Functional test selection based on unsupervised support vector analysis

Conference Paper · July 2008

DOI: 10.1145/1391469.1391536 · Source: IEEE Xplore

4 authors, including:

L. C. Wang
Dalian University of Technology
148 PUBLICATIONS **2,526** CITATIONS

SEE PROFILE

Jeremy R Levitt
Stanford University
10 PUBLICATIONS **458** CITATIONS

SEE PROFILE

Harry Foster
Mentor Graphics
58 PUBLICATIONS **879** CITATIONS

SEE PROFILE

Functional Test Selection Based on Unsupervised Support Vector Analysis *

Onur Guzey, Li-C. Wang
University of CA - Santa Barbara

Jeremy Levitt, Harry Foster
Mentor Graphics Corporation

ABSTRACT

Extensive software-based simulation continues to be the mainstream methodology for functional verification of designs. To optimize the use of limited simulation resources, coverage metrics are essential to guide the development of effective test suites. Traditional coverage metrics are defined based on either a functional model or a structural model of the design. If our goal is to select a subset of tests from a set of tests, using these coverage metrics require simulation of the entire set before the effectiveness of tests can be compared. In this paper, we propose a novel methodology that estimates the input space covered by a set of tests. We use unsupervised support vector analysis to learn such a space, resulting in a subset of tests that represent the original set of tests. A direct application of this methodology is to select tests before simulation in order to reduce simulation cycles. Consequently, simulation effectiveness can be improved. Experimental results based on application of the proposed methodology to the OpenSparc T1 processor are reported to demonstrate the practicality of our approach.

Categories and Subject Descriptors:
B. Hardware, B.6 Logic Design, B.6.3 Design Aids

General Terms: Design, Verification

Keywords: Functional verification, Support Vector, Learning

1. INTRODUCTION

Functional verification continues to be a key bottleneck that drags time-to-market in a typical design process. Practical functional verification relies on extensive simulation of directed and/or guided random tests due to its flexibility and scalability. Although simulation based verification can be very effective, its success in terms of both total effort spent and final verification quality achieved, depends heavily on the quality of the tests in use.

The effectiveness of tests is often measured through various coverage metrics [1]. A coverage metric is defined based on a model of the design. If one wants to measure the effectiveness on a set of tests, these tests are simulated on the model and coverage is measured based on the simulation result.

More effective tests can achieve higher verification coverage in shorter time, which save simulation resources. However, generating effective tests for complex designs has always been a challenging problem. Typically, functional test generation falls into one of three categories:

- Direct tests manually created by designers.
- Constrained-random tests [2] instantiated randomly from biased and constrained test-benches developed by designers.
- Coverage-directed tests [3, 4, 5] dynamically generated by an algorithm to achieve higher verification coverage.

In practice, direct and constrained random tests are mostly used. Direct tests are written to cover corner cases and important features of a design. Writing direct tests has been a dominant test generation methodology because these tests may be crucial for verification. In many situations they can be the only tests that exercise specific corner cases. This important advantage is partly offset by the amount of manual effort involved to prepare the tests. Constrained-random test generation alleviates part of the problem by producing a large number of controlled random tests where many verification targets can be fortuitously covered. This reduces the need for direct tests. However, for complex designs, achieving a required verification coverage goal even with both approaches can still be very challenging. As a result, tremendous simulation cycles are spent before the simulation model of a design can be declared golden.

To reduce the simulation cycles needed for achieving a required coverage goal, this work takes an approach that is different from the existing ones. The core of the approach is to be able to estimate the input space covered by a given set of tests. The result of this estimation is a model for the covered space where this model utilizes only a subset of the tests. Then, instead of simulating the original set of tests, we use this model to select tests for simulation.

For example, assume that we have a test set $T = \{t_1, \ldots, t_n\}$. Suppose our application is to select a subset of l tests $T_s = \{t_{s1}, \ldots, t_{sl}\}$ such that T_s gives the most verification coverage. The naive approach is to simulate the n tests and obtain coverages for them. Based on their coverages, then one would apply a search algorithm to find the best subset. In our case, we want to avoid the simulation. Hence, we need to find an alternative. Figure 1 illustrates this alternative.

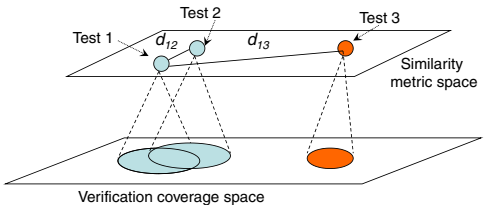

Figure 1: Defining a *similarity* metric space

Suppose somehow we can define a function k such that for any pair of tests t_i, t_j, $k(t_i, t_j)$ measures the *similarity* between them. This similarity function essentially creates a similarity metric space

*This work is supported in part by Semiconductor Research Corporation project task 1360.001.

DAC 2008, June 8–13, 2008, Anaheim, California, USA

where tests are projected onto. In other words, the input space of the tests is altered by the similarity estimate function $k()$, and becomes the similarity metric space.

The figure shows a simple example of three tests where test 2 is more similar to test 1 than to test 3. If we treat this space as a 2-dimensional Euclidean space and the similarity is measured by the distance, then we have $\frac{1}{d_{12}} > \frac{1}{d_{13}}$.

The key is that we want to define a k such that, if $k(\text{test 1}, \text{test 2}) > k(\text{test 1}, \text{test 3})$, the space covered by tests 1 and 2 in the actual verification coverage space will also likely be larger than the space covered by tests 1 and 3, as shown in the figure. Suppose that we have such a k function. Then, Figure 2 illustrates the test selection problem in the similarity metric space.

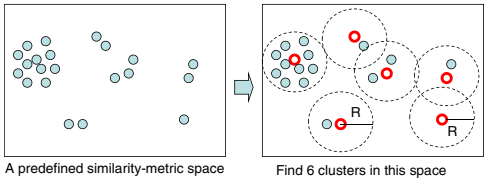

Figure 2: An example to illustrate the idea of selecting important tests based on relative similarity measures

Suppose we project a set of 21 tests on the similarity metric space as shown in the figure and our objective is to pick the most important 6 tests. The picture illustrates that, by drawing 6 circles with the same radius R we can cover all 21 tests. We see that all tests within a circle are more similar to each other than to tests outside the circle. Therefore, to pick 6 tests, intuitively we would choose 1 test from each circle. And as the picture shows, this test would be picked closer to the center because it minimizes the average distance to other points inside the circle.

The problem in Figure 2 can be seen as solving an *unsupervised learning* problem [6]. For example, clustering a set of data points is a typical unsupervised learning problem. In this context, the similarity measure function k is called a *kernel* function [7]. Therefore, we see that the alternative to the naive approach described earlier, can be to solve a kernel-based unsupervised learning problem [7].

Based on the two ideas just described, this work proposes a pre-simulation methodology to estimate the relative effectiveness among tests. In this methodology, we first define a kernel function that computes the similarity of two tests. Then, test selection is carried out by solving a kernel-based unsupervised learning problem.

Figure 3 depicts the overall picture of the work. A test set T_1 is given and unsupervised learning based on support vector analysis is used to generate a learned model M_T. This model captures the sub-space covered by the tests in T_1 (in the similarity metric space). After this model is generated it can be used as a filtering tool to screen out those tests that are similar to the tests already in T_1. A reduced set T_{s2} is produced. The objective is to have $T_1 \cup T_{s2}$ achieve a similar coverage as $T_1 \cup T_2$.

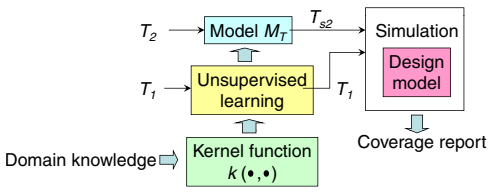

Figure 3: The overall picture of this work

If the proposed method can be used to select l most effective tests from a large test set, then it can be applied to re-order the tests in the set. For example, given a set of m tests where m is large, we can apply the method to select l tests. Then, we can apply the method again on the remaining $m - l$ tests to select a second set of l tests. This process can iterate and at the end, report an ordered list of $\frac{m}{l}$ subsets of l tests, ranked by their estimated effectiveness. If the estimation is accurate, then we can save simulation cycles. If it is inaccurate and the rank is mostly wrong, then in the worst case, we just need to simulate the original m tests. From this perspective, we see that the proposed method can be used in a non-intrusive way. We note that the learning run time is usually order-of-magnitude less than the simulation time (see section 5).

For the methodology to be effective, it demands a similarity (kernel) function that can reflect the similarity between the coverages of two tests. In this work, we will show that the proposed methodology allows domain knowledge to be incorporated into the kernel function $k()$. This is an important feature of the methodology. The analogy is that a commercial constrained-random test generation framework allows a user to incorporate input biases and constraints (domain knowledge) into a test bench. Our methodology allows biases and constraints to be put into the kernel function.

It is important to note that the learning algorithm only utilizes the kernel function as a relative measure rather than an absolute measure, i.e. to compare if a pair of tests is more similar than another pair or not. Hence, the exact coverage of a given test on a design model is unknown until after the test is simulated. This information is not required and never used in the proposed methodology.

The rest of the paper is organized as below. Section 2 briefly compares this work to others. Section 3 explains support vector analysis (SVM) [6] for unsupervised learning. Section 4 discusses the development of kernel function and how to incorporate domain knowledge such as constraints and biases into a kernel. Section 5 presents experimental results on OpenSparc T1 processor [12] to confirm the effectiveness of the proposed approach. Section 6 discusses applying our method to the test set selection problem rather than the test selection problem, and Section 7 concludes.

2. COMPARISON TO RELATED WORK

Coverage metrics [1][8] are most related to this work. Various coverage metrics were proposed in the past, including code coverage, structural coverage, FSM coverage, functional coverage, and design error coverage as categorized in [1]. As mentioned above, this work is not for measuring coverage. It is for test selection by analyzing tests in a similarity metric space.

Coverage hole analysis [9, 10] converts a large amount of coverage results into compact information that can be more effectively utilized to guide the test preparation. Our work, on the other hand, intends to convert a large amount of tests into a compact set of tests so that verification efficiency is improved by saving simulation cycles. For improving verification efficiency, a more intuitive approach is coverage-directed test generation (CDTG).

CDTG techniques dynamically analyze coverage results and automatically adapt the test generation process to improve the coverage. Some techniques utilize knowledge from high-level specification [4][5] [8] or restricted symbolic simulation [11] to help generate target tests. Automatic CDTG is an attractive approach because it allows test generation to work with coverage metrics to quickly produce a set of focus tests for achieving a high coverage.

Our work and CDTG techniques should be seen as two orthogonal approaches, although both aim to cut down the required number of simulation cycles by providing more effective tests. Our work tries to solve a test selection problem in the input space. The proposed methodology is designed to work independently of a simulation design model. From this perspective, it is similar to a commercial constrained random test generation framework. In both, a

user is required to supply domain knowledge to the tool. In a constrained random framework, this knowledge is in the constraints and biases. As illustrated in Figure 3 before, in our methodology, this knowledge is in the kernel.

3. SUPPORT VECTOR ANALYSIS

Suppose we are given a set of m tests $T = \{t_1, \ldots, t_m\}$. Without loss of generality, assume that each test t_i is encoded as a vector $\vec{x}_i$. Therefore, m tests become m vectors $X = \{\vec{x}_1, \ldots, \vec{x}_m\}$. Let k be a given kernel function $k(\cdot, \cdot)$ for similarity measurement. For example, for two vectors $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{z} = (z_1, \ldots, z_n)$ the simplest kernel function can be the dot product of the two vectors, i.e. $k(\vec{x}, \vec{z}) = \langle \vec{x}, \vec{z} \rangle = \sum_{i=1}^{n} x_i z_i$.

For a function $k(\cdot, \cdot)$ to be an admissible kernel function, the requirement is that $k(\vec{x_i}, \vec{x_j}) = \langle \vec{\phi}(\vec{x_i}), \vec{\phi}(\vec{x_j}) \rangle$ for some mapping function ϕ [7]. Hence, ϕ is the mapping that maps a test from the original input space into the similarity metric space explained in Figure 1 before. The standard way is to call it a *feature space*. Design of an admissible similarity kernel function will be discussed in Section 4. For now, we assume that such a k has been given.

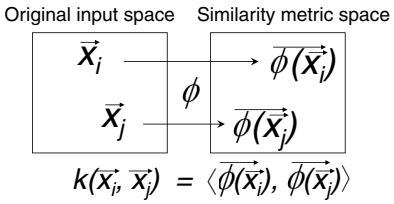

Figure 4: An admissible similarity kernel function $k(\vec{x_i}, \vec{x_j})$ corresponds to first mapping $\vec{x_i}, \vec{x_j}$ to $\vec{\phi}(\vec{x_i}), \vec{\phi}(\vec{x_i})$ and then computing the dot product $\langle \vec{\phi}(\vec{x_i}), \vec{\phi}(\vec{x_i}) \rangle$. in the similarity metric space.

3.1 The sub-space covered by the test set T

Our goal in the support vector analysis is to establish the boundary of the sub-space in the original input space such that this sub-space includes most of the tests. Figure 5 illustrates the idea.

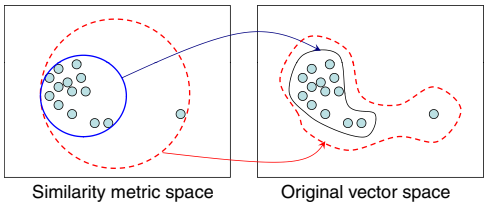

Figure 5: Find a hypersphere to include most of the points in the similarity metric space and the hypersphere corresponds to an irregular-shape region in the original input space

To find such a region in the original input space, one tries to find the smallest hypersphere in the similarity metric space that includes most of the points. In a 2-dimensional space, a hypersphere is a circle. A hypersphere in the similarity metric space then corresponds to an irregular region in the input space. This irregularity is implicitly dictated by the similarity kernel function k.

To see why we include most of the points instead of all points, observe the two circles in the similarity space. If we want to include all points, a much larger circle is required. The problem is that a larger circle contains more space where there is no point falling into it. Hence, in the example, the more appropriate way is to treat the rightmost point as an *outlier* and find the smallest circle to include all other points. Suppose the smaller circle in the similarity space can be represented by a model $S(\vec{\phi}(\vec{x}))$. Given a new point $\vec{x}$ we can compute $S(\vec{\phi}(\vec{x}))$ to decide whether it is inside the region or outside. If it is inside, we consider it to be similar to the points used to derive the model, i.e. it is similar to the tests in T if the model is built based on T. Otherwise, it is not similar to those points.

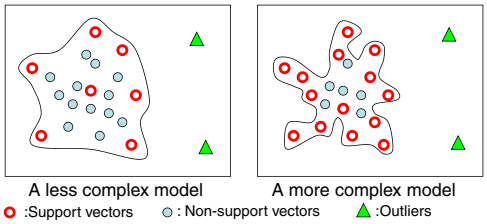

Figure 6: The kernel k defines the complexity of a model in the original vector space. A more complex model uses a larger number of SVs

As mentioned above, the shape of the region is decided by the kernel k. To define a region, a number of support vectors (SVs) are used. As shown in the Figure 6, a more complex (or irregular) region requires a larger number of SVs to define the boundary.

3.2 The equations

Given m vectors $X = \{\vec{x}_1, \ldots, \vec{x}_m\}$ and the kernel $k()$, the support vector model takes the form:

$$S(\vec{x}) = R^2 - \sum_{\forall i,j} \alpha_i \alpha_j k(\vec{x_i}, \vec{x_j}) + 2 \sum_{\forall i} \alpha_i k(\vec{x_i}, \vec{x}) - k(\vec{x}, \vec{x}) \quad (1)$$

For each $\vec{x_i}$, an α_i is computed. Only SVs have non-zero α_i. Hence, non-support vectors are not used in the model. The term $\sum_{\forall i,j} \alpha_i \alpha_j k(\vec{x_i}, \vec{x_j}) - 2 \sum_{\forall i} \alpha_i k(\vec{x_i}, \vec{x}) + k(\vec{x}, \vec{x})$ can be seen as a weighted average of distance squares between $\vec{x}$ and all SVs. The weights are decided by the alpha's. In the similarity metric space, R is the size of the radius of the hypersphere. $S(\vec{x}) \geq 0$ means that $\vec{x}$ is inside the region. Otherwise, it is outside.

$S()$ is found by solving a quadratic optimization problem using the Lagrangian method. Therefore, each α_i is nothing but the Lagrangian multiplier for $\vec{x_i}$. In the following, we show how to formulate the problem of finding the smallest hypersphere to include most points, into a quadratic optimization problem. We note that for support vector analysis, efficient algorithms are available for solving the optimization problem [7]. Hence, the formulation is only for explanation of the concept, not for implementation.

Given a set X of m points, the smallest hypersphere containing X is the point $\vec{c}$ such that it minimizes the distance r from the furthest point. To allow a point falling outside, we associate each point $\vec{x_i}$ a slack variable ξ_i.

$$\xi_i = \max(0, \|\vec{c} - \vec{\phi}(\vec{x_i})\|^2 - r^2) \quad (2)$$

Hence, ξ_i is the distance to the boundary of the hypersphere if $\vec{x_i}$ is outside. Otherwise, $\xi_i = 0$.

Let R be the size of the radius of the hypersphere. The optimization problem is formulated as the following.

$$\begin{array}{ll} \text{minimize} & R^2 + \sum_{i=1}^{m} \xi_i \\ \text{subject to} & \|\vec{\phi}(\vec{x_i}) - \vec{c}\|^2 \leq R^2 + \xi_i \\ & \xi_i \geq 0, \text{ for } i = 1 \ldots m \end{array}$$

With this formulation, the model leaves at least one outlier outside [7]. To solve it, we obtain the Langragian $\mathcal{L}(\vec{c}, R, \vec{\alpha}, \vec{\xi}) =$

$$R^2 + \sum_{i=1}^{m} \xi_i + \sum_{i=1}^{m} \alpha_i [\|\vec{\phi}(\vec{x_i}) - \vec{c}\|^2 - R^2 - \xi_i] - \sum_{i=1}^{m} \beta_i \xi_i \quad (3)$$

Differentiating $\mathcal{L}$ with respect to the primal variables in the original optimization formulation, we obtain a set of constraints. Substitute them into the Lagrangian, we obtain the dual problem formulation as the following.

$$\begin{array}{ll} \text{maximize} & \mathcal{L}(\vec{c}, R, \vec{\alpha}, \vec{\xi}) = \mathcal{L}(\vec{\alpha}) \\ & = \sum_{i=1}^{m} \alpha_i k(\vec{x_i}, \vec{x_i}) - \sum_{i,j=1,1}^{m,m} \alpha_i \alpha_j k(\vec{x_i}, \vec{x_j}) \\ \text{subject to} & \sum_{i=1}^{m} \alpha_i = 1 \text{ and } 0 \leq \alpha_i \leq 1, \text{for } i = 1 \ldots m \end{array}$$

It is interesting to observe that in the dual problem, the only variables to optimize are α's. Moreover, the mapping function $\vec{\phi}()$ disappear. Instead, only the kernel function $k()$ is used. What this means is that, as long as we can define a similarity kernel function k, we do not need to know what the mapping function $\vec{\phi}()$ really is because it is never used explicitly in the support vector analysis.

4. KERNEL FUNCTIONS

As illustrated in Figure 4 before, not all functions can be used as similarity kernel functions. A kernel function k has to satisfy the condition that $k(\vec{x_i}, \vec{x_j}) = \langle \vec{\phi}(\vec{x_i}), \vec{\phi}(\vec{x_j}) \rangle$ for some mapping function ϕ. This condition can be checked by the Mercer theorem [6] that basically says that $k()$ is a kernel function only if the kernel matrix $M_k = |k(\vec{x_i}, \vec{x_j})|_{i,j=1,1}^{m,m}$ is positive semi-definite. A positive semi-definite matrix has all its eigenvalues greater than or equal to zero. We note that M_k also has to be symmetric.

In general, suppose we come out with any similarity measure function $k'()$. Since it is easy to define such a function whose kernel matrix $M_{k'}$ is symmetric, the symmetry requirement is usually not the concern. If $M_{k'}$ is positive semi-definite, then it is a kernel. Suppose it is not positive semi-definite. Then, we can build a kernel function by taking $k(\vec{x_i}, \vec{x_j}) = \sum_{l=1}^{m} k'(\vec{x_i}, \vec{x_l}) k'(\vec{x_l}, \vec{x_j})$. Then, we have $M_k = M_{k'}^2$, i.e. M_k is the matrix square of another matrix. We note that the square of a symmetric matrix is always positive semi-definite because its eigenvectors remain unchanged but the corresponding eigenvalues are each squared (hence ≥ 0). Therefore, $k()$ becomes a kernel function.

In practice, the generic way to make a kernel function described above is not used often because it is more complex to compute $k()$. Two commonly used kernels are the dot product and Gaussian kernel. Given $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{z} = (z_1, \ldots, z_n)$, the dot product kernel is $k(\vec{x}, \vec{z}) = \langle \vec{x}, \vec{z} \rangle = \sum_{i=1}^{n} x_i z_i$, as mentioned before. The Gaussian takes the form $k(\vec{x}, \vec{z}) = e^{-g\|\vec{x}-\vec{z}\|^2}$ where $\|\vec{x} - \vec{z}\|^2 = \sum_{i=1}^{n} (x_i - z_i)^2$. g is a parameter that decides the Gaussian width that scales the similarity measure. Notice that for the two basic kernels, one uses the product $x_i z_i$ as the base similarity measure and the other uses the difference $|x_i - z_i|$.

Between the two basic kernels and the generic method, we would like to have a kernel that can take design related domain knowledge into account. This can be accomplished by applying the rules for making a kernel:

Combination If $k_1()$ and $k_2()$ are kernels, then their weighted average $k() = w_1 k_1() + w_2 k_2()$ is also a kernel. Furthermore, $k() = k_1() k_2()$ is a kernel [7].

Dataset manipulation Let X_1 be the data matrix given for the analysis. Suppose we manipulate the data to obtain a new matrix X_2. Let $k_1()$ and $k_2()$ be two kernels and $k_1()$ is applied to X_1 (kernel matrix M_1) and $k_2()$ is applied to X_2 (kernel matrix M_2). Then, $k() = w_1 k_1() + w_2 k_2()$ (kernel matrix M) is also a kernel. This is because the symmetric matrix obtained from adding two positive semi-definite symmetric matrices is also positive semi-definite.

Explicit mapping ϕ If we can establish an explicit mapping ϕ, then the corresponding kernel is just the dot produce defined in the ϕ space. By definition, that is a kernel.

4.1 Constraints and biases in a kernel

Before we show how constraints and biases can be put into a kernel, we need to explain how one may encode a test into a vector for support vector analysis. Given a test set $T = \{t_1, \ldots, t_m\}$, we need to first convert each test t_i into a vector $\vec{x_i}$. Without loss of generality, assume that each test t_i is a bit vector $(b_{i1}, \ldots, b_{il})$. Each bit b_{ij} takes three values, 0, 1, and unknown X. We note that t_i can be a multi-cycle input stimulus. Suppose the number of signals is l_1 and the number of cycles is l_2, we have $l = l_1 l_2$.

First we consider how to measure the similarity between two bits. Recall from above discussion that the similarity in the two basic kernels can be measured either by product or by difference. For convenience, we use *-* to denote the similarity of two bits where * can be one of the 0,1,X.

Suppose we desire an encoding that allows dot product and Gaussian kernel to behave similarly (This may not be true with any encoding scheme). For each bit in a test, we can use two variables to encode it. Each of these variables is called a *feature*. We use $(0, \sqrt{2})$ to encode logic 0, $(\sqrt{2}, 0)$ to encode logic 1, and $(1, 1)$ to encode X. With this encoding, we see that using the difference (where the absolute differences of the two numbers are summed) 0-0, 1-1, X-X all give 0. 0-X and 1-X give $2 - \sqrt{2}$. 0-1 gives $2\sqrt{2}$. Using the product, 0-0, 1-1, X-X all give 2. 0-X and 1-X give $\sqrt{2}$. 0-1 gives 0. We observe that the relative similarity measures between using the difference and using the product are consistent in the sense that if $k(\vec{x_1}, \vec{x_2}) < k(\vec{x_1}, \vec{x_3})$ is true with k being the dot product, replacing k with the Gaussian kernel would not change the relation. Note that with the encoding, 0-X and 1-X are considered less similar than that having the exact same two values.

With the encoding, each test t_i is encoded as a vector $\vec{x_i} = (x_{i1}, \ldots, x_{in})$ where $n = 2 * l$. Assume that we want to put a constraint into the kernel. For convenience, we use $b_1, \ldots, b_l$ to denote the l input signal variables. A constraint can be viewed as a logic equation based on a subset of the input signal variables. In the 3^l logic space (0,1,X), it defines a sub-space. By putting a constraint inside a kernel, we want the similarity measure inside this sub-space to be different from that outside.

Suppose we want tests inside the sub-space to be considered more similar with each other than with tests outside. The purpose of this can be that we want more tests to be selected from outside the sub-space. The left example in Figure 7-(a) shows how this can be done by modifying the data matrix.

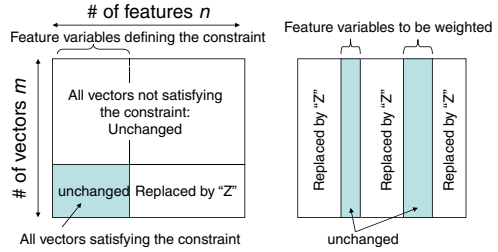

Figure 7: Putting constraints and biases into kernels by manipulating the data matrix X used to encode the test set T

Basically, given a data matrix, there are vectors that satisfy the constraint and vectors that do not. For those that do, we replace, with a special symbol Z, the values of all features that are not used in the definition of the constraint. The similarity of Z-Z is enforced, in the kernel computation, to be the highest value (most similar). In addition, the similarity of Z-* is enforced to be the lowest value (most dissimilar). After this modification, the kernel would interpret all vectors satisfying the constraint more similar to each other and also as a group more dissimilar to the rest (because Z is most dissimilar to all other values). During support vector analysis, dissimilar tests will more likely be selected, resulting in more selected tests from the vectors that do not satisfy the constraint.

Let $k_1()$ denote the kernel computation by applying one of the

basic kernels to the original data matrix. Let $k_2()$ denote the kernel computation on the modified dataset. We then let our final kernel to be $k() = k_1() + wk_2()$ where $w \gg 1$. By using a large weight w, we achieve two things: (1) Those vectors satisfying the constraint are considered to be more similar. (2) Among them, they can still be dissimilar, decided by the $k_1()$.

Suppose we want tests inside the sub-space to be considered more dissimilar with each other than with tests that are outside. The purpose of this can be that we want more tests to be selected inside the constrained sub-space. In this case, we just move Z to the upper right box in the example. Then, all vectors satisfying the the constraint would be considered more dissimilar. As a result, more tests will be selected from the vectors satisfying the constraint.

The right example in Figure 7-(b) shows how to put more weight on some variables. Again we modify the data matrix by replacing with Z, all entries in all columns whose corresponding feature variables are not considered. In this way, these feature variables do not participate in the kernel calculation. Then we use the same trick as before to make a kernel take the form $k() = k_1 + wk_3()$.

In general, one can incorporate any number of constraints and biased schemes on the variables, into the kernel. Suppose we have u constraints and v biases. The resulting kernel would look like:

$$k() = k_0() + w_1k_1() + \cdots + w_uk_u() + \cdots + w_{u+v}k_{u+v}() \quad (4)$$

If one have no further knowledge, a single largest weight can be assigned to all kernels from the constraints, followed by a large weight assigned to all kernels from the biases.

We note that the encoding scheme and the methods to manipulate the data matrix are examples. In general, one can design a different scheme or make a different kernel to better suit the application. It is also important to note that Figure 7 is for illustration purpose. In the implementation, the computation should take place on the fly when the value $k(\vec{x_i}, \vec{x_j})$ is actually needed. Hence, the implementation does not need to explicitly modify the data matrix before the support vector analysis is carried out.

5. EXPERIMENTAL RESULTS

OpenSPARC T1 is a 64-bit open-source microprocessor developed by SUN Microsystems[12]. For the experiments, we selected the execution unit. We assume that tests are applied at the input boundary of this unit in other words, we are performing unit-level verification. Execution unit (EXU) consists of around 10000 lines of RTL code and has 600 input bits. It features four arithmetic subunits, control blocks and a large number registers. EXU's internal integer register file features 128 registers.

For our experiments, we also create a constrained random testbench for the execution unit. We bias the inputs based on the statistics seen in the regression test sequences that are included in OpenSPARC's open-source release.

5.1 An illustrative example – ALU

Execution unit includes an Arithmetic Logic Unit (ALU). This block has a total of 335 input bits including 5 64-bit word-level inputs. We generate block level constrained random tests that exercise different logical and arithmetic operations. For the experiment we generate 300 tests that perform the logical AND operation. We collect input values during simulation to train a support vector model $S()$. After obtaining the model, we randomly generate additional tests covering all possible operations including the AND operation. For each of the additional tests, we use the model $S()$ to compute its similarity value. Result is shown in Figure 8.

A higher $S()$ value means that the test is more similar to the original tests used to train the model (AND operation tests). As shown

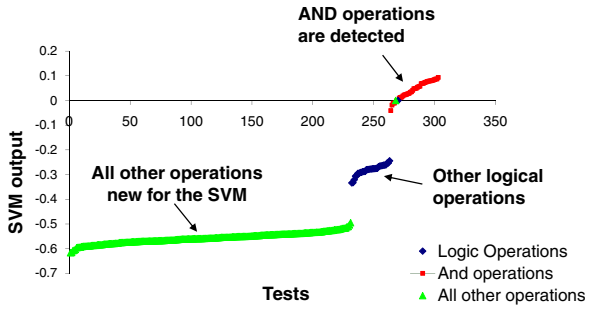

Figure 8: Using ALU to illustrate the result of support vector learning

in the figure, the AND operation tests from the additional tests all show up with high $S()$ values. AND operation shares certain input combinations with the other logical operation. As a result, we see that other logic operation tests have medium $S()$ values. Then at the bottom, there are other tests whose characteristics are quite different from the AND operation (arithmetic operations, etc.). Their $S()$ values are generally low. This simple example demonstrates that indeed if a group of tests are similar, support vector analysis is able to derive a good model to represent them.

5.2 Test filtering

To demonstrate how a verification engineer might utilize support vector analysis to verify the ALU block, we develop a simple scenario for the experiment. In this scenario our objective is to exercise as many unique operations and internal flag combinations as possible.We select 3-consecutive ALU operations and their corresponding internal flags as our targets. We use 6 different operations and 2 flags that are related to the overflow and zero conditions of the internal signals. For example, a 3-cycle test may result in operation and internal flag combination {(AND,No Overflow,Not Zero), (ADD,No Overflow,Zero), (XOR,Overflow,Not Zero)}. The target coverage metric is the state coverage between these points over 3 cycles for a total of 24^3 possible states.

Constrained-random verification enables us to use bias values to adjust our test generation scheme by giving higher priority to certain combinations that might help to achieve the verification objective. For this scenario, since we are interested in generating zero and overflow conditions, we want our word-level inputs to take the value zero or a high value more frequently. In constrained-random verification this can be achieved by increasing the possibility of inputs taking those values. The corresponding adjustment for support vector analysis can be adding new Boolean variables for every word-level input in our design. These variables are only set to 1 if their corresponding input was zero or higher than a certain value. These variables can then be weighted differently to adjust their importance. By doing so we incorporate the importance of these values similar to the way we would with bias values. Using these additional variables then lead to a better kernel function and ultimately a more desirable support vector model.

To test this approach, we randomly generate 2000 3-cycle tests and train a support vector model $S()$. Then we simulate two separate test-benches, the first test-bench uses $S()$ to filter out tests that are potentially redundant(inliers) and the second test-bench is the original one without the filtering. The filtered tests are discarded without being simulated to save simulation time. Figure 9 shows the state coverage results achieved by these two test-benches. Filtering removes a large number of redundant tests and achieves very similar final coverage result with far less simulation effort. For this experiments, simulating the initial 2000 cycles took 32 seconds while the SV model generation took 0.26 seconds and the model had 164 support vectors. It should be noted that the model contains much fewer tests than the initial 2000 tests used for training the

model. This indicates redundancy in the initial test set.

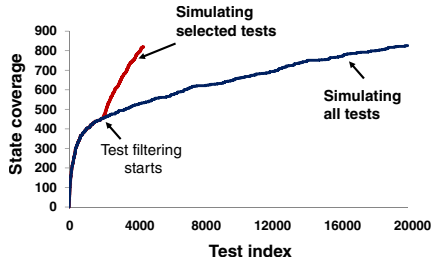

Figure 9: Effect of test filtering on the verification efficiency

5.3 Test selection

For this application we use a constrained random test-bench to produce 30000 tests of length 3 for the execution unit. We take the first 5000 tests to train a support vector model. We then order the remaining 25000 tests in two ways: (1) Apply the outliers first, followed by the rest of the tests (2) Apply the inliers first, followed by the rest of the tests. We also simulate the original 25000 tests without changing their ordering. The simulation results are plotted in Figure 10. By the fact that the coverage curve from the outliers stay above the curve from the original tests, and the coverage curve from the inliers stay below, the effect of using $S()$ can clearly be observed, which implies that $S()$ has indeed learned some characteristics of the first 5000 tests.

For this experiment, we use the Gaussian kernel with two simple types of domain knowledge. For a word-level input, say a 32-bit input, we give a much higher weight to the 5 most significant bits and the 5 least significant bits. Moreover, if we know that a set of inputs should be one-hot, then we make all the tests falling into the non-one-hot space to be considered very similar, i.e. the tests satisfying the one-hot constraint are more dissimilar. This is to collapse the non-one-hot space so that tests are very unlikely to be selected from that space.

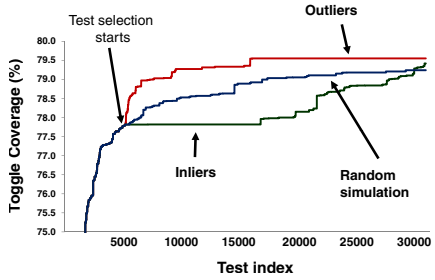

Figure 10: Result to show that outlier tests are more effective

6. TEST SET SELECTION

Suppose we are given a collection of test sets $\{T_1, \ldots, T_j\}$. After T_1 is simulated, suppose we want to find among $T_2, \ldots, T_j$ which one can provide the highest additional coverage. This is a test set selection problem. This problem is different from the test selection problem because each set is selected as a whole. Hence, this problem require us to compare the similarity between two test sets as a single measurement.

Suppose by learning from all tests in T_1, we obtain a support vector model $S()$. Our strategy is use $S()$ to identify the number of outliers in the remaining sets. We use this number to rank the sets. For the experiment, we select 10 unit-level regression test sequences for the execution unit. After learning from one of them, we use the model to find the number of outlier tests in the remaining nine. We then simulate the one used in the learning, followed by each one of the nine test sets. These nine simulation runs produce nine coverage gain values by the nine sets. Figure 11 correlates the outlier number to the coverage value.

We see that the correlation is good for 7 out of the 9 sets. The test set with the highest number of outliers also has the highest coverage value. However, there are two sets with small number of outliers but can provide high coverage increase. As we examine these two sets more carefully, we found that they are for injecting recoverable errors to the data so that the error correction logic can take effect. Some of the input values on those tests contain parity errors and those errors trigger certain part of the design that has not been activated before. Because we did not have this knowledge when running the experiment, this domain knowledge is not incorporated into the kernel. Hence, the support vector analysis failed to detect them as special tests that should be considered as very dissimilar to all other tests.

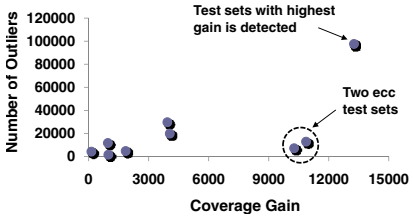

Figure 11: Outlier number correlates to coverage gain

7. CONCLUSION

This paper proposes a general framework, based on kernel-based support vector analysis, to select functional tests for improving simulation efficiency. We describe how domain knowledge can be put into a kernel function to allow the kernel-induced similarity metric space to correlate better with the actual verification coverage space. With limited domain knowledge, we show the effectiveness and potential of our approach through various applications like test filtering and test set selection. Experiments were done using OpenSPARC units that are realistic and complex designs. We plan to extend the experiments to other units where temporal correlation among inputs are critical for exercising the unit. An example is the instruction fetch unit where results will be reported in the future.

8. REFERENCES

[1] S. Tasiran, K. Keutzer, “Coverage Metrics for Functional Validation of Hardware Designs,” IEEE D&T, vol. 18, no. 4, 2001, pp. 36-45.

[2] J. Yuan, C. Pixley, A. Aziz and K. Albin. “A Framework for Constrained Functional Verification," Proc. 2003 *International conference on Computer-aided design* , 2003, pp. 142.

[3] Gilly Nativ, Steven Mittermaier, Shmuel Ur, Avi Ziv. “Cost evaluation of coverage directed test generation for the IBM mainframe," Proc. 2001 *International Test Conference*, 2001, pp. 793-802.

[4] Nina Saxena, Jacob A. Abraham, Avijit Saha. “Causality based generation of directed test cases," Proc. 2000 *Asia and South Pacific Design Automation Conference*, 2000, pp. 503-508.

[5] P. Mishra, N. D. Dutt, "Functional Coverage Driven Test Generation for Validation of Pipelined Processors", in *Proc. DATE*, 2005.

[6] Vladimir N. Vapnik. The Nature of Statistical Learning Theory. 2nd edition, Springer 1999.

[7] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analaysis*, Cambridge University Press, 2004.

[8] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, R. Smeets, “A study in coverage-driven test generation," Proc. 1999 *Design Automation Conference*, 1999, pp. 970 - 975.

[9] O. Lachish, E. Marcus, S. Ur, and A. Ziv, "Hole analysis for functional coverage data", in *Proc. DAC*, 2002, pp. 807-812.

[10] H. Azatchi, L. Fournier, A. Ziv, K. Zohar, "Advanced Analysis Techniques for Cross-Product Coverage," in Proc. HLDVT, 2005.

[11] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfstahl, “Coverage-Directed Test Generation Using Symbolic Techniques," *FMCAD*, 1996, pp. 143-158.

[12] OpenSPARC at http://www.sun.com/processors/opensparc/