# Probabilistic Regression Suites for Functional Verification

Shai Fine
fshai@il.ibm.com

Shmuel Ur
ur@il.ibm.com

Avi Ziv
aziv@il.ibm.com

IBM Research Laboratory in Haifa
Haifa, 31905, Israel

## ABSTRACT

Random test generators are often used to create regression suites on-the-fly. Regression suites are commonly generated by choosing several specifications and generating a number of tests from each one, without reasoning which specification should be used and how many tests should be generated from each specification. This paper describes a technique for building high quality random regression suites. The proposed technique uses information about the probability of each test specification covering each coverage task. This probability is used, in turn, to determine which test specifications should be included in the regression suite and how many tests should be generated from each specification. Experimental results show that this practical technique can be used to improve the quality, and reduce the cost, of regression suites. Moreover, it enables better informed decisions regarding the size and distribution of the regression suites, and the risk involved.

## Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification

## General Terms

Verification, Measurement, Algorithms, Experimentation

## Keywords

Functional Verification, Coverage Analysis, Regression Suite

## 1. INTRODUCTION

Functional verification is widely acknowledged as the bottleneck in the hardware design cycle [4]. To date, up to 70% of the design development time and resources are spent on functional verification. The investment in expert time and computer resources is huge, as is the cost of delivering faulty products [3].

In current industrial practice, simulation-based verification (or dynamic verification) is the main functional verification vehicle for large and complex designs. With dynamic verification, the verification is done by generating a massive amount of tests using random test generators [1, 2, 10], simulating the tests on the design, and checking that the design behaves according to its specification. Coverage [9] is often used to monitor the progress of the verification process and point to areas in the design that have not been properly tested.

Test generators are not quite as common in software testing. However, noise generators of various kinds, which impact the execution of tests, are common. For example, tools are used to simulate network traffic, change the apparent speed of the network, affect response time, and cause many other changes that stress the system under testing. In the testing of multi-threaded or parallel applications, tools, which are often referred to as noise makers, are used to change the apparent behavior of the scheduler [6], and coverage is measured to assess the quality of the noise. The coverage may be inspected on the application code, but is most often collected on system parameters such as number and length of peak traffic and throughput.

Regression testing [11, 4] plays an important part in dynamic verification. In regression testing, a set of tests, known as a *regression suite*, is simulated periodically and after major changes in the design or its environment, in order to check that no new bugs were introduced. A regression suite must, on one hand, be comprehensive so that it can discover bugs introduced, and on the other hand, be small so that it can be economically run many times. Tests are added to regression suites for many different reasons. For example, tests that led to the discovery of hard-to-find bugs are often included in regression suites. Another, complementary, approach for constructing regression suites is to build suites that yield high coverage. That is, suites that produces similar coverage to the one attained by the entire verification effort, conducted so far.

One approach for creating high coverage regression suites is to find the smallest set of tests, out of the tests that have been executed so far, that achieves the same coverage as the entire set of tests. This is an instance of the *set cover* problem, which is known to be NP-Complete [8]. However, an efficient and simple greedy algorithm can provide a good approximation of the optimal solution. An on-the-fly version of the algorithm produces good results for very large sets of tests [5].

Regression suites that are built from a set of predetermined tests have several inherent problems. First, they are sensitive to changes in the design and its environment. Changes in the design may affect the behavior of the tests and lead to areas that are not covered by the regression suite. In addition, tests that were previously used are less likely to find bugs than those that were not yet tried. Finally, the maintenance cost of the suite is high because every test in the suite has to be automated and maintained.


*DAC'04*, June 7–11, 2004, San Diego, California, USA.

In an environment where random-based test generators are available, these problems can be overcome using *random regression suites*. A regression suite is comprised of a set of test specifications. Whenever regression is done, random test generators are used to generate tests out of these specifications. On stable designs, random regression suites are less accurate with respect to coverage than the fixed suites, because tests generated from the same specification can cover different tasks. On the other hand, these suites are less sensitive to changes in the design or its environment, and contain new and different tests each time they are run. The test suites themselves do not have to be maintained, just the test generator, which is maintained for other reasons, therefore, the savings are substantial. As a result, random regression suites are often preferred over maintaining a suite of regression tests.

Creating regression suites on-the-fly using test generators is a common practice in hardware verification and software testing. The method for generating random regression suites is to choose a few specifications and generate a number of tests from each specification. There is no process for reasoning about which specification should be used and how many tests should be used from each specification. Consequently the quality of the regression tests generated is totally haphazard.

An important source of information that can be used to create efficient random regression suites is the probability of each test specification to cover each of the coverage tasks. This information can come from past executions of tests that are generated from the specification files. Another source of information can be a Coverage Directed Generation engine [7], which provides estimates of these probabilities.

Given a set of reliable probability estimates, we show how the construction of efficient random regression suites can be formalized as an optimization problem. We deal with two variants of the problem. First, we show how to construct a regression suite that uses the minimal number of tests required to achieve a specific coverage goal. Then, we show how to create a regression suite that maximizes coverage when a fixed number of tests are used.

Before it is executed, the coverage properties of a given regression suite are probabilistic; therefore, there are many possible ways to measure the quality of a given suite. One possible measure is the average number of tasks covered by the suite. This measure is simply the sum of the probabilities of covering each coverage task. This measure helps us get the maximum coverage out of the suite. Another measure is the smallest coverage probability of all tasks. Maximizing this probability helps us focus on the harder regions to cover, and in turn, ensures that all the areas in the design are covered by the regression suite.

In general, the various optimization problems for random regression suites are hard to solve. Essentially these are nonlinear integer programming problems, some of which are constructed of many constraints. We thus provide some methods to ease the computation burden via linear relaxations, some well known approximation techniques, and simple greedy procedures.

Whenever possible, we devise a "soft" optimization problem, which tolerates controlled violations of some of the constraints, thus leading to an enhanced regression suite, while keeping the computation burden tolerable. For example, when the goal is to cover each task with a predefined probability, using as small a number of tests as possible, it may be beneficial to allow some tasks to have a lower probability of being covered, provided this significantly reduces the number of required tests. This relaxation can be easily formulated into the optimization problem of probabilistic regression suites, and often simplifies the solution to the problem.

The rest of the paper is organized as follows. In Sections 2 and 3 we show how to formalize the construction of random regression suites as optimization problems and describe the methods we use to simplify these problems. In Section 4 we provide some experimental results. Section 5 concludes with a few summarizing remarks and leads for future study.

## 2. CONSTRUCTING PROBABILISTIC REGRESSION SUITES

Suppose, for example, we conducted massive simulations with many different parameter sets (specification files), achieved satisfactory coverage, and would like to maintain only the sets of parameters that are most important. That is, from a large repository of parameter sets we wish to select a subset that is as small as possible, yet capable of producing similar coverage. We allow the use of the same set of parameters several times, so in a sense we are not just looking for an efficient and reliable selection technique—we would like to pair it with an activation policy that will specify how many times to activate simulations using each of the selected parameter sets. Obviously, we would like to minimize the total number of simulations while achieving a similar coverage percentage.

The following terminology and notations are used throughout this sequel:

- Denote $\mathbf{t} = \{t_1, \ldots, t_n\}$ the set of tasks to be covered.
- Denote $\mathbf{s} = \{s_1, \ldots, s_k\}$ the sets of parameters that are allowed to be used in the simulation runs (one set per simulation run (*i.e.,* no switching of sets or mixing of individual parameters).
- Denote $P_j^i$ the probability of covering task $t_j$ in a simulation that uses parameter set (test specification) $s_i$. We make the simplifying assumption that $P_j^i$ are statistically independent.
- *We assume these statistics are reliable.* This assumption is well justified by the motivating scenario for constructing a regression suite based on data obtained by massive simulations and, more accurately, many simulations per parameter set.
- Denote $\mathbf{w} = \{w_1, \ldots, w_k\}$ the activation policy, such that $w_i \in \mathbf{N}$ is an integer specifying how many times a simulation, using the parameter set $s_i$, must be activated. We also denote $W = \sum w_i$ the total number of simulation runs derived by using policy $\mathbf{w}$. We note in passing that by our independence assumption, the order of activating the simulations based on a given policy is insignificant, and in fact, most often they will run in parallel.

Given a policy $\mathbf{w}$, the probability of covering a task $t_j$ is represented by

$$P_j = 1 - \prod_i \left(1 - P_j^i\right)^{w_i} \tag{1}$$

and since the event of covering task $t_j$ is Bernoulli, $P_j = E(t_j)$ is the expected coverage of task $t_j$. The construction of a random regression suite can thus be expressed by the next optimization problem:

DEFINITION 2.1. **Probabilistic Regression Suite** *Find the policy* $\mathbf{w}$*, which minimizes the number of simulation runs and with*

*high probability provides a desired coverage*[1],

$$\begin{array}{ll} \min_{\mathbf{w}} & \sum w_i \\ s.t.\ \forall j & P_j = 1 - \prod_i \left(1 - P_j^i\right)^{w_i} \geq Ecnst \\ \forall i & \mathbf{N} \ni w_i \geq 0 \end{array}$$

This is an integer programming (IP) problem which is difficult to solve. We thus suggested a relaxation to a linear programming (LP) problem. However, the obtained solution provides a "fractional" policy, $\mathbf{w}$, where the $w_i$'s are real numbers, that must be discretized to construct the final policy.

By taking the log of both sides, the constraints in the problem statement defined in Definition 2.1 can be re-written as linear constraints

$$\sum_i w_i \cdot g_{ij} \leq \log(1 - Ecnst) \tag{2}$$

where $g_{ij} = \log\left(1 - P_j^i\right)$

The resulting optimization problem is linear[2]

$$\begin{array}{ll} \min_{\mathbf{w}} & \sum w_i \\ \text{s.t. } \forall j & \sum_i w_i \cdot g_{ij} \leq \log(1 + -Ecnst) \\ \forall i & w_i \geq 0 \end{array}$$

## 2.1 Soft Probabilistic Regression Suite

Even with the relaxation techniques suggested so far, the previous optimization problem can be impractical, either because the feasibility region is empty, or some of the $P_j^i$ probabilities are too small, and therefore the resulting policy specifies the use of many simulations in order to get a complete coverage. We handle these difficulties with the introduction of a "soft" formulation, that permits that some of the constraints be violated; for each violated constraint, we "charge" the objective function with the violation magnitude times the cost $C$. This is done using an additional set of non-negative slack variables, $\xi_i \geq 0$, a function

$$F_\alpha = \sum_i \xi_i^\alpha \tag{3}$$

with parameter $\alpha > 0$, where in the current formulation we chose $\alpha = 1$; and an additional cost parameter $C$, which controls the trade-off between complexity (small policies) and accuracy (constraints violation). The resulting optimization problem follows:

DEFINITION 2.2. **Soft Probabilistic Regression Suite** *Find the policy* $\mathbf{w}$ *which minimizes the number of simulation runs and with high probability provides a coverage that permits violation of the lower bounds for task coverage*

$$\begin{array}{ll} \min_{\mathbf{w}} & \sum_i w_i + C \sum_j \xi_j \\ s.t.\ \forall j & \sum_i w_i \cdot g_{ij} - \xi_j \leq \log(1 - Ecnst) \\ \forall i & w_i \geq 0 \\ \forall j & \xi_j \geq 0 \end{array}$$

Thus, a task $t_j$ for which $\xi_j > 0$, is not guaranteed to have an expected coverage that meets the predefined constraint $E(t_j) \geq Ecnst$.

[1]Note, we are not targeting a specific coverage distribution. However, by careful selection of a different lower bound to constrain the coverage probability of every task, and with the addition of upper bounds, we can actually design a desired coverage distribution, provided that the optimization problem remains feasible.

[2]To avoid numerical instability (when $P_j^i \to 1$) we use the following simple approximation

$$\log(1 - x) \approx \log(1 + \varepsilon - x) - \log(1 + \varepsilon) \approx \log(1 + \varepsilon - x) - \varepsilon$$

# 3. CONSTRUCTING PROBABILISTIC REGRESSION SUITES WITH LIMITED RESOURCES

A slightly different, yet very common, scenario is the requirement to construct the best possible regression suite, using a limited amount of resources. We identify resources with simulation runs, hence the term "limited resources" translates to an upper bound on the total number of simulations permitted (for example, due to limitations of the batch scheduler at the site). However, resources may translate to other measurable quantities related, for example, to the length of a simulation (number of cycles), memory consumption, etc. Moreover, the constraints for resource usage may very well be defined per parameter set, instead of a global restriction over the total resource consumption.

There's no definite meaning to the term "best" possible regression suite. The next interpretation to this notion focuses on the expected coverage probability.

DEFINITION 3.1. **Expected Coverage Probability with Limited Resources** *Given a bound on the total number of simulations permitted, $W$, and a bound on the cost of the resource consumption $C$, find the policy* $\mathbf{w}$*, which maximizes the expected coverage probability*

$$\begin{array}{ll} \max_{\mathbf{w}} & \sum_j \left[1 - \prod_i \left(1 - P_j^i\right)^{w_i}\right] \\ s.t. & \sum_i w_i \leq W \\ & \sum_i c_i w_i \leq C \\ \forall i & w_i \geq 0 \end{array}$$

*where $c_i$ is the cost of the overall resource consumption while using the parameter set $s_i$.*

The above problem is nonlinear and can be solved using standard optimization techniques, though the solution will probably be just an approximation of the true objective. However, an incremental greedy technique can be devised, exploiting the next simple observation

$$\max_j \prod_i (1 - P_j^i)^{w_i} = \max_j (1 - P_j^k) \prod_i (1 - P_j^i)^{w_i^{old}} \tag{4}$$

where $\mathbf{w}^{old}$ denotes the policy before the last increment, and without loss of generality parameter set $k$ was selected for the increment, *i.e.*, $w_k = w_k^{old} + 1$. The incremental greedy algorithm thus starts from an initial guess $\mathbf{w_0}$, (either given by the user or set to all zeros). At each step it increases by 1 the $w_k$ that minimizes

$$\sum_j \prod_i (1 - P_j^i)^{w_i} \tag{5}$$

and thus maximizes the objective function.

The next definition represents a different perspective on the quality of coverage attained with limited resources, which focus on the least probable (or most difficult) tasks to cover. This formulation can be solved via similar incremental greedy techniques.

DEFINITION 3.2. **Least Probable Coverage Task with Limited Resources** *Given a bound on the total number of simulations, $W$, and a bound on the cost of the resource consumption $C$, find the policy* $\mathbf{w}$ *that maximizes the probability of the least probable task to cover,*

$$\begin{array}{ll} \max_{\mathbf{w}} & \min_j \left[1 - \prod_i \left(1 - P_j^i\right)^{w_i}\right] \\ s.t. & \sum_i w_i \leq W \\ & \sum_i c_i w_i \leq C \\ \forall i & w_i \geq 0 \end{array}$$

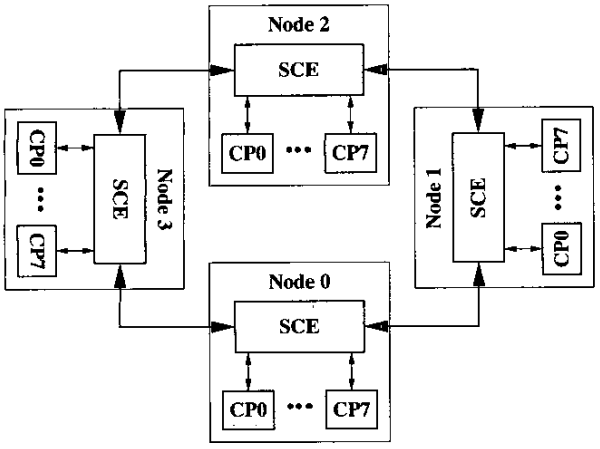


**Figure 1: Structure of the SCE simulation environment of an IBM z-series system**

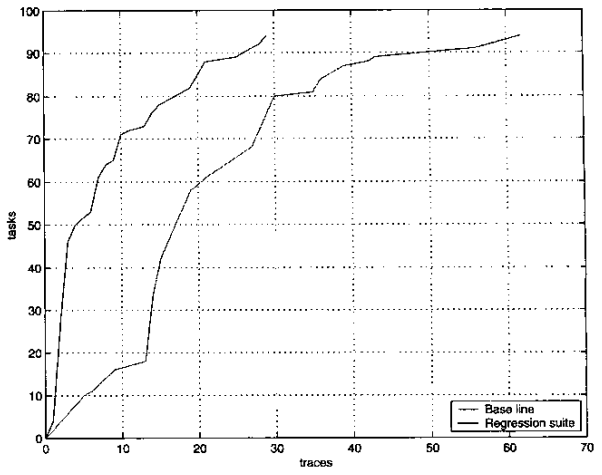


**Figure 2: Coverage progress of the unrecoverable error (UE) subset of the SCE**

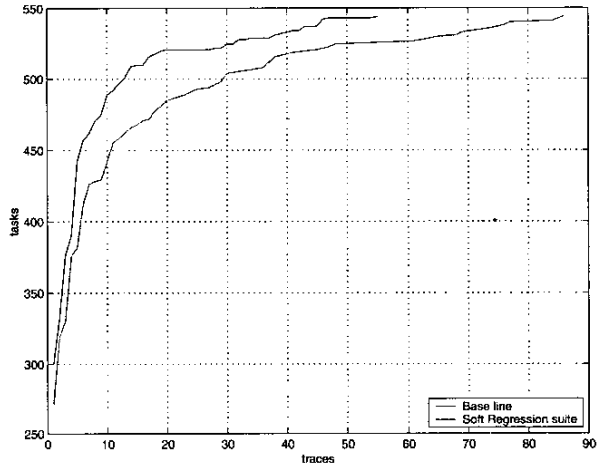


**Figure 3: Coverage progress of the SCE model**

*where $c_i$ is the cost of the overall resource consumption while using the parameter set $s_i$.*

# 4. EXPERIMENTAL RESULTS

To demonstrate the feasibility and applicability of the suggested formalisms for the construction of random regression suites, we conducted several experiments in both hardware verification and software testing environments, using real-life applications and coverage models.

## 4.1 Regression Suite for Hardware Verification with Minimal Number of Simulations

The goal of the next two experiments was to construct, based on the formalism developed in Section 2, random regression suites paired with activation policies, which minimize the number of simulation runs.

We conducted these experiments on subsets of a coverage model used in the verification of the *Storage Control Element* (SCE) of an IBM z-series system, as shown in Figure 1. The environment and coverage model used in the experiments are similar to those used in [7]. The environment contains four nodes that are connected in a ring. Each node is comprised of a local store, eight CPUs (CP0 – CP7), and an SCE that handles commands from the CPUs. Each CPU consists of two cores that independently generate commands to the SCE. Each SCE handles incoming commands using two internal pipelines. When the SCE finishes handling a command, it sends a response to the commanding CPU.

The coverage model consists of all the possible transactions between the CPUs and the SCE. It contains six attributes: the core that initiated the command, the pipeline in the SCE that handled it, the command itself, and three attributes that relate to the response. In the first experiment, we concentrated on a subset of the coverage model that deals with unrecoverable errors (UE). The size of the UE space is 98 events, all of which are relatively hard to cover.

Our goal was to construct a regression suite by selecting, from a repository of test specifications designed to cover the UE space, a much smaller subset that will produce a similar coverage. The repository we used consisted of 98 sets of test specifications, each of which was designed to provide the best possible configuration that will cover one of the events. The inherent randomness in the test generation mechanism enables covering other events as well, during a simulation run; otherwise, there's no hope of constructing a regression suite smaller than the whole repository.

The regression suite and activating policy were generated using the linear relaxation of the formalism depicted in Definition 2.1, using a lower bound of $1/2$ for the expected coverage for each task. The resulting regression suite was constructed using only three parameter sets, and the activating policy, $[8,5,1]$, specified only 14 simulations. Indeed, as depicted in Figure 2, approximately two rounds of the activating policy (29 simulations) suffice to cover 94 events (96%). This is compared with the common practice of activating each of the parameter sets in the repository, one at a time. It took 61 simulations to reach the same coverage, cf. Figure 2.

The second experiment targeted the entire SCE coverage model, which has 564 tasks, most of them are not too hard to cover. The repository that we used consisted of 126 test specification, that were generated during a Coverage Directed Generation (CDG) process [7]. The regression suite and activating policy were generated using the soft probabilistic regression formalism depicted in Defi-

nition 2.2, using a lower bound of 1/2 for the expected coverage for each task. The resulting regression suite was constructed using 11 parameter sets, and the activating policy specified 32 simulations. Indeed, as depicted in Figure 3, after less than two rounds of the activating policy (55 simulations), 554 events (98%) were covered. This is compared with the activation of every parameter set in the repository, one at a time. It took 86 simulations to reach the same coverage, cf. Figure 3.

## 4.2 Regression Suite for Software Testing with a limited Number of Test Executions

In many cases, the amount of resources and time allocated for execution of a regression suite is limited. For example, a nightly regression suite cannot last more than six hours, during which only 1000 tests can be executed. The second experiment examines the formalism developed in Section 3 for building a random regression suite while limiting the resources available for constructing the regression. A natural application for this formalism is software testing (*e.g.*, in a multi-threaded domain), where a set of predefined test heuristics is used to provide coverage, while limiting the total number of test executions.

A test in the multi-threaded domain is a combination of inputs and interleaving, where interleaving is the relative order in which the threads were executed. Running the same inputs twice may result in different outcomes, either by design or due to race conditions that exist in the code. Re-executing the same suite of tests may result in other tasks being executed. Therefore, a regression suite does not have the same meaning as in the sequential domain.

ConTest [6] is a tool for generating different interleaving for the purpose of revealing concurrent faults. ConTest takes a heuristic approach of seeding the program with instrumentation in concurrently significant locations. At run-time, we make heuristically, possibly coverage-based, decisions regarding which noise (sleep(), yield(), or priority()) to activate at each interleaving. ConTest dramatically increases the probability of finding typical concurrent faults in Java programs. The probability of observing the concurrent faults without the seeded delays is very low.

In a typical ConTest user scenario, a given functional test $t$ is run (without human intervention) against $P$, the program under test. This is done repeatedly until a coverage target is achieved. The common practice in unit, function, and system tests (but not in load testing), is to execute the test once, unless a fault is found. The process is depicted in Figure 4. Load testing (*i.e.*, testing the application under a real or simulated workload), increases the likelihood of some interleaving that would be unlikely under light load. However, load testing is not systematic, it is expensive, and can only be performed at the very end of the testing cycle.

The process of re-executing tests using ConTest is depicted in Figure 5. Each time the functional test $t$ is run as a result of the seeding technique, ConTest produces a potentially different interleaving. During the execution of the functional test $t$, coverage information is produced and collected. For example, as coverage information, we might collect data about whether or not a context switch occurred while executing a program method. We run the test multiple times, and with different heuristics, in order to try and reveal a bug if one can be revealed by the tests.

During the testing process depicted above, we collect the statistics needed to construct probabilistic regression suites. Prior to this work, our practice was to use a predefined random mix of heuristics. No tuning was done for specific applications and test cases. We will show the benefit of such tuning.

The program tested in the experiment is a crawling engine for a large web product. For the experiment, we used 13 different heuristics as the test specifications. The heuristics differ in the probability that noise is created for each instrumentation point and in the noise strength. For example, if the noise is `yield()`, the number of times it is executed depends on the strength. Low strength means that yield is executed just a few times and high strength means that yield is executed many times. For `sleep()` the strength parameter impacts the length of the sleep. Some heuristics have additional features such as limiting the location of the noise to variables shared between threads or having additional types of noise primitives.

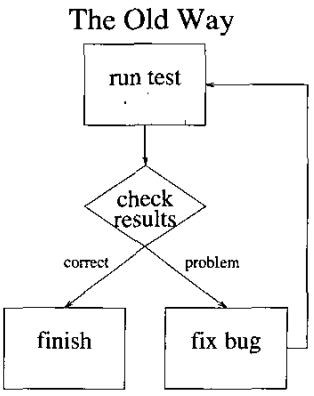


**Figure 4: Each test executed once**

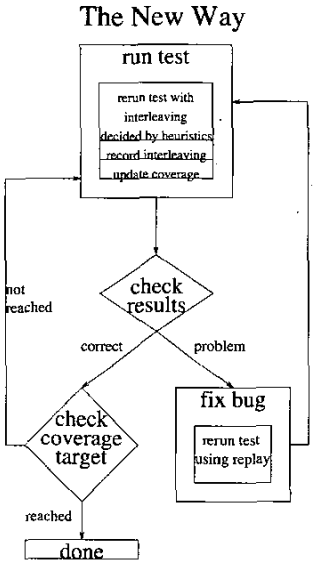


**Figure 5: Executing until reaching a coverage target**

Given the statistics collected for the 13 heuristics, we constructed policies designed to maximize the coverage of the 10,000 events, using no more than 250 and 1000 test runs. To this end, we applied the formalism depicted in Definition 3.1, while using the greedy technique to get the solutions. The policy designed to yield the best coverage with only 250 test runs was constructed of only two (out of the 13) heuristics. The policy for the 1000 test runs was constructed of four heuristics, two of which are very dominant (should be used roughly 83% of the time). Table 1 presents the the total

| | Num. Events Covered | |
|---|---|---|
| | 250 | 1000 |
| Uniform Policy | 1449 | 1988 |
| Best Pure Heuristic | 1699 | 2258 |
| Greedy Policy | 1723 | 2429 |

**Table 1: Total Coverage using ConTest for 250 and 1000 test runs**

number of events covered, using 250 and 1000 test runs. Note in passing, that by using every heuristic in 1000 test runs and combining the results (*i.e.,* a total of 13,000 test runs), only 4338 events were covered. Namely 56.62% of the events are very hard to cover with this set of heuristics using such a limited amount of test runs. The first row in the table present the coverage obtained by the best single heuristic throughout the entire 250 (1000) test runs. The second row represents a policy uniformly distributed over the 13 heuristics, and the last row presents the results of the policies generated by the greedy algorithm. These policies proved to be favorable in both experiments.

# 5. CONCLUSIONS AND FUTURE WORK

Constructing regression suites on-the-fly using test generators is a common practice in hardware verification and software testing. In this paper, we proposed a method to formulate the construction of random regression suites as optimization problems. The solution to the optimization problem provides information on which specification should be used and how many tests should be generated from each specification. This formalism allows us to design random regression suites that either maximize coverage with a limited number of tests, or minimize the number of tests for a given coverage goal. In all cases, the quality of the regression suite becomes a known quality and informed decisions regarding the size and distribution of the regression suites can be made.

Experimental results show that our method is economically rewarding. We showed that the regression suites created are expected to be much better than those created at random. This is true for both the limited resource and specific utilities scenarios. Specifically, we showed that when there are several specifications, none of which dominates the other, a smart selection of the amount of resources dedicated to the use of each is much better that using all the resources on the best parameter source or distributing the resources evenly.

We plan to expand the work described in this paper is several directions. First, we plan to investigate the use of the proposed technique to guide the entire functional verification process. Currently, test specifications for the test generators are usually chosen in an ad-hoc manner. We estimate that selecting the parameters that are likely to cover more tasks using the methods described in this paper, can either markedly reduce the simulation cost or improve the quality.

We also investigate the use of hybrid schemas that combine the building of probabilistic regression suites with snapshots of the current coverage state. This will serve to define checkpoints at which coverage is measured and re-generate the regression suite at these checkpoints, only for the non-covered tasks. As the cost of regenerating the suite is small compared to the cost of simulation, we expect that an optimal solution will have several such checkpoints.

Special attention should be given to tasks that are covered with a very low probability. Under some of the cost functions, these tasks greatly impact the specification selection toward specifications that are better than others in covering these tasks, but may not be very good specifications in general. One possible solution to this issue is not to try to cover these tasks using a random regression suite. Instead, keep tests that cover such tasks along with other tests that are kept in a deterministic regression suite.

# 6. REFERENCES


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

[2] A. Ahi, G. Burroughs, A. Gore, S. LaMar, C. Linand, and A. Wieman. Design verification of the HP9000 series 700 PA-RISC workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[3] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[5] E. Buchnik and S. Ur. Compacting regression-suites on-the-fly. In *Proceedings of the 4th Asia Pacific Software Engineering Conference*, December 1997.

[6] O. Edelstein, E. Farchi, Y. N. G. Ratzaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.

[7] S. Fine and A. Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *Proceedings of the 40th Design Automation Conference*, pages 286–291, June 2003.

[8] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

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

[10] A. Hosseini, D. Mavroidis, and P. Konas. Code generation and analysis for the functional verification of microprocessors. In *Proceedings of the 33rd Design Automation Conference*, pages 305–310, June 1996.

[11] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.