group contains 8 coverage points $A_0 \ldots A_7$. The second group contains 6 coverage points $B_0 \ldots B_5$. The second rows of Tables 4.5 and 4.6 show their initial coverage based on the test templates developed by the verification engineers. For group A, 400 tests were simulated, which hit A_0 10 times and A_1 17 times. Others had no coverage. For group B, >30K tests were simulated, which hit none of the points.

The 8 points in group A correspond to filling the 8 slots in a queue. If slot i is filled, A_i is activated. The queue is consumed by other parts of the processor depending on the machine state. While filling a slot may not be difficult, once a slot is filled, it can be consumed quickly and removed. Hence, it is difficult to keep many slots filled simultaneously.

TABLE 4.5: Coverage improvement after learning

Stage	# of tests	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
Initial	400	10	17	0	0	0	0	0	0
based on 400 tests	100	3	11	10	10	4	2	1	1
based on 500 tests	50	72	59	71	83	79	97	96	87

Table 4.5 shows the improvement after applying the learning schemes discussed above. After learning based on the initial 400 tests, we were able to find constraints to improve the test item. The improved test template was used to produce 100 new tests. All points in group A were covered at least once.

The 100 new tests provide additional information to learn from. Hence, the learning schemes were applied to all tests, including the 100 new tests. The test template was further improved accordingly. The last row shows that with 50 tests, the further-improved test template can cover all points many times. The coverage frequencies are greater than 50 because each test consists of a sequence of instructions and by focusing the test template on hitting the coverage points, many instructions can hit them, i.e. a test can cover a point more than once.

Refer to example 2 in Figure 4.18. In this case, none of the points was covered in the initial run. However, by analysing the block of interest, one can easily discover that in order to hit coverage points in group B, setting signal C and signals D_0 to D_5 is necessary. Hence, instead of learning to reach points in B, which is not possible because the tests and simulation data provide no information to learn from, the learning objective is to provide better controllability for C and D_0 to D_5.

TABLE 4.6: Coverage improvement after learning

Stage	# of tests	B_0	B_1	B_2	B_3	B_4	B_5
Initial	>30K	0	0	0	0	0	0
based on >30K tests	1200	1	0	0	16	25	26
based on >30K + 1200 tests	100	2	1	1	56	61	77

After learning based on the initial >30K tests, Table 4.6 shows that B_3 to B_6 could be covered many times. B_0 was covered once. Again, the learning can include the 1200 new tests. Results are shown in the last row of the table.

4.9 Summary

This chapter 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 event coverage. Experimental results demonstrate the effectiveness of the proposed learning methodology in various scenarios where event coverage could be further improved after a substantial verification effort had been spent.

Chapter 5

Data Driven Test Plan Augmentation in Platform Verification

5.1 Overview

Verification of a platform design can be divided into two parts, core verification where an individual core is verified and platform verification where integration of multiple cores is verified. In both parts, constrained random test generation is applied and verification is driven by certain predefined coverage metrics. This chapter focuses on platform verification and shows that a data learning approach designed for core verification is not feasible for platform verification. We explain the differences between the two from the perspective of applying data learning and point out a fundamental problem to be solved in platform verification. We propose a data driven approach for test plan augmentation

and demonstrate its effectiveness using a latest commercial platform design. Experimental result shows that the approach can be used to further improve coverage during a verification cycle.

5.2 Introduction

Functional verification starts with a verification plan, specifying the aspects of design to verify [6]. In constrained random verification, test cases are generated by constrained random test generation, which is guided by constraints and biases specified in a test item (or test template). Verification quality is measured by coverage metrics and coverage results are analysed to guide the setting of constraints and biases.

In a design process, the design evolves over time. This means that functional verification also evolves accordingly. From one design version to another, functional verification has two important goals: to identify bugs in the current version and to develop a collection of test items and/or direct tests that eventually will be used to simulate the final version of the design.

In this chapter, we call a constrained random test generator a *randomizer*. We call a particular setting of constraints and biases a *test item* and a collection of test items a *test plan*. A randomizer processes a test item and produces a set of tests. For example, a test for verifying a processor core is a sequence of assembly instructions. A test for verifying a platform is a sequence of transactions.

For test plan development, it often relies on a coverage metric to drive the improvement of a test plan. The specifications of coverage models are often

encapsulated into *coverage groups* where each group consists of *coverage points*. A coverage point is usually defined as a particular combination of signal values.

Suppose a randomizer generates a set of tests intended for a coverage group. Suppose some or none of the coverage points are covered and others are not. This is a typical scenario where a verification engineer analyses the coverage result and tries to improve the test item so that the randomizer can produce tests covering the uncovered points.

In this context, the data learning methodology proposed in Chapter 4 analyzes the simulation traces and the tests to assist understanding of why some tests hit the coverage points and some do not. The result is then used to add additional constraints for better controllability for hitting the points in the coverage group. The experiment results showed that in practice such a methodology could be useful for processor verification. However, for platform verification, we discover that the data learning methodology is ineffective.

At processor level, the learning is for incrementally improving a test item. Finding a better set of constraints often means adding more constraints such that they can reach a desired microarchitecture state. Which constraints to add can be learned from the tests and their simulation traces. At platform level, we observe that the core of the problem is the incompleteness of a test plan. This means that to apply the same learning idea, one has to be able to learn from one test item to discover another. We observe that in such an application, the tests and simulation traces often provide little or no information to enable the learning.

Figure 5.1 illustrates the difference between learning in the processor verification context and learning in the platform verification context. For processor

verification, the task is to learn from the behavior of a collection of tests to improve a test item. In platform verification, the task is to learn from a collection of test items to improve a test plan. We discovered that in such an application, the tests and their simulation traces often provided little or no information to enable the learning. Therefore, while the learning approach proposed for processor verification is more like a supervised learning approach, we found that for platform verification, we had to develop an unsupervised approach.

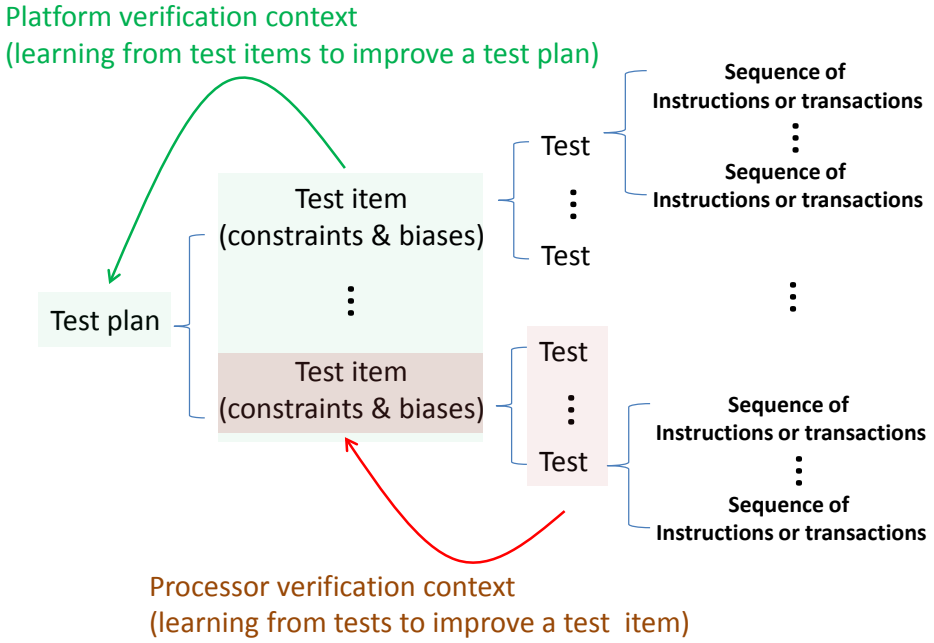

FIGURE 5.1: Processor Verification vs Platform Verification

In this chapter, we propose a different data driven methodology to overcome the test plan incompleteness problem. The data to analyse is the collection of existing test items. The objective is to identify feasible augmentations to the test plan by adding new test items. The approach analyses only the test items without using the actual tests and their simulation results. We apply the methodology to a latest commercial platform design and demonstrate its effectiveness of use during a verification cycle.

The rest of the chapter is outlined as follows: Section 5.3 introduces platform verification and explains why it is difficult to learn from tests and simulation traces to improve coverage. Section 5.4 formulates the data learning problem present at the platform level. Section 5.5 discusses the algorithms to tackle the problem. Section 5.6 shows results based on the commercial platform design. Section 5.7 concludes the paper.

5.3 Platform Verification

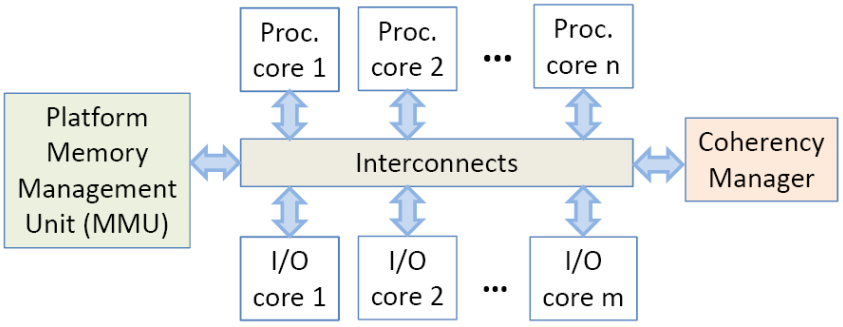

FIGURE 5.2: Illustration of a platform

Figure 5.2 illustrates a platform architecture. A platform may comprise a number of processor cores, I/O cores, a platform MMU including cache, which are interconnected through some interconnect structure. A coherency manager is responsible for routing the transactions between cores and for ensuring data coherency in the system.

Verification can be divided into two parts, individual core verification and platform verification. For example, if the processor core is designed in house, verification of an individual processor core is required before platform verification. In platform verification, the focus is on the interconnection of cores. Therefore, coverage groups often are defined on the interconnect interfaces.

As mentioned in the discussion referring to Figure 5.2 before, by assuming each individual core is verified, platform verification focuses on verifying the interconnection. Figure 5.3 illustrates the idea.

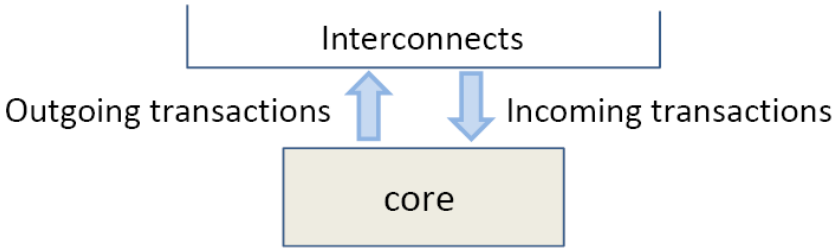

FIGURE 5.3: Transaction view in platform verification

In platform verification, most of the coverage groups are defined based on interface signals between the interconnects and a core. For example, a group may comprise a particular transaction type with combinations of attributes. From a core perspective there are two types of transactions, outgoing and incoming. Usually coverage points based on outgoing transactions are easier to hit because hitting those points only depends on supplying the required inputs to the core. Verification engineers usually have a good grasp of the behavior of the core and hence, it is relatively easy to write a test item to produce those outgoing transactions.

Incoming transactions depend on the rest of the system. If an incoming transaction represents a response to an outgoing transaction, then whether the desired response can be observed or not depends on the system states and behavior of the rest of the system. It is much harder to prepare a test item targeting on a particular response.

If one considers all possible system configurations and state variables combined from all cores, the state space is enormous. Typically, a randomizer takes a test item and instantiates a test with a few hundred transactions. Thousands of tests can be generated based on the configuration and initial state setting

specified in the test item. From the system state space perspective, because the length of each test is short, a test usually does not go far from its initial state. This is illustrated in Figure 5.4 below.

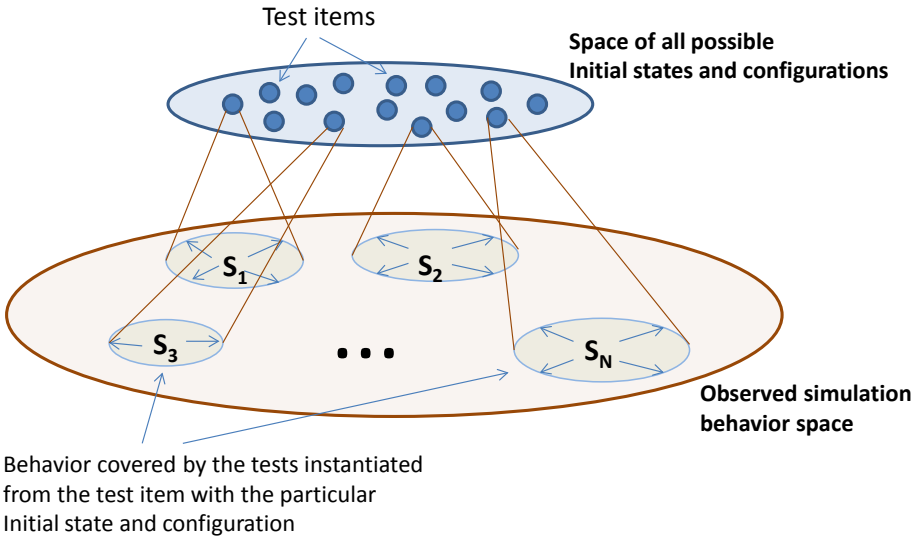

FIGURE 5.4: Illustration of platform verification

In Figure 5.4, each S_i denotes an initial system state of a test item based on a system configuration. Constraints and biases in the test item influence how the transaction sequences are generated based on the initial state. With a number of tests generated and simulated, the system states close to the initial state may be explored. However, it is unlikely that a transaction sequence would start from one initial state S_i and reaches another state S_j.

In platform verification, each S_i is defined based on a set of predefined architecture state variables. There can be tens of variables for each core and hundreds for the system. Each test item involves one initial state. In platform verification, the challenge lies in the coverage of the initial states.

For platform verification, the core of the problem is to ensure a complete coverage of the initial states and configurations. The figure tries to illustrate that for such a problem, learning from the behavior traces of a collection of

tests based on one test item provides little help to reasoning about another test item. In other words, in Figure 5.4 learning carried out in the lower level space (simulation behavior space) would not be effective to improve the coverage of the upper level space (initial space and configuration space). Because of this, the learning approach proposed for processor verification is not effective for platform verification.

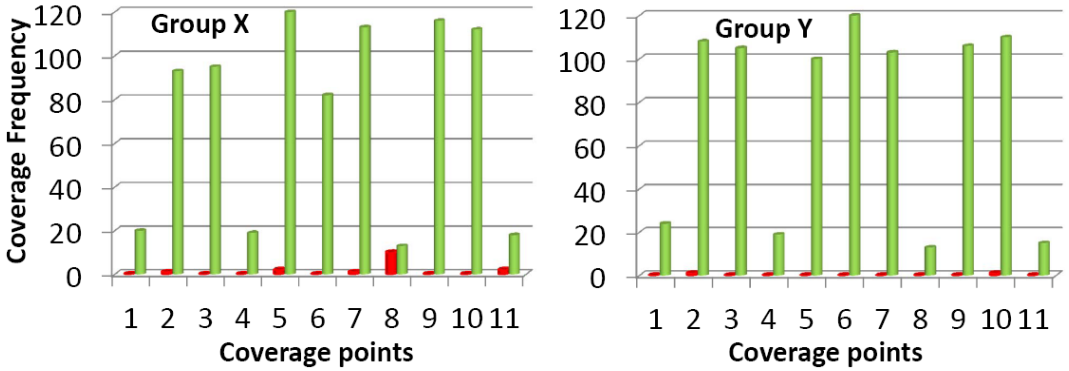

FIGURE 5.5: Platform coverage examples

To illustrate this point, Figure 5.5 shows two experimental results. In each plot, two coverage results are shown. The red bars correspond to the result from an original test item. The green bars correspond to the result from an improved test item based on the original test item. The improvement is based on manual analysis.

Both coverage groups are based on the requirements of seeing a particular response from the system when a particular type of transaction is entered into the system from the core. The difference between group X and group Y is that they are based on different types of responses. Within each group, the differences between two coverage points are based on their data sizes and settings of some attributes of the transactions.

After analysing the results from the original test item, we discovered that in order to cover the points, we had to set the cache in the platform MMU (refer

to Figure 5.2) into specific states, different for different groups. The initial state setting in the original test item does not specify those cache-related state variables. As we observe in the left plot for coverage point 8 of group X, there is still a chance that the random transaction sequences can reach the specific cache state and consequently hit the point. However, in most of other cases, the coverage is zero.

Figure 5.5 illustrates the same concept shown in Figure 5.4. Given a test item with an initial state S_i, random transaction sequences are unlikely to bring the system to another state S_j. Suppose we start with the test item and try to hit the missing coverage points that depend on state S_j. In this case, tests and simulation results from the test item provide no information to learn about the importance of S_j. We see that in this case, the data learning approach previously used for processor core verification cannot be applied.

5.4 Test Plan Augmentation Problem

At platform level, a missing coverage point is usually covered by finding the proper initial state setting. It is rather difficult to modify only the constraints of a test item to bring the system from one initial state to a desired system state. Therefore, the main challenge in platform verification lies in the selection of initial states.

Because the data learning schemes proposed for processor core verification do not apply here, we formulate a different learning problem. The new learning problem is unsupervised, meaning that the learning is not based on the results of a test item, i.e., tests and simulation traces. Instead, the learning is based on

analysing a collection of test items themselves and trying to uncover additional test items. Because the most relevant portion of a test item is its system configuration and initial state setting, the learning focuses on analysing those settings.

Simulation environments and the randomizer define a set of architecture state variables that can be used to set the system configuration and the initial state. Let this set be $S = \{S_1, \ldots, S_n\}$. Due to system architecture constraints and constraints from the simulation models of the cores, the state variables can be partitioned into groups $G = \{G_1, \ldots, G_m\}$. Each group comprises a few or tens of variables.

Based on the variables in a group, *choices* are defined as combinations of some or all variables. For example, each group G_i contains c_i choices. When defining a configuration with an initial state, certain groups can be combined together. For example, given G_1, G_2, G_3 with numbers of choices c_1, c_2, c_3, an initial state can be set with any one of the $c_1 \times c_2 \times c_3$ choice combinations.

It is difficult for a single engineer to fully grasp all system architecture constraints. Furthermore, often a simulation model is given as a behavior model that is not developed by the verification engineer - it is also difficult for the engineer to fully grasp the constraints imposed by a simulation model. Because of these two reasons, the partitioning boundaries of groups, the possible choices within a group, and the possible combinations of multiple groups are usually not entirely clear to a person. When preparing test items, an engineer relies on partial knowledge to define what he/she understands, a possible configuration and a feasible initial state.

The test plan augmentation problem can be stated as follows. Given a test plan consisting of N test items, each with a set of some variables in S, uncover groups and choices in each group. Further select groups to produce a permutation of choices across the groups. Each combination in the permutation represents an initial state setting for a test item.

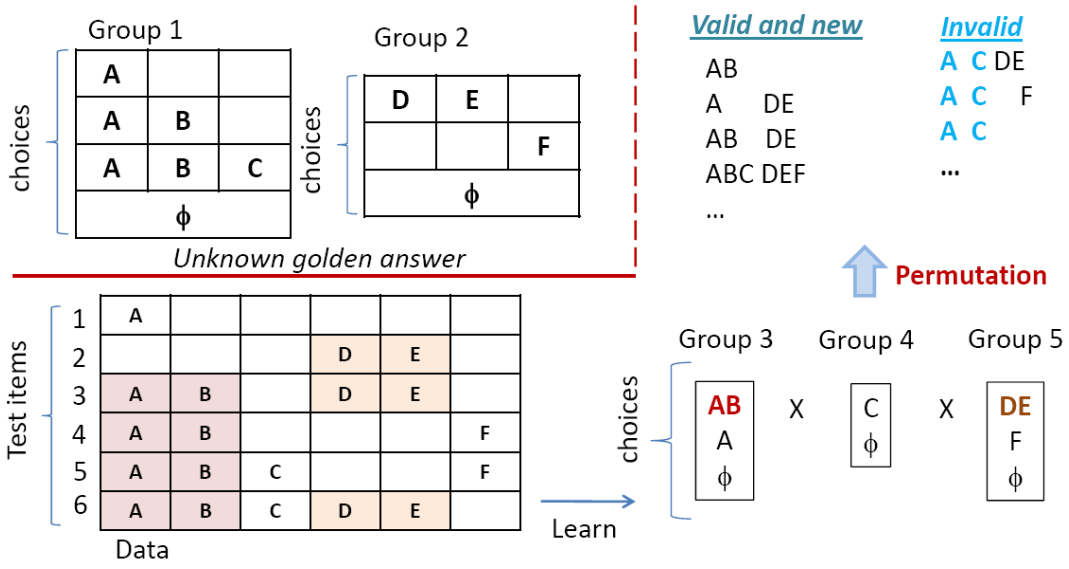

FIGURE 5.6: Illustration of the learning problem

The problem is illustrated in the simple example depicted in Figure 5.6. Suppose the golden answer contains two groups, one with four choices and the other with three choices including the "null choice" option ϕ as shown. For example, Group 1 involves three variables A,B,C with four choices A, AB ABC and ϕ. Because an engineer only has partial knowledge of this grouping and choices, six test items are prepared. Their initial state settings are A, DE, ABDE, ABF, ABCF, and ABCDE (shown in the table below the "unknown golden answer"). This table becomes the data to learn from.

From the six items in the table, observe that we can extract frequent patterns. For example, AB appears most frequently by discounting the frequency of a single variable appearance. Hence, we make AB a choice and consequently

variables A and B are put into the same group. This group then contains choices A (1st row), ϕ (2nd row), and AB (3-6 rows). This decision leaves four variables ungrouped. We then apply the process iteratively. The next most frequent pattern is DE. Then, variables D and E are put into the same group. The group contains the choice DE based on the data. This leaves variables C and F ungrouped.

Variable F can be grouped with variables D and E because F and DE appear disjointly in the data. The result contains 3 groups, denoted as groups 3, 4 and 5.

Based on the resulting groups, our goal is to permute all combinations of choices to augment the original test plan containing six test items. This leads to $3 \times 2 \times 3 = 18$ combinations. Note that the grouping guarantees that the 18 combinations include the original six. However, notice that the original groups 1 and 2 give only 12 combinations. This means that there are 6 combinations "invalid" from the perspective of the golden answer. To remedy this issue, we need to find a way to merge groups. Reducing the number of groups decreases the chance of generating an invalid combination.

The simple example shows several characteristics of the learning problem: (1) The golden answer is unknown. (2) Data contains partial information about the golden answer. (3) The learning result may contain more groups than the golden answer due to incomplete information. (4) The resulting permutation gives a set of test items that is a superset of the original test items. (5) The resulting permutation may give an invalid test item.

In summary, the learning problem is to find groups and choices from the data

such that the resulting permutation augments the original test plan and minimizes the number of invalid test items (initial state settings). In the simple example, the invalid state settings can be removed by merging group 4 into group 3. This motivates us to develop a two-stage algorithm that first produces the groups and identifies the choices, and second selects groups to merge in order to reduce the number of invalid settings.

5.5 Platform Learning Algorithm

5.5.1 Test Item Clustering

The example in Section 5.4 assumes only one "unknown golden answer". In practice, a collection of test items might reflect several different "unknown golden answers". Hence, before searching for groups and choices, we need to partition test items into different test plans and then, within each test plan, decide groups and identify choices.

We formulate the partitioning problem as a clustering problem. Common clustering algorithms include k-means [40], affinity propagation [41], mean shift [42], spectral clustering [43], and hierarchical Clustering [44]. The performance of many clustering algorithms depends on the selection of the number of clusters. Mean shift is a non-parametric clustering technique which does not require prior knowledge of the number of clusters or the shape of the underlying distribution. Mean shift finds clusters by searching for the modes (dense areas) in the density estimation of data samples [42].

To apply mean shift to a set of test items, each item is encoded as a feature vector where the features are all variables in the data and the feature values indicate the appearances of the variables. Given n data samples $\vec{x}_i, i = 1, ..., n$ on a d-dimensional space, a kernel function $K(\vec{x})$ measures the similarity between two samples. Given a window radius h, the density of the data is estimated as:

$$f(\vec{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{\vec{x} - \vec{x}_i}{h}\right) \tag{5.1}$$

The modes, which are the local maxima of the kernel density function, represent the dense regions in the feature space. At these modes, the gradient of the density estimation $\bigtriangledown f(\vec{x}) = 0$. The main idea of the mean shift algorithm is to find the modes for each data point using a gradient ascent search until converging to the point at which $\bigtriangledown f(\vec{x}) = 0$. Then data points associated with the same modes belong to the same clusters. Mean shift might not result in the optimal clustering with one shot. For each cluster, we can apply mean shift iteratively to refine the clustering until it does not yield new clusters.

5.5.2 Group Partitioning & Choice Generation

The next step is to process each cluster individually. In this step, the goal is to decide group boundaries among variables and also to form choices, as illustrated using the simple example in Figure 5.6 above. As described before, the algorithm first iteratively identifies the most frequent patterns and variables in each frequent pattern are put into a separate group. This first step produces an initial set of groups.

For each group, the choices can be formed by collecting all variable combinations appearing in the data based on the variables contained in the group.

The next step intends to merge groups. In this step we apply a simple graph algorithm. Each group is a vertex. There is an edge between two groups if their sets of choices are disjoint where two choices are considered disjoint if the combination of the choices does not appear in any test item. Then, groups belonging to the same strongly connected component are merged. For example, in the example in Figure 5.6, DE and F will be merged into the same group in this step.

Note that in each group, a "null choice" may be added to indicate the option that none of the choices in the group is selected. The addition is based on the data where if there exists a test item that does not involve any choice in the group, then a "null choice" is added.

5.5.3 Further Group Merging

The example in Figure 5.6 shows that even after the first merging there is still possibility that some groups should be merged with others. In the example, the group with choice C should be merged with the group with choice AB. Otherwise, invalid combinations will be produced.

We apply a postprocessing step to further refine the group boundaries. Given two groups G_i and G_j, the idea is to calculate a *gain* by the action of merging. Then, we use these gains to rank all group pairs and the pair with the highest gain will be merged. The process iterates until the highest gain is below a user-defined threshold.

The gain of merging is calculated based on a *bias* estimate. Given a group with choices $c_1, c_2, \ldots, c_k$, let the frequency of appearance of c_i in the data be f_i. The bias is calculated as $bias = \sum(f_i - ave)^2$ where $ave = \frac{\sum f_i}{k}$. Let the merging result of G_i and G_j be denoted as group G. The gain of merging is $gain = \min(bias_{G_i}, bias_{G_j}) - bias_G$.

Conceptually, the bias measures the uniformity of the frequencies of choices across a group appearing in the data. The *gain* heuristic intends to merge groups that would result in a distribution of frequencies closer to the uniform distribution. This is based on the assumption that if a set of choices is meant to be in a group, their usage in the data should appear random and hence, their appearance frequencies should appear close to the uniform distribution.

5.6 Experiment Results

The experiment was conducted with an in-house simulation-based verification environment of a latest commercial SoC platform. Figure 5.7 illustrates a simplified view of the SoC platform. The platform features 3 core complexes, each of which contains 4 dual-thread Power Architecture-based microprocessor cores. Each processor core has its own L1 instruction cache and data cache while each core complex shares a unified L2 cache.

The system fabric is an on-chip coherent interconnect network that conforms to a proprietary bus protocol compatible with Power Architecture. The traffic routing, transaction ordering and coherency maintenance are managed by the coherency manager. The core complexes send requests to the system through Processor Requester Ports (PRP). I/O devices are attached to I/O host bridges

which communicate with the coherency manager through I/O Requester Ports (IRP) and I/O Target Ports (ITP). There is an on-chip L3 platform cache integrated with on-chip memory controllers. The coherency manager routes the memory requests to the platform cache through Memory Target Ports (MTP).

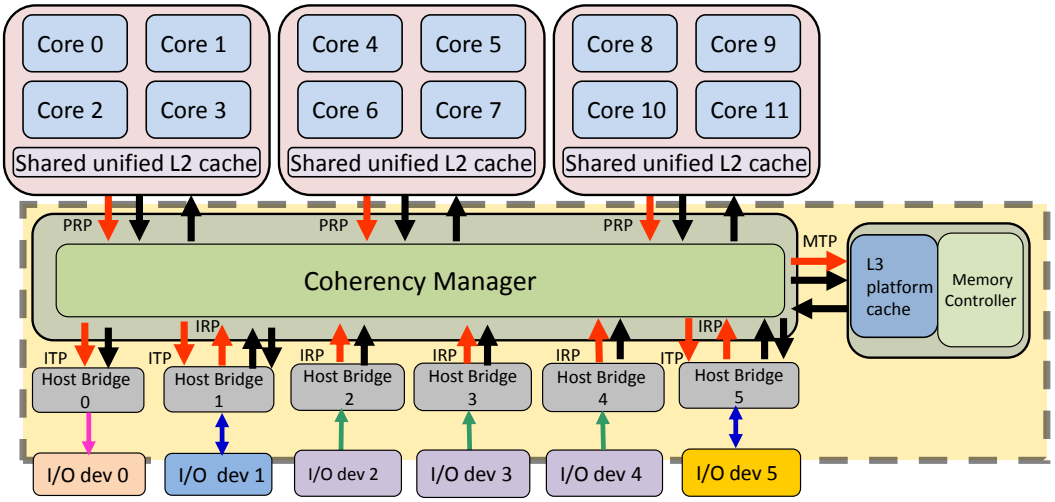

FIGURE 5.7: A simplified illustration of the SoC platform

The verification environment conforms to a state-of-the-art transaction-based verification flow. A random transactor embedded in the constrained random testbench is used to generate transactions based on user-supplied parameters. RTL models of the system logic (the parts encapsulated in the dashed rectangle in Figure 5.7) and Bus Functional Models (BFM) of the cores, I/O devices and external memories are used in simulation. The bus traffic events are monitored and checked for correctness.

The test plan to be analysed contains 572 test items developed manually with 157 state variables. In the experiment, we selected six coverage groups for the analysis. In total, they contain 4204 coverage points. The analysis was applied at a late stage of the verification cycle. Because of this, the test plan had been improved over some time. Consequently, most of the points had been

covered by the test plan and only 75 points remained uncovered. The test plan augmentation was applied to address these 75 remaining points.

Table 5.1 shows the result of applying the test plan augmentation algorithm. The iterative clustering partitions the 572 test items into five clusters, labelled as A to E in the table. The 2nd row shows the number of test items in each cluster.

The 3rd row shows the number of groups found for each cluster. The number of choices in a group ranges from a few to more than 70. The "# of permutations" row shows the number of resulting test items by permuting all choices across all groups. As noted before, these test items include the original test items in the data. The last row shows the number of newly generated test items. Simulation results found no invalid test item among them.

Cluster	A	B	C	D	E
# of test items	394	68	28	12	70
# of groups	7	5	3	3	2
# of permutations	540	90	45	15	70
# of new test items	146	22	17	3	0

TABLE 5.1: Result of test plan augmentation

To see the impact of each test item, each of the new test items was given to the randomizer to produce a single test. A test can have a few hundreds to a few thousands transactions. While the number of transactions intended for a core is fixed, depending on system configuration and the number of cores involved, the number of total transactions can differ.

Coverage group	C_1	C_2	C_3	C_4	C_5	C_6	total
Augment	31	5	11	3	6	0	56
Manual	0	1	0	15	0	3	19

TABLE 5.2: Coverage gain of test plan augmentation

Table 5.2 shows the coverage impact by the test plan augmentation approach ("Augment"). Because the augmentation does not cover all 75 coverage points,

we performed manual analysis to ensure that the rest of the points were covered. These results are shown in the row "Manual." As the table shows, 56 out of 75 points (74.66%) are covered by the augmentation.

5.7 Summary

This chapter studies the fundamental difference between processor core verification and platform verification from the perspective of developing a data learning methodology. We show that the rule learning approach proposed before for processor verification is not suitable for platform verification and illustrate the reason why it does not work. A different data learning problem is formulated for platform verification. It is unsupervised, and analyses a collection of test items to augment them. We propose an algorithm to tackle the new learning problem and show that it can address over 70% of the missing coverage points based on an experiment run on a latest commercial platform design.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

This dissertation evaluates the feasibility and effectiveness of developing practical data learning methodologies for functional verification, more specially, constrained random verification. Our proposed methodologies were developed based on the verification environment of commercial microprocessors and SoC platforms and can be used as complementary approaches to the existing verification flow. We focus on the problem of verification test generation and the goal is to discover knowledge for improving the effectiveness of the tests in terms of reaching satisfactory coverage level. We do not intend to provide a one-click solution to the problem but rather learning components that can be

used iteratively to provide verification engineers with interpretable and actionable knowledge. We propose data learning methodologies for three application scenarios where such knowledge is desired for making informed decisions.

The first application is to decide which tests to simulate. As illustrated in Chapter 3, there are many redundant tests in simulation and thus it is not necessary to simulate all of them. The novel test detection framework described in Chapter 3 helps identify the novel tests that are most likely to contribute to coverage increase. Experimental results show that it can achieve up to 95% saving of simulation cost.

The second application is to decide how to refine a test template to increase the chances of hitting certain coverage events. The rule learning methodology proposed in Chapter 4 extracts special properties from the novel tests that can be used to guide the test template refinement. Our experiments conducted parallel to the on-going verification efforts show that the proposed methodology can help hit coverage events that otherwise had low or zero coverage in extensive simulation.

The third application is to decide what test items can be added to augment a test plan. The test plan augmentation method proposed in Chapter 5 analyses a collection of existing test items and discovers the possible augmentation to make the test plan more complete. The proposed methodology is shown to be able to address over 70% of the missing coverage points based on an experiment run on a latest commercial platform design.

6.2 Future Research Directions

The works in this dissertations show the promise of building practical learning framework for functional verification. This section discusses the future research directions that can be extended from the works reported in this dissertation.

- **Verification knowledge management:** In the knowledge extraction framework, the extracted rules can be stored in a database and expanded along the verification process. This serves a verification knowledge database. We already show that the knowledge can be reused to refine the test templates in Chapter 4. Moreover, the knowledge can be used to help design kernels in the novelty detection framework proposed in Chapter 3, as another approach to overcome the limitation of single-instruction database. The verification knowledge can evolve as the design evolves. A lot of research can be done on how to manage the verification knowledge to make the best use of it.

- **Intelligent feature selection:** A good feature set is crucial for the rule learning framework to succeed. In our current framework, the user selects the important microarchitecture states to start with and the simple learning scheme enables the quick search over a set of features. Intelligent feature selection schemes can be explored to alleviate the manual efforts. For example, if the current feature set is not effective, selection of features that have less mutual information with the current ones is likely to provide extra information needed.

- **Confidence metrics for learning:** The data learning approaches are not guaranteed to succeed in all cases. When evaluating the knowledge

output from learning, the user desires to know to what extent he/she can trust the result. The development of such confidence metrics is very important for the adoption of a data learning framework in practice.

Bibliography

[1] The International Technology Roadmap for Semiconductors (ITRS), System Drivers, 2009, http://www.itrs.net/.

[2] Noah Bamford, Rekha K. Bangalore, Eric Chapman, Hector Chavez, Rajeev Dasari, Yinfang Lin, and Edgar Jimenez. Challenges in system on chip verification. In *Proceedings of the Seventh International Workshop on Microprocessor Test and Verification*, pages 52–60, 2006.

[3] Daniel Geist, Giora Biran, Tamara Arons, Michael Slavkin, Yvgeny Nustov, Monica Farkas, Karen Holtz, Andy Long, Dave King, and Steve Barret. A methodology for the verification of a system on chip. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 574–579, 1999.

[4] T. Schober, B. Hoppe, S. Landa, and R. Morad. Ibm system z functional and performance verification using x-gen. In *High Level Design Validation and Test Workshop, 2008. HLDVT '08. IEEE International*, pages 93–100, 2008.

[5] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of Data Mining*. MIT Press, Cambridge, MA, USA, 2001.

[6] Y. Katz and et al. Learning microarchitectural behaviors to improve stimuli generation quality. In *ACM/IEEE Design Automation Conference*, pages 848 –853, 2011.

[7] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[8] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std. 1800*, 2012.

[9] Jun Yuan, C. Pixley, A. Aziz, and K. Albin. A framework for constrained functional verification. In *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 142 – 145, nov. 2003.

[10] A. Aharon, B. Dorfman, E. Gofman, M. Leibowitz, V. Schwartzburd, and A. Bar-David. Verification of the ibm risc system/6000 by a dynamic biased pseudo-random test program generator. *IBM Syst. J.*, 30(4):527–538, October 1991.

[11] A. Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, Gil Shurek, and Yossi Malka Charlotte Metzger. Test program generation for functional verification of powepc processors in ibm. In *Design Automation, 1995. DAC '95. 32nd Conference on*, pages 279–285, 1995.

[12] 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. *Design Test of Computers, IEEE*, 21(2):84–93, Mar 2004.

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

[14] Laurent Fournier, Avi Ziv, Ekaterina Kutsy, and Ofer Strichman. A probabilistic analysis of coverage methods. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):38:1–38:20, October 2011.

[15] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 970–975, 1999.

[16] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 175–180, 1999.

[17] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 678–683 Vol. 2, March 2005.

[18] D. Moundanos, J.A. Abraham, and Y.V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *Computers, IEEE Transactions on*, 47(1):2–14, Jan 1998.

[19] Nina Saxena, Jacob Abraham, and Avijit Saha. Causality based generation of directed test cases. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, pages 503–508, New York, NY, USA, 2000. ACM.

[20] Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtenstein, Shmuel Ur, and Yaron Wolfsthal. Coverage-directed test generation using symbolic techniques. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1996.

[21] Heon-Mo Koo and Prabhat Mishra. Test generation using sat-based bounded model checking for validation of pipelined processors. In Gang Qu, Yehea I. Ismail, Narayanan Vijaykrishnan, and Hai Zhou, editors, *ACM Great Lakes Symposium on VLSI*, pages 362–365. ACM, 2006.

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

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

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

[25] Kerstin Eder and et al. Inductive logic programming. chapter Towards Automating Simulation-Based Design Verification Using ILP, pages 154–168. Springer-Verlag, Berlin, Heidelberg, 2007.

[26] O. Guzey and et al. Functional test selection based on unsupervised support vector analysis. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 262 –267, june 2008.

[27] Po-Hsien 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.

[28] Hu-Hsi Yeh and Chung-Yang Huang. Automatic constraint generation for guided random simulation. In *Asia and South Pacific Design Automation Conference*, pages 613–618, 2010.

[29] O. Guzey and et al. Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(1):138 –148, jan. 2010.

[30] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[31] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[32] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[33] Nada Lavrač, Branko Kavšek, Peter Flach, and Ljupčo Todorovski. Subgroup discovery with cn2-sd. *J. Mach. Learn. Res.*, 5:153–188, December 2004.

[34] 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.

[35] Power Architecture. http://www.power.org.

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

[37] Peter Clark and Tim Niblett. The cn2 induction algorithm. *Mach. Learn.*, 3(4):261–283, March 1989.

[38] Ljupco Todorovski, Peter A. Flach, and Nada 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.

[39] Chengqi Zhang and Shichao Zhang. *Association rule mining: models and algorithms*. Springer, 2002.

[40] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[41] D. Dueck and B.J. Frey. Non-metric affinity propagation for unsupervised image categorization. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, Oct 2007.

[42] D. Comaniciu and et al. Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[43] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances In Neural Information Processing Systems*, pages 849–856. MIT Press, 2001.

[44] StephenC. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32 (3):241–254, 1967.