Online Selection of Effective Functional Test Programs Based on Novelty Detection *

Po-Hsien Chang, Dragoljub (Gagi) Drmanac, Li.-C Wang
Department of ECE, UC-Santa Barbara

ABSTRACT

This paper proposes an online functional test selection approach based on novelty detection. Unlike other test selection methods, the idea of this paper is selecting novel functional tests to improve coverage from a large pool of available test programs before simulation. A graph based encoding scheme is developed to measure the similarity between test programs and map them into a set of feature vectors. We employ one-class SVM as the learning algorithm to detect novel tests to be simulated. While leaving the general test selection framework unchanged, the developed test program similarity measure can easily be tailored to specific applications and coverage targets based on existing simulation results. Experiments on a public domain MIPS processor design are presented to demonstrate the effectiveness of the approach.

Categories and Subject Descriptors

1.2 [**SYNTHESIS, VERIFICATION AND PHYSICAL DESIGN**]: Simulation and Formal Verification

1. INTRODUCTION

Functional verification continues to be the key bottleneck that delays time-to-market during the design process. Practical functional verification relies on extensive simulation of automated and/or guided random testing because of its flexibility and scalability.

Traditionally, the effectiveness of tests is often measured through various coverage metrics [1] such as statements, branch, expression and toggle coverage. Each coverage metric is specifically defined based on a model of the design. If one wants to measure the effectiveness of a set of tests, they will be simulated on the device model and coverage will be measured based on the simulation results.

Generating good quality test sets automatically is challenging. The source of these challenges lies in the complexity of the coverage search space. Guided test generation methods intend to avoid the complexity of deterministic search in order to achieve the coverage goal in a short period of time. One of the popular ideas for guided test generation is constrained random verification (CRV) [2], where a user provides biases and constraints in test templates as inputs to CRV, guiding its automatic generation of functional tests.

*This work is supported by Semiconductor Research Corporation, project task 1848.001

Another popular test generation approach explored in [5]-[11] is coverage-directed test generation (CDTG). CDTG techniques dynamically analyze coverage results and automatically adapt the test generation process to improve coverage. While many promising techniques have been proposed, CDTG remains an on-going and active research area.

Simulating today's designs is slow and requires intensive computation. Due to the required test time and computational power, not all generated tests can be simulated. In order to execute more tests, it is crucially important to have an effective *test selection* scheme to identify tests that can achieve high coverage quickly. In this paper, we propose a functional test program selection framework which is independent of the test generation method mentioned earlier. This framework treats the selection process as *novelty detection* and includes a mechanism for correlating test similarity to coverage contribution. Based on this framework, we propose an online support vector machine (SVM) algorithm to select new tests to be simulated which can contribute more coverage compared to the already simulated test set.

The rest of the paper is organized as the following. Section 2 discusses the online functional test program selection in detail. Section 3 discusses novel test program detection and the related algorithm. Section 4 explains the details of kernel adjustment based on edit cost functions. Section 5 presents the experimental results and Section 6 concludes the paper.

2. ONLINE FUNCTIONAL TEST SELECTION

Suppose N test programs are given in a simulation environment. The test selection goal is to select a small subset of available test programs to achieve almost the same coverage objective as simulating the whole set.

Traditional functional test selection is preformed in two steps [12, 13]. The test selection process is illustrated in Figure 1. Suppose there are N tests in a pool of tests to be simulated. The first step is to evaluate the coverage results for all N tests by simulation as shown in the coverage table. A total coverage N is collected by accumulating all the coverage results from 1 to N. The second step is to select the smallest number (M) of tests whose total coverage M that can achieve the same coverage space covered as the whole set N. From this perspective, we see that selecting M tests with maximum coverage can be seen as an example of the set-coverage problem, which is a well-known NP-complete

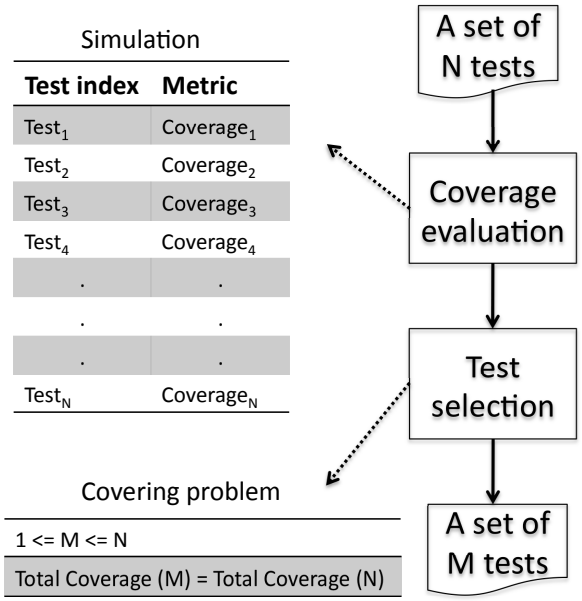

Figure 1: Test selection after simulation

problem. In practice, such a problem is usually tackled by a greedy algorithm, i.e. selecting the next test that maximizes the current coverage until there is no further coverage improvement with additional tests.

It is useful to solve the test selection problem in Figure 1 but it provides little help to the on-going functional verification effort because more tests are continually generated to hit the coverage holes in the design. We cannot assume that the coverage results shown in Figure 1 are available prior to the selection process, because we cannot afford to simulate all the generated tests due to time and computational power constraints. Therefore, selection has to be preformed before simulation in order to take advantage of the compact test selection results. The online selection approach is shown in Figure 2 and is designed to achieve this objective. In this approach, suppose i test programs are already simulated and a large pool of N new test programs are generated and ready to be simulated. The problem here is to select a small subset of the N available test programs that are most likely to increase coverage without any prior simulation.

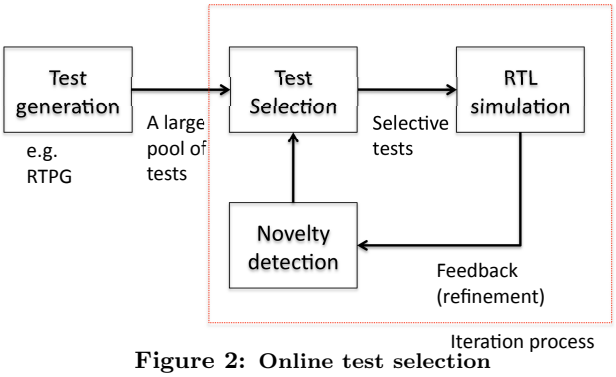

Figure 2: Online test selection

Similarly, authors in [14] avoided the use of simulation in coverage estimation by introducing the use of a *kernel* function [15] in conjunction with a learning algorithm to achieve functional test selection. The work in [14] was based on a CRV environment and test selection was designed for fixed-cycle functional tests. Results were shown on selected units of the OpenSparc T1 processor to demonstrate the feasibility of the approach. However, the idea of using kernel functions in place of coverage estimation was not fully explored in [14]. Moreover, the test selection scheme was limited to fixed-cycle functional tests. This work proposes a novelty detection approach which provides a more flexible kernel and extends to variant cycles test programs that are useful in practice.

3. NOVEL TEST PROGRAM DETECTION

Novelty detection is the identification of unforeseen data instances that are embedded in a large amount of data. This has been studied extensively in machine learning [15]. The basic idea of novel test program detection is to select novel tests which can contribute more coverage than similar tests compared to a set of already simulated programs. Figure 3 illustrates the concept of novelty detection applied in test selection. Functional test programs already simulated are given to a one-class learning algorithm to obtain a model that captures the boundary of the coverage area in an estimated coverage space. The learning model is applied to measure if a new test program (one not yet simulated) is inside the boundary or not. If it is inside, the test program is filtered out. If it is outside, a distance is calculate for the test program to measure how close it is to the boundary. Test programs outside the boundary are considered novel. The degree of novelty is measured as the distance to the boundary. This distance can then be used to select the m most novel test programs that will improve coverage.

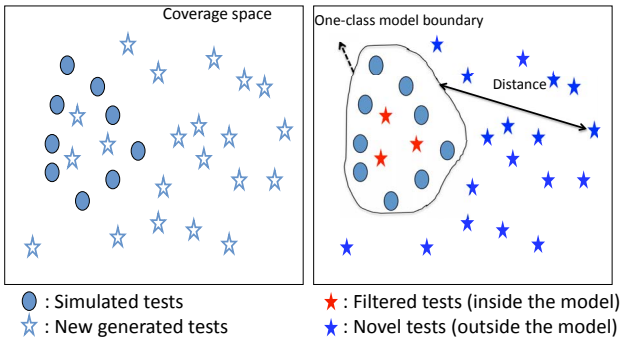

Figure 3: Test program selection by novelty detection

3.1 Kernel-Based Learning

For a learning algorithm to work in test selection, a kernel function is required to define the estimated coverage space shown in Figure 3 above. Figure 4 illustrates how a kernel based learning algorithm works conceptually [15]. With a dataset as input, a kernel-based learning algorithm to utilize the learning engine to build a learned model M which is a representation of this dataset.

A kernel-based learning algorithm consists of two parts: a kernel function $k()$ and an optimization engine. Moreover, the communication between the optimization engine and the kernel function is preformed by querying a pair of samples (x_i, x_j) and returning their similarity measure as $k(x_i, x_j)$. In other words, the optimization does not need to access the information from the original data sets to learn a model. Instead, all it needs is the similarity measure between pairs of data.

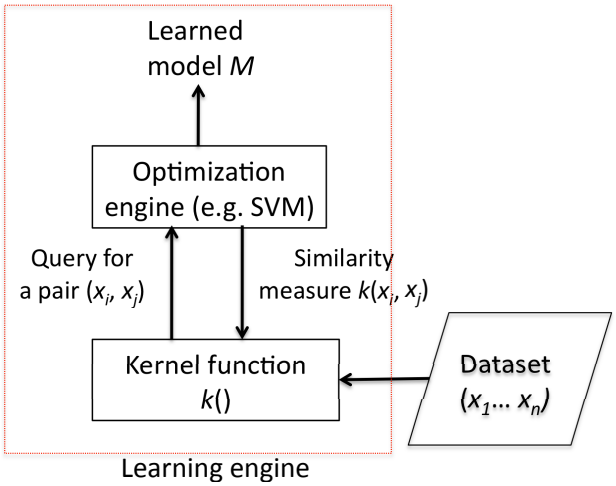

Figure 4: Illustration of kernel-based learning algorithm

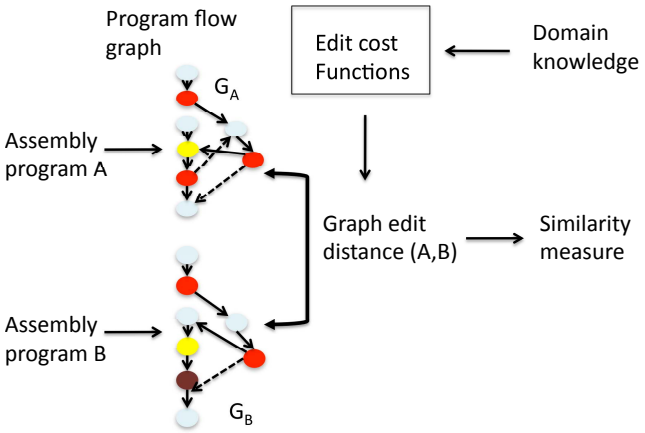

Figure 5: Similarity between two assembly programs

Address	Instruction		Vertex
4	add	a1,v0,2	A
8	j	0x22	B
12	addu	v1,v1,24	C
16	sb	v0,0(a1)	D
20	beq	v1,a1,0x28	E
22	sub	a0,a1,4	F
24	beqz	v0,0x16	G
28	addiu	v0,a2,10	H

Branch Arithmetic Memory

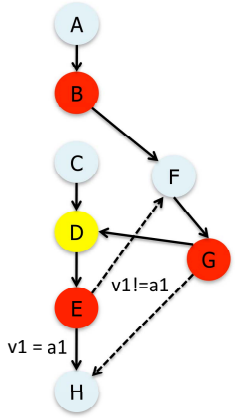

Figure 6: Assembly program in a graph representation

The distinction between kernel function and optimization engine in kernel-based learning makes it a powerful approach that can be applied in diverse areas [15]. For example, the SVM one-class algorithm, ν-SVM [16], is a particular optimization algorithm for computing a one-class model. The ν-SVM optimization algorithm is independent of the kernel used. Hence, one can design different kernels to be used with the ν-SVM optimization engine for different applications.

Figure 4 can be interpreted from another perspective. The optimization engine is responsible for computation during learning. The kernel function, on the other hand, captures a person's intuition about the learning problem. This intuition is encoded as a similarity measure using a kernel function. In other words, a kernel function is where a user can input domain knowledge to direct the learning process.

In this work, we employ one-class SVM as the optimization engine for novel test program detection. One-class SVM can only be applied on a set of vectors and is not directly applicable to test programs. Therefore, we convert assembly test programs into a set of feature vectors. A graph encoding scheme is developed in this paper to map assembly programs into a set of feature vectors based on the similarity measure between programs allowing one-class SVM and common kernel functions to be applied. The details will be discussed later in this section.

3.2 Graph Edit Distance

Figure 5 illustrates how to measure the similarity between two assembly programs based on a graph representation. An assembly program is converted into a directed graph as in the example shown in Figure 6. In this graph-based representation, a vertex represents an instruction, annotated by several attributes such as type of instruction, the opcode, the register usage, and the number of operands.

The edges in a graph capture the actual execution flow of an assembly program. The number of outgoing edges of a vertex is not restricted to one. Vertices that correspond to jump or branch instructions have two outgoing edges which depend on the conditional expression of the instruction. For example, vertex E in Figure 6 is a branch instruction which determines the destination of its outgoing edge by checking if v_1 is equal to a_1 or not.

Let $\mathcal{G}$ be the space of all possible program graphs. Let $\mathcal{R}$ be the space of real numbers. A graph kernel function is a mapping $k : \mathcal{G} \times \mathcal{G} \rightarrow R$. In our work, the kernel $k()$ takes two program graphs G_i, G_j and outputs a numerical value $k(G_i, G_j) \in [0\ 1]$ to quantify their similarity.

$\mathcal{K}(G_i, G_j) = e^{-\|d(G_i,G_j)\|}$ where $d(G_i, G_j)$ denotes the *graph edit distance* (GED) between G_i and G_j. GED defines the similarity of two graphs as the minimum amount of *edit operations* that are needed to transform one graph into another [19]. A standard set of the edit operations include, insertion, deletion, and substitution of vertices and edges. A sequence of edit operations $(e_1, \ldots, e_k)$ that transform a graph G_i into a graph G_j is called an *edit path*. In order to measure similarity, an edit cost function is defined assigning non-negative cost to each edit operation. To obtain the total cost of an edit path, individual costs of edit operations are accumulated. The similarity between the two graphs is defined as the minimum cost edit path between the two.

Figure 7 shows an example of a possible edit path between graph G_1 and G_2. Graph G_1 can be transformed into graph G_2 by the following edit path: vertex substitution (A→D), edge deletion, edge insertion, and another vertex substitution (B→E). The graph edit distance is calculated by summing the costs of these four edit operations.

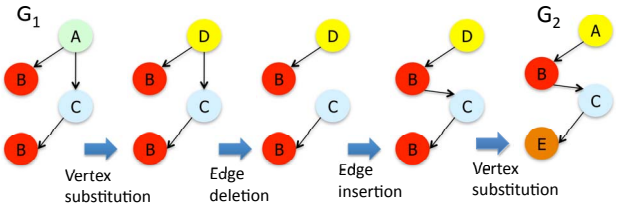

Figure 7: A graph edit path between G_1 and G_2

Using the graph edit distance as a similarity measure has several advantages: it captures the partial similarities between graphs; moreover, it introduces edit cost functions to quantify whether an edit operation influences graph similarity heavily or slightly. An edit cost function assigns a cost value to each edit operation reflecting the significance of the modification applied to the graph. A flexible edit cost function allows various types of domain knowledge to be incorporated. For example, substituting an add instruction with another add instruction has a lower cost than substituting an add instruction with a multiply instruction because they exercise different block in the design. This means that to integrate domain knowledge into test selection approach we only need to define the costs associated with all possible edit operations.

As mentioned earlier, three types of edit operation are used to compute the graph edit distance. The edit cost function C() can take several forms: C($i \rightarrow j$) denotes the cost of a vertex substitution i→j, C(i→ϵ) denotes the costs of a vertex i deletion, and C(ϵ→j) denotes the cost of a vertex j insertion.

When defining cost, both vertex type (opcode) and attribute of the vertex (operand) should be taken into account to achieve a better approximation of the true edit distance. Take three instructions in Table 1 as an example. C(A→B) is calculated as follows: C(addiu→addu) + C(v0,a2) where C(addiu→addu) denotes the cost between instruction addiu and add while C(v0, a2) denotes comparison of different registers used. Moreover, C(A→C) is calculated as C(addiu→beq). Note that the edit cost definition is a relative value rather than an absolute value and is dependent on the design. For instance, C(add→addiu) = 10 << C(add→beq) = 1000. Therefore, its values are different for other architectures while defining the same instruction substitution and can be refined based on measured coverage results. Since the definition of edit cost function does not affect the computation algorithm of graph edit distance, it can be easily expanded and refined for specific coverage targets. The details of edit cost functions adjustment will be discussed in section 4.

Table 1: Edit cost function

Instruction	Opcode	Operand
A	addiu	a1, v0, 2483
B	add	a1, a1, a2
C	beq	v1, a1, 0x238

Instruction substitution	Cost
add→add	0
add→addiu	10
add→beq	1000

Once the edit cost functions are set, the GED can be computed by a branch-and-bound like search algorithm, where possible edit paths are iteratively explored and the minimum-cost edit path can be retrieved from the search tree. This method can find the optimal edit path between two graphs. However, such a strategy may result in exponential search complexity, making it applicable to only small graphs. Therefore, this brute-force search algorithm is not practical.

A number of suboptimal methods [18, 19] were proposed to overcome the complexity issue. We implement the approach used in [19] which approximates graph edit distance by finding an optimal matching between vertices and local graph structure. The computation is based on bipartite graph matching using Munkres algorithm [20] as a heuristic. In the worst case, the algorithm complexity is O($(n+m)^3$) where n and m are the numbers of vertices in the two graphs. Note that O($(n+m)^3$) complexity is much smaller than the O($(n+m)!$) required by a brute force algorithm.

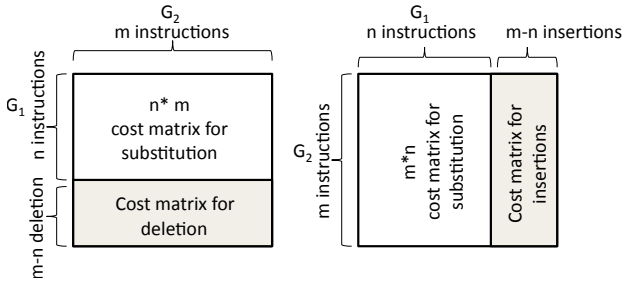

Figure 8: Cost matrix for transforming graph G_1 with n instructions into graph G_2 with m instructions and vice versa, for $m > n$

Figure 8 illustrates the basic concept of applying the bipartite matching heuristic. Suppose we are transforming a graph with n vertices into a graph with m vertices. The left side of the figure illustrates this situation. Each of the n vertices corresponds to a row in the cost matrix. Each of the m vertices corresponds to a column. Suppose $m > n$, then, $m - n$ deletions are required, which can occur for any one of the m instructions. Therefore, $m - n$ rows are added to define the costs for all possible deletion operations.

Given such a cost matrix, an edit path consists of selected entries in the matrix, covering each row with each column. The GED is therefore the minimum cost edit path based on the given cost matrix. Conversely, suppose we are transforming the graph with m vertices into the graph with n vertices. The right side of the figure illustrates this situation. In this case, $m-n$ insertions are required to expand the n instructions. A cost matrix for these insertions is therefore defined.

Entries in a cost matrix are defined based on domain knowledge and utilize recorded information in vertex annotations to estimate the cost for each operation. As described above, these annotations describe the structure of an instruction as well as its dependency to other instruction. Therefore, our intuition on how different types of annotated information will influence coverage and can be incorporated to define edit cost functions. It is important to note that this intuition is local to each edit operation individually.

3.3 Feature Vector Encoding

Suppose we are given N assembly programs, and obtain N program graphs $G_1, G_2, \ldots, G_N$ representing each program. For each graph G_i, the graph kernel function described above computes a similarity measure $k(G_i, G_j)$ between all pairs of programs G_i, G_j. Each assembly program is then converted to a feature vector $V_i = (k(G_1, G_i), k(G_2, G_i), \ldots, k(G_N, G_i))$ that records how similar G_i is to all other graphs (programs) illustrated as the symmetric matrix in Figure 9. The idea to map a test program into a feature vector representation is that can capture the similarity between programs in global view (i.e. compared to all other program graphs), rather than a local view (comparing only two program flow graphs).

n*n vector matrix	$Test_1$	$Test_2$	...	$Test_n$	
$Test_1$	$K(G_1,G_1) = \mathbf{1}$	$K(G_1,G_2)$	...	$K(G_1,G_n)$	V_1
$Test_2$	$K(G_1,G_2)$	$K(G_2,G_2) = \mathbf{1}$	...	$K(G_2,G_n)$	V_2
.	...	...	**1**	.	.
$Test_n$	$K(G_1,G_n)$	$K(G_2,G_n)$	...	$K(G_n,G_n) = \mathbf{1}$	V_n

Figure 9: Mapping the test programs into a set of feature vectors

After converting a set of assembly programs into as set of feature vectors, one-class SVM can be used to compare similarity between programs. Given two feature vectors V_i, V_j for graphs G_i, G_j, we can apply a common kernel functions such as the dot-product kernel or Gaussian kernel [15] to compute the similarity between the two programs. This allows newly generated programs (graphs) to be compared to the entire population of existing test programs. For example, the Gaussian kernel is written $\mathcal{K}(G_i, G_j) = e^{-g\|V_i - V_j\|^2}$ where $\|V_i - V_j\|^2 = \sum_{h=1}^{N}(K_{i,h} - K_{j,h})^2$ and g is a parameter that decides the Gaussian width that scales the similarity measure.

3.4 Kernel Comparison

To select subsets of programs that will improve coverage we use the One-Class SVM algorithm [16]. One-class SVM can be viewed as a novelty detection algorithm that assumes the most novel (unique) programs are near the origin while samples far from the origin are normal. Note that the similarity space is always contained in the positive quadrant since program similarity ranges from 0 (very dissimilar) to 1 (very similar). In the context of functional test program selection and the graph kernel similarity representation, the origin represents programs that are dissimilar to all others in the currently simulated population.

Figure 10 shows a program similarity space represented in two dimensions for illustration. Novelty detection in this space is preformed by identifying all programs that fall near the origin since this area reprints test programs with low similarity to the current test population. In reality, there will be many dimensions in the program similarity space (one for each simulated program) and several novel test programs may be identified. To detect novelty in the program similarity space a decision boundary is learned to maximally separate the vast majority of programs from the origin where few novel programs will be found. This boundary can subsequently be used to classify programs yet to be simulated as similar or novel given the already simulated population, enabling informed test program selection.

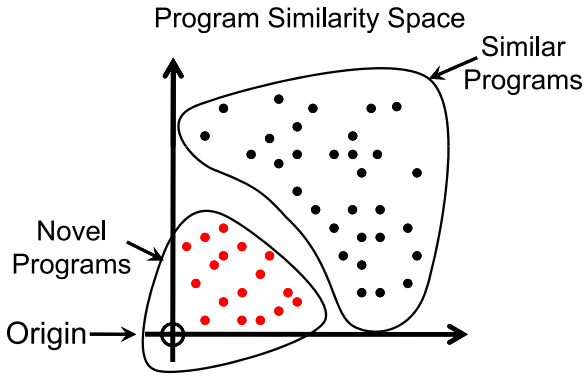

Figure 10: 2-D program similarity space

Given the graph edit distance representation and the One-Class SVM algorithm, it is essential to choose a top level kernel function to ensure samples near the origin have low similarity and samples far from the origin have high similarity. This will guarantee that programs similar to the existing test population are labeled normal while dissimilar programs are labeled novel. A popular kernel choice often used with SVMs is the Gaussian kernel [16]. Figure 11 shows how the Gaussian kernel computes novelty in a two dimensional similarity space. Since the Gaussian kernel is distance based we see that sample similarity is high along the diagonal where programs are nearly identical. However, this poses a problem for detecting novel program with one-class SVM since programs falling diagonally near the origin are treated as similar rather than novel. For this reason the Gaussian kernel is not ideally suited to detecting novelty using program similarity as defined by the graph edit distance. An alternative choice for the top level kernel is the dot product kernel.

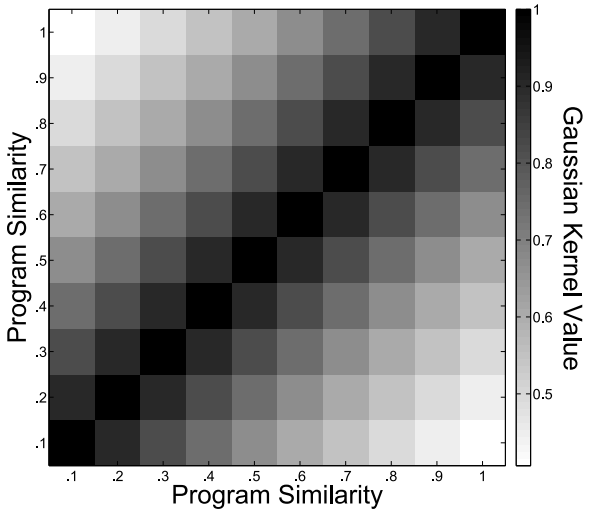

Figure 11: Gaussian kernel in a 2-D similarity space

Figure 12 shows the distribution of the dot product kernel output in a two dimensional program similarity space. The gray levels in Figure 12 show that a new test program falling near the origin is treated as novel while programs falling far

from the origin (near (1,1)) are treated as similar. Comparing this to the Gaussian kernel in Figure 11 we see that similar areas (darker regions) only occur near the top right corner while the origin is considered novel. With this construction we can use the One-Class SVM novelty detection algorithm to detect and select novel test programs.

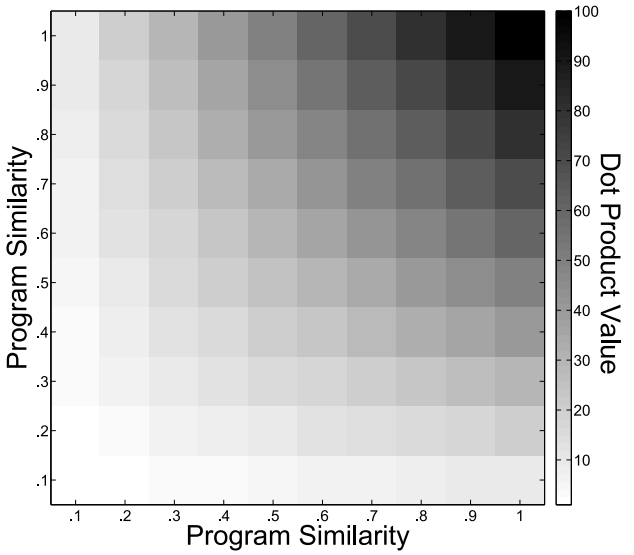

Figure 12: Dot product kernel in a 2-D similarity space

3.5 One-Class SVM

Novelty detection is performed using the one-class ν-SVM algorithm. Each program is represented by a feature vector $\vec{V_i} = (s_1, s_2, ..., s_n)$ where s_i is the similarity of a new test programs to the current test program population and n is the current number of simulated programs. The one-class algorithm maximally separates samples from the origin using the following quadratic programming optimization, ensuring that programs labeled novel are those with very little similarity to the already simulated population.

$$Minimize \ \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j K(\vec{V_i},\vec{V_j})$$

$$Subject\ to \ 0 \le \alpha_i \le \frac{1}{\nu m}, \ \sum_{i=1}^{m}\alpha_i = 1.$$

Where $\alpha_{i,j}$ are Lagrangian multipliers, $\vec{V_{i,j}}$ are program examples, ν is the fraction of most unique test programs we want to find, and $K(\vec{V_i},\vec{V_j})$ is the dot product kernel described above. After optimization is complete, programs with nonzero Lagrangian multipliers are called support vectors and define the novelty detection boundary. Once the boundary is established a novelty measure for each sample can be computed using the function $g(\vec{x})$ shown below.

$$g(\vec{x}) = \sum_{i=1}^{m}\alpha_i K(\vec{x},\vec{V_i}) - \rho$$

In general, there will be several novel programs of various degrees, so it is important to perform ranking to select the most novel programs. $g(\vec{x})$ returns positive and negative numbers, where positive numbers correspond to normal programs and negative numbers correspond to novel programs. Using $g(\vec{x})$ it is possible to select the most novel programs compared to the currently simulated population to further improve coverage.

4. KERNEL REFINEMENT BASED ON EDIT COST FUNCTIONS

The graph edit distance concept used in the graph kernel allows us to measure the similarity between program flow graphs. A major difficulty is that the edit distance requires an accurate definition of edit cost functions, which eventually determine which tests are similar in coverage contribution. The edit cost functions can be defined in a manual fashion where users input the domain knowledge such as similarity of specific instructions. However, domain knowledge may not be enough to accurately set the edit cost functions, in this case, learning from existing coverage results can result in a more accurate cost function and therefore a more accurate graph kernel. While leaving the general test selection framework unchanged, the graph kernel can easily be tailored to specific applications and coverage goals by adjusting the definition of the edit cost functions.

Moreover, if we would like to focus on a specific block instead of a whole design, we can adjust the edit cost functions to achieve this goal. Figure 13 illustrates how the cost edit functions influence the similarity measure .

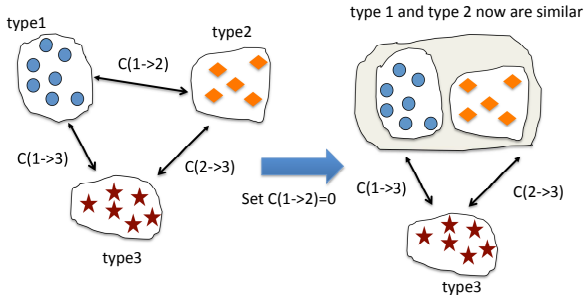

Figure 13: Edit cost function adjustment

Consider three types of instructions and each of which exercise different functional blocks in a design: block 1, 2 and 3 as shown in Figure 13. For example, add instructions such as (ADDU, ADDI, ADDIU) exercise the ALU block and shift instructions (SLL, SLV, SRA) exercise the shifter block. In this case the cost distance between instructions of similar types are set smaller relative to the cost associated with different instruction types. Suppose the current coverage for block 3 is relatively low, we would like to improve the coverage by selecting more tests which can specifically target that block. In order to achieve this objective, we can decrease the edit distance by treating type 1 and type 2 instructions as more similar in the coverage space. Therefore, test programs containing type 1 and 2 instructions are considered more similar and test programs containing type 3 instructions become more dissimilar after the edit distance adjustment. During subsequent test selection more programs that excite block 3 will be selected and the coverage contribution for block 3 can improve quickly. However, the coverage in other blocks may get worse since few tests related to those blocks are chosen to be simulated.

5. EXPERIMENTAL RESULTS

We conduct experiments based on a public domain 32-bit RISC processor design, the Plasma/MIPS CPU core [17]. The RTL design of the Plasma CPU is described in VHDL with about 4800 lines of code. The Plasma core executes all MIPS I(TM) user mode instructions. Table 2 shows the

major blocks of the Plasma core. All the experiments are run on a Intel(R) Xeon(TM) machine with 3gb ram.

Table 2: Plasma Block

Block name	Purpose
pc_next	Program Counter Unit
mem_ctrl	Memory Controller
control	Opcode Decoder
reg_bank	Register Bank for 32, 32-bit Registers
bus_mux	BUS Multiplex Unit
alu	Arithmetic Logic Unit
shifter	Shifter Unit
mult	Multiplication and Division Unit

To demonstrate the effectiveness of the proposed online functional test selection approach, we generated 2000 assembly programs to be the total test set. To obtain a high quality set of tests, the 2000 assembly programs generated in two steps. We first randomly generated 1000 tests and preformed coverage analysis in order to subsequently target more coverage holes. Based on the results from the initial 1000 tests, we generated another 1000 tests with modified RTPG constraints and bias. In this way, we started with a high quality test set that achieved high coverage across the tested design.

Each set of random test programs was generated based on random selection from more than 50 types of instructions such as arithmetic, logic, branch, load, store, and jump supported by the PLASMA core. Each randomly generated assembly program has various numbers of instructions, ranging from 80 to 100, including the boot sequence used to initialize the core. All programs share the same boot sequence that consists of about 50 instructions. Testbenches were set to have a clock cycle time of 50 ns and each assembly program took around 10 to 20 ms to simulate. Several coverage metrics are evaluated during the simulation, including statement, branch, expression, condition, and toggle coverage.

Since the boot sequence provides baseline coverage of every test program, Table 3 shows the coverage results achieved by the boot sequence and 250 randomly selected tests, compared to the final coverage results achieved by running all 2000 test programs. The coverage results from the boot sequence should be seen as the starting point for accessing the coverage contribution of each selected test program, while the 250 random tests help illustrate how much additional testing is required to push coverage several percent higher using traditional methods.

Table 3: Overview of test programs

Coverage metric	boot	250 tests	2000 tests
Statements	57.5	93.9	97.2
Branches	60.6	93.2	95.4
Conditions	63.0	87.7	89.1
Expressions	76.9	93.3	96.2
Toggle	52.4	82.8	84.8

Figure 14 shows the relationship between coverage difference and graph edit distance on a scatter plot. The coverage difference is calculated by the difference of toggle coverage metric between two tests and correlated its value with the graph edit distance between two tests. The correlation coefficient is 0.72 which can be considered a strong positive correlation. This means the similarity measure defined by the edit distance can capture similarity in the test coverage space fairly well.

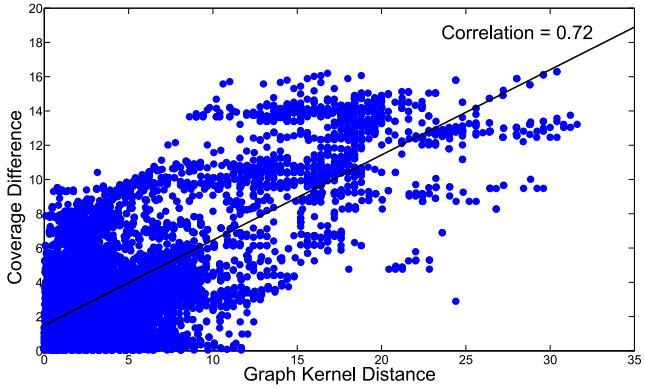

Figure 14: Coverage difference vs edit distance

For comparison, we run the selection experiment using the dot product kernel and the gaussian kernel. Initially, we randomly select 50 tests as the starting set. In the next iteration, 50 most novel test programs detected by the one-class SVM learning model based on two kernels are selected. Table 4 summarizes the coverage results for the two different kernels during the first iterations. The results shown in Table 4 are consistent with our intuition to choose the dot product kernel over the gaussian kernel.

Table 4: Dot product kernel v.s Gaussian kernel

Kernel	Initial	Dot product	Gaussian
Statements	81.0	87.9	81.0
Branches	80.6	87.7	81.1
Conditions	75.0	84.8	75.0
Expressions	88.5	92.3	88.5
Toggle	76.4	79.8	77.9

To demonstrate the difference between novel tests and similar tests in terms of coverage contribution, we run the two experiments based on the dot product kernel. When selecting the next 50 tests to be simulated in each iteration, we choose the 50 most novel tests and the most 50 similar tests for comparsion. Table 5 and Table 6 show the coverage results when selecting novel and similar tests across four iterations. Observe that the novel test programs have better coverage contribution than the similar test programs. Figure 15 shows the average coverage during these four iterations. This confirms our intuition that selecting programs that are novel compared to the existing test population will provide more coverage.

Table 5: Selecting novel test programs for 4 iterations

Iteration	Initial	1st	2nd	3rd	4th (250)
Statements	81	87.9	92.8	94.3	97.2
Branches	80.6	87.7	91	93.4	95.4
Conditions	75	84.8	89.1	89.1	89.1
Expressions	88.5	92.3	92.3	92.3	96.2
Toggle	76.4	79.8	81.5	83.4	84.8

Based on Table 5 we see that online iterative test selection can achieve the same coverage with 250 intelligently selected test programs as the full 2000 test set. It takes around 12 hours CPU time to simulate all the 2000 test programs. The total CPU time for simulating the 250 selected tests is 0.5 hours during the four iterations where the bottleneck is the

Table 6: Selecting similar test programs for 4 iterations

Iteration	initial	1st	2nd	3rd	4th (250)
Statements	81	82	83.2	88.6	88.6
Branches	80.6	81.3	82.6	87	87
Conditions	75	75	75	83.7	83.7
Expressions	88.5	88.5	88.5	88.5	88.5
Toggle	76.4	76.4	77.9	79.3	79.3

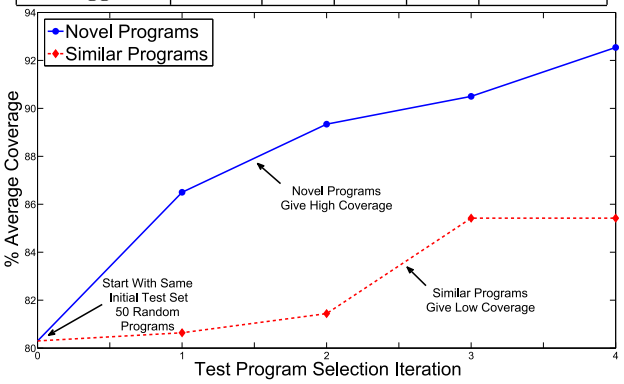

Figure 15: Average coverage contribution between similar vs unique test programs

computation of graph edit distance and 2 hours for running 250 test programs. Our method can achieve a reduction of 80% in simulation time. This shows that the concept of program similarity can accurately identify unique tests and avoid test redundancy. Moreover, this method was shown to improve an already optimized test set that was generated by biasing and constraining RTPG. This is of particular interest due to the tremendous test speedup that can be achieved by removing redundancy when generating thousands of extra tests to slightly improve coverage.

We conducted another experiment to demonstrate how kernel adjustment can be used to preform targeted coverage. Based on the same initial set of 50 tests as in other experiments, we select 50 novel tests to be simulated with two different kernels and compare the results. Table 7 shows the coverage results on a memory control block before and after we adjust the edit cost functions in order to select tests to target on the control block. Experimental results show the new kernel can achieve better coverage contribution on this block with the same number of tests.

Table 7: Coverage results on memory control block

Iteration	initial	1st (original)	1st(adjust)
Statements	73.4	84.3	90.4
Branches	68.1	81.2	87
Conditions	33.3	66.7	66.7
Expressions	100	100	100
Toggle	93.6	96.8	96.8

6. CONCLUSIONS AND FUTURE WORK

In this work, we proposed an online functional test program selection approach which utilizes the kernel-based learning algorithm. We developed a graph based encoding scheme to map assembly programs into a set of feature vectors based on the similarity measure between programs allowing one-class SVM and common kernel functions to be applied. While leaving the general test selection framework unchanged, the kernel can easily be tailored to specific applications and coverage goals by adjusting the edit cost functions. Experimental results show that the proposed approach was able to reduce the number of test programs required to achieve high coverage target. Moreover, the test selection method is effective on highly optimized test sets generated by constrained and biased RTPG, reducing the total time by 80% while maintaining the same coverage.

Several future possible directions of this research can be explored. Our future work will seek to apply this approach on other designs and/or in different applications. For example, to select functional tests for post-silicon validation such as circuit marginality. Besides, the complexity of GED computation for the similarity measure could be improved.

7. REFERENCES

[1] S. Tasiran and K. Keutzer, Coverage metrics for functional validation of hardware designs *IEEE Design Test*, 2001, vol. 18, no. 4, pp. 36 - 45.

[2] J. Yuan, C. Pixley, A. Aziz, and K. Albin. A Framework for constrained functional verification. Proc. *ICCAD*, 2003.

[3] Aharon Aharon, et al., Test Program Generation for Functional Verification of PowerPC Processors in IBM. Proc. *DAC*, 1995.

[4] Aharon Aharon, et al., Verification of the IBM RISC System16000 by a dynamic biased pseudo-random test program generator. *IBM Systems J.*, Vol 30, No 4, 1991.

[5] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, A study in coverage-driven test generation. Proc. *DAC*, 1999, pp. 970 - 975.

[6] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv, Cost evaluation of coverage directed test generation for the IBM mainframe. Proc. *ITC*, 2001, pp. 793 - 802.

[7] P. Mishra and N. D. Dutt, Functional coverage driven test generation for validation of pipelined processors. Proc. *DATE*, 2005, pp. 678 - 683.

[8] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y.Wolfstahl, Coverage-directed test generation using symbolic techniques. Proc. Int. Conf. Formal Methods Comput.-Aided Design, 1996, pp. 143 - 158.

[9] S. Fine and A. Ziv, Coverage directed test generation for functional verification using Bayesian networks. Proc. *DAC*, 2003, pp. 289 - 261.

[10] H.-W. Hsueh and K. Eder, Test directive generation for functional coverage closure using inductive logic programming. Proc. *IEEE HLDVT*, 2006, pp. 11 - 18.

[11] Onur Guzey and Li-C. Wang, Coverage-directed test generation through automatic constraint extraction. Proc. *IEEE HLDVT*, 2007.

[12] V Gangaram, D Bhan, JK Caldwell Functional Test Selection for High Volume Manufacturing. *MTV*, 2006.

[13] J Kang, SC Seth, V Gangaram Efficient RTL Coverage Metric for Functional Test Selection. *VTS*, 2007.

[14] Onur Guzey, Li-C Wang, Jeremy Levitt and Harry Foster, Functional test selection based on unsupervised support vector analysis. Proc. *DAC*, 2008, pp. 262 - 267.

[15] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*, Cambridge Univ. Press, 2004.

[16] Bernhard SchŽlkopf, and Alexander J. Smola. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, 2001.

[17] PLASMA at http://www.opencores.com/project,plasma

[18] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini and C. Watkins, Text classification using string kernels, *J. Mach. Learn. Res. 2*, pp. 419 444, 2002.

[19] K. Riesen, M. Neuhaus, and H. Bunke, Bipartite graph matching for computing the edit distance of graphs, *Graph-Based Representations in Pattern Recognition*, 2007

[20] R. Wilson and E. Hancock, Structural matching by discrete relaxation, *IEEE Tran. on Pattern Analysis and Machine Intelligence*, Vol 19, pp. 634 - 648, 1997.