H-0251 (H0406-025) June 19, 2004
Computer Science

IBM Research Report

Future Aware Algorithms for Probabilistic Regression Suites

Shady Copty, Shai Fine, Shmuel Ur, Avi Ziv
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Future Aware Algorithms for Probabilistic Regression Suites

Shady Copty
shady@il.ibm.com

Shai Fine
fshai@il.ibm.com

Shmuel Ur
ur@il.ibm.com

Avi Ziv
aziv@il.ibm.com

IBM Research Laboratory in Haifa
Haifa University Campus
Haifa, 31905
Israel

ABSTRACT

Automated regression suites are essential in developing large applications while maintaining reasonable quality and timetables. The main objection to automation of tests, in addition to the cost of creation and maintenance, is the observation that if you run the exact same test many times it becomes a lot less likely to find bugs. To alleviate those problems, a new regression suite practice, which uses random test generators to create regression suites on-the-fly, is becoming more common. In this regression practice, instead of maintaining tests, regression suites are generated on-the-fly by choosing a several specifications and generating a number of tests from each.

This paper describes techniques for optimizing random generated regression suites. It first shows how the set cover greedy algorithms, commonly used for selecting tests for regression suites, may be adapted to selecting specifications for randomly generated regression suites. It then introduces a new class of greedy algorithms, namely future aware greedy. These algorithms are as computationally efficient and generate more effective regression suites.

General Terms

Verification, Measurement, Algorithms, Experimentation

Keywords

Functional Verification, Coverage Analysis, Regression Suite

1. INTRODUCTION

Regression testing [17, 3, 23, 20, 14, 12, 10, 22, 4, 24] plays an important part in software testing. In regression testing, a set of tests, known as a *regression suite*, is simulated periodically and after major changes in the application or its environment, in order to check that no new bugs were introduced – a very common problem. 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 produce similar coverage to that 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 those that have been executed so far, that achieve 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 [9]. However, in practice, many efficient approaches exist. In addition to finding small regression suites, there is a large body of work on finding the tests that are relevant for the change in the code [23, 16] and on tests case prioritazation which is of importance when only a subset of the regression suite may be executed [12]. Empirical results [10] showed that the test prioritization technique achieves code coverage at a faster rate and more importantly the rate of detecting bugs is increased. It was also shown in [24] that no single regression optimization technique is best for all scenarios. Each has strengths and weaknesses that depend on the use scenario.

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 have not been tried [18]. Finally, the maintenance cost of the suite is high because every test in the suite has to be automated and maintained.

In an environment where random based test generators are available, these problems can be overcome using *random regression suites*. A random regression suite comprises of a set of test specifications. Whenever regression is done, random test generators are used to generate tests out of these specifications. On stable applications, random regression suites are less accurate with respect to measurable criteria such as coverage than the fixed suites, because tests generated from the same specification can cover different tasks. However, these suites are less sensitive to changes in the applications 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 anyway for other reasons. Therefore, the savings are substantial. As a results, 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 when random test generators are used. Test generators are ubiquitous in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA 2004, July 11–14, 2004, Boston, MA, USA.
Copyright 2004 ACM 1-58113-688-9/03/0001 ...$5.00.

hardware [1, 2, 13] and are fairly common in software testing. The most common test generators in software are test data generators, which select new data at random, biased by the specification, for each test [19, 21, 6, 15, 11]. In addition, noise generators of various kinds, which impact the environment in which the tests are executed, 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 [5]. As the test is composed of the input and the noise use, and the noise is biased random using specific parameters, one may think of the noise makers as random test generators.

The common 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 were generated from the specification files. Another source of information can be a coverage directed generation engine [8] that provides estimates of these probabilities.

Given a set of reliable probability estimates, we showed, in a previous paper [7], how the construction of efficient random regression suites can be formalized as an optimization problem. In that paper we dealt with two variants of the problem. First, we showed how to construct the regression suite that uses the minimum number of tests required to achieve a specific coverage goal. Then, we showed how to create a regression suite that maximizes coverage when a fixed number of tests is used. When the selection is on tests and not on properties, this is known as test prioritization.

In this paper, we focus on the problem of creating regression suites that maximize coverage given limited resources. This paper contains two important contributions:

- A new type of future-aware greedy algorithm. The algorithms suggested in [7] as well as all standard greedy algorithms are aware of the past but not of the future. Indeed algorithms that try to predict the future, usually using searches, are not considered greedy. In this paper, we introduce a greedy algorithms that takes statistical expectation of the future into account. Specifically, tasks that are likely to be covered in the future are given less weight when choosing which parameters to use. We show, (without a deep investigation) that this algorithms is better then the greedy algorithm for the classical set cover problem.

- Expansions of the work done in [7] to the case where coverage is used to measure the utility of tests executed so far. This additional information enables us to get better results but requires modification of the algorithms to selection on-the-fly

We show experiments on a number of random regression suite problems. We start with synthetic, simple problems on which we explain the algorithms and motivations. We then demonstrate how the algorithms perform on real world problems. The focus of this paper is on computationally simple yet very efficient algorithms for the random regression suite problem.

The rest of the paper is organized as follows. In Sections 2 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 3 we explain the ideas behind the future aware greedy algorithm. Section 4 describes the difference between regression with and without feedback from coverage. In Section 5, we provide some experimental results. We conclude with a few summarizing remarks and leads for future study.

2. PROBABILISTIC REGRESSION SUITES WITH LIMITED RESOURCES

We started by formulating the problem of constructing a probabilistic regression suite based on statistical estimates (predictors) of the covering performances for the various test specifications. To this end, we set the following terminology and notations: Denote $\mathbf{t} = \{t_1, \ldots, t_n\}$ the set of tasks to be covered. Test specifications are often sets of parameters which govern and bias the generation of tests by a random test generation tool. Thus, we use the term *set* as an abbreviation, and denote $\mathbf{s} = \{s_1, \ldots, s_k\}$ the repository of sets for which statistical coverage predictors exist. We assume that a single test specification is used for a single test generation run, i.e., (dynamic) switching of sets or mixing of individual parameters is not allowed. The probability of covering the task t_j using a test generated based on the set (test specification) s_i is denoted P_j^i. We make the simplifying assumption that P_j^i are statistically independent. We assume that these statistical estimates are reliable, and hence we won't deal in this sequel with issues related to accuracy and confidence of these predictors. The resulting regression suite is represented by the vector $\mathbf{w} = \{w_1, \ldots, w_k\}$ which specifies an activation policy, such that $w_i \in \mathbf{N}$ is an integer specifying how many tests must be generated using the set s_i. We also denote $W = \sum w_i$ the total number of tests derived by the policy $\mathbf{w}$. We note in passing that by our independence assumption, the order of executing the tests 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

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

and since the event of covering a 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 following optimization problem [7]:

DEFINITION 2.1. **Probabilistic Regression Suite** *Find the policy* $\mathbf{w}$*, which minimizes the number of test executions and, with high probability, provides a desired coverage*

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

The formulation at Definition 2.1 doesn't take in consideration any limitations of the resources available for the coverage process. However, practical limitations do exist. They most probably will have a major impact on our ability to carry out a policy, and we should therefore incorporate them in the problem definition. Our motivating scenario is thus the requirement to construct the "best" possible regression suite, while limiting the amount of resource consumption. We identify resources with CPU time, hence the term "limited resource consumption" translates to a bound on the total number of tests executed (for example, due to limitation of the batch scheduler in the site). However, resource may translate to

other measurable quantities such as memory consumption. Moreover, the constraints for resources usage may be defined per set, resources allocated by different sets might be charged differently, and there may be a restriction of the total cost of resources allocated to carry on the coverage task.

There's no definite meaning to the term "best" possible regression suite. In this sequel we adapt an interpretation that focuses on the expected coverage probability as a quality measure, and thus the next problem definition follows.

DEFINITION 2.2. **Expected Coverage Probability with Limited Resources** *Given a bound on the total number of executed tests, W, and a bound on the cost of the resource consumption C, find the policy* **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 problem formulated at Definition 2.2 is a nonlinear IP, which is quite difficult to handle. One possible approach is to apply the annealed approximation

$$\log \sum_j \prod_i (1 - P_j^i)^{w_i} \approx \sum_i w_i \sum_j \log\left(1 - P_j^i\right) \qquad (2)$$

where $g_{ij} = \log\left(1 - P_j^i\right)$. This approximation, in turn, yields the following linear IP

$$\begin{array}{ll} \max_{\mathbf{w}} & \sum_i w_i \sum_j g_{i,j} \\ \text{s.t.} & \sum_i w_i \leq W \\ & \sum_i c_i w_i \leq C \\ \forall i & w_i \geq 0 \end{array}$$

However, the solution is an approximation of the true objective, which heavily depends on the distribution of values of $g_{i,j}$.

A much simpler and more direct approach, suggested at [7], devise an incremental greedy technique by exploiting the next simple observation

$$\max_j \prod_i (1 - P_j^i)^{w_i} = \max_j (1 - P_j^k) \prod_i (1 - P_k^i)^{w_i^{old}} \qquad (3)$$

where $\mathbf{w}^{old}$ denotes the policy before the last increment, and k denotes the set selected for the next increment, i.e. $w_k = w_k^{old} + 1$. The incremental greedy algorithm thus starts from an initial guess $\mathbf{w_0}$, (either provided by the user or set to all zeros). At each step increases by 1 the w_k that minimizes

$$\sum_j \prod_i (1 - P_j^i)^{w_i} \qquad (4)$$

and thus maximizes the objective, since

$$\max_{\mathbf{w}} \sum_j \left[1 - \prod_i \left(1 - P_j^i\right)^{w_i}\right] \equiv \min_{\mathbf{w}} \sum_j \prod_i \left(1 - P_j^i\right)^{w_i} \qquad (5)$$

3. FUTURE-AWARE GREEDY ALGORITHM

The greedy algorithm for constructing probabilistic regression suites with limited resources provides efficient and high-quality regression suites, but these suites are usually not optimal. One reason for the sub-optimality of the greedy algorithm is that it does not consider future steps in the algorithm. Specifically, the greedy algorithm ignores the contribution of the heuristics selected in future steps to the overall coverage. As a result, the greedy algorithm may select a heuristic with a high probability of covering a given task, ignoring the fact that this task will be covered with high probability in the future, even without the selected heuristic.

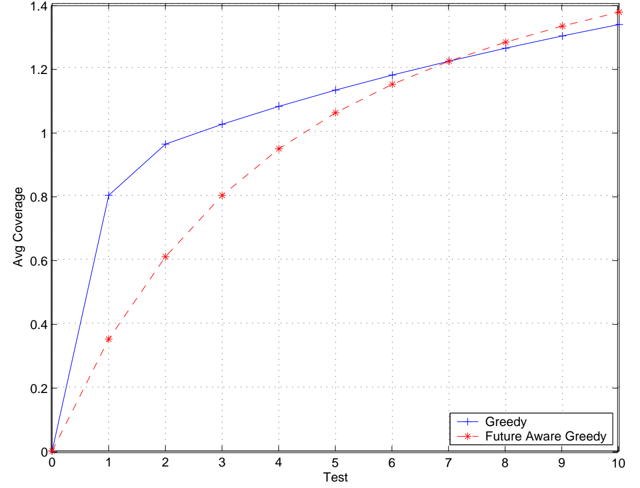

Figure 1: Progress of the greedy and future-aware greedy algorithms

For example, consider a simple case, where the goal is to maximize the coverage of two tasks (T_1 and T_2) with 10 test cases selected from the two heuristics (H_1 and H_2) and the coverage probability matrix shown in Table 1.

	$\mathbf{T_1}$	$\mathbf{T_2}$
$\mathbf{H_1}$	0.80	0.00
$\mathbf{H_2}$	0.30	0.05

Table 1: Coverage probability matrix example

When the greedy algorithm presented in the previous section is used, in the first step of the algorithm, heuristic H_1 is used because of its contribution to the coverage of T_1. For the same reason, the greedy algorithm also selects H_1 in the second step. After the second step, the probability of covering T_1 is high enough (0.96), such that the contribution of H_1 to its coverage in future steps is small. Therefore, the contribution of H_2 to the coverage of T_2 is dominant in the next 8 steps. The resulting regression suite created by the greedy algorithm is $W = \{2, 8\}$, with average coverage of 1.3343. The progress of the average coverage for the greedy algorithm is shown in Figure 1.

The greedy algorithm ignores the fact that at each step the probability of covering T_1 increases regardless of the heuristics used. Even if H_2 is used in all 10 test cases, the probability of covering T_1 is $1 - (1 - 0.3)^{10} = 0.9718$. If this fact is used in the first two steps of the greedy algorithm, the contribution of selecting H_1 becomes much lower, and H_2 becomes the preferred heuristic. The resulting regression suite in this case is $W = \{0, 10\}$, with average coverage of 1.3730 (see the dashed line in Figure 1).

The performance of the greedy algorithm can be improved by considering at each step of the algorithm not only the probability that a task is covered in previous steps of the algorithm, but also the probability that the task will be covered by future steps. This

improvement is possible only if the size of the regression suite or an estimate of it are known in advance. Otherwise, prediction of the future is impossible, because the future may not exist (i.e., we are in the last step). From this point on we assume that the size of the regression suite (and thus the number of steps in the algorithm) is known.

The basic greedy algorithm looks for the heuristic that maximizes the coverage after the $k+i$'th step, given the heuristics used in the previous k steps. That is, in each step the goal is to maximize

$$\arg\max_i \sum_j (1-S_j)P_j^i,$$

where S_j is the probability of covering task j in the previous steps and P_j^i is the probability of covering task T_j using heuristic H_i. The future-aware greedy algorithm replaces this goal function with

$$\arg\max_i \sum_j (1-S_j)P_j^i(1-F_j), \qquad (6)$$

where F_j is an estimation of the probability of covering task T_j in future steps.

The quality of the estimation of F_j affects the quality of the solution provided by the future-aware algorithm. It is easy to show that exact knowledge of F_j leads to an optimal solution. The problem is that exactly calculating F_j is as hard as providing an optimal solution to the probabilistic regression suite. An optimistic estimation of F_j may degrade the quality of the solution, since it may unnecessarily punish good heuristics because of a too optimistic future. In the extreme case, if we use $F_j = 1$, the greedy algorithm is reduced to a random selection of heuristics.

When a good method for estimating F_j is used, the future-aware algorithm should perform better than the greedy algorithm. But if we look at coverage progress as function of the step in the algorithm, the greedy algorithm should perform better than the future-aware greedy in the early steps. This happens because the greedy algorithm tries to maximize the current gain, while the future-aware algorithm looks at a farther horizon. Figure 1 illustrates this. In general, we expect the future-aware algorithm for n steps to perform better than the future-aware algorithm for $m > n$ steps after n steps.

In our experiments (see Section 5), we examined several methods of estimating F_j. These estimation methods differ in their accuracy and computational complexity. But generally, they all improved the quality of the generated regression suites compared to the simple greedy algorithm. The estimation methods we used for F_j were:

- $F_j = 1-(1-M_j)^k$, where k is the number of steps left in the algorithm and M_j is the minimum probability of covering task T_j over all heuristics (i.e., $M_j = \min_i P_j^i$). This estimation is obviously pessimistic and it is useful only if the coverage probability matrix $\{P_j^i\}$ does not contain many zero entries. Otherwise, this estimation is reduced to the simple greedy algorithm.

- Use data from previous executions of the algorithm. This method takes advantage of the fact that in many cases the regression suite is used (and generated) many times. In this case, F_j can be estimated as the average of the probabilities of covering T_j in previous activations of the regression suite generation algorithm.

- When data from previous activations of the future-aware greedy algorithm is not available, we can use data from the greedy algorithm itself to estimate the future. Let H_{s_l} be the heuristic selected at step l of the algorithm, then an estimation of the future probability of covering task T_j after the k step is given by

$$F_j = 1 - \prod_{l=k+1}^{n} (1-P_j^{s_l})$$

- The problem with the previous estimation methods is that they do not take into account the heuristic selected in the current step when estimating the future. To overcome this problem, we can modify the previous estimation to consider the various possibilities for the current step. That is, for each heuristic, we calculate $(1-S_j)P_j^i$ for all the coverage tasks and use this as the starting point for the greedy algorithm. As a result, the output of the greedy algorithm uses the current step, and the estimation of the future is more accurate. The cost of this estimation is that in each step we need to execute the greedy algorithm once for each heuristic instead of only once.

- An algorithm that turns out to combine good performance with excelent results is one in which instead of estimating the future we choose in the future! This is accomplished by running the regular greedy algorithm to completion but instead of choosing the heuristic for the first step we choose the on that was chosen for the last step. The intuition is that we choose the one that is chosen with the most knowledge. We repeat the process for every choice. The number of simple greedy algorithm steps is a square of the naive algorithm. We called this algoritm the *reverse greedy* algorithm.

Note that the future-aware greedy approach is not limited to probabilistic problems. There is a large body of work on using greedy algorithms for the set cover problem [20]. While this is not the focus of this work, we compared the future-aware approach suggested in this paper to greedy algorithms for deterministic set cover. The idea is very similar. Instead of choosing the algorithm that adds the most tasks, we weight the tasks according to the probability that we will see them in the future and choose according to this weight. Experimental results show that the future-aware approach is superior to the greedy approach and achieves full coverage with fewer sets or better coverage with a limited number of sets.

4. REGRESSION SUITES WITH INTERMEDIATE COVERAGE MEASUREMENT

Until now, we assumed that the generation of the regression suites is done off-line. That is, we first create the entire regression suite and then we execute it and measure the coverage it achieves. The quality of the off-line (or static) generation of the regression suite can be improved if we generate the regression suite on the fly. That is, execute parts of the regression suite and measure its coverage while generating the rest of the suite.

In general, finding the optimal regression suite with intermediate coverage measurement is as complex as finding the optimal regression suite without intermediate coverage measurement. Therefore, our generation schemes are based on the greedy and future-aware greedy algorithms presented in the previous sections. In the following discussion, we assume that the generation and execution of parts of the regression suite are done sequentially. That is, we generate part of the regression suite, execute it, measure the coverage, and after the coverage measurement is completed, we continue with the generation of the next part of the suite. Under this assumption, at the end of each step, we can remove all the tasks that are

	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$
$\mathbf{H_1}$	1.00	0.00	0.00
$\mathbf{H_2}$	0.00	1.00	0.00
$\mathbf{H_3}$	0.10	0.10	0.10

Table 2: Coverage probability matrix for regression suite with intermediate measurement example

covered in the step and start generating a "new" suite with fewer tasks and fewer tests. In addition, we assume that in each step a single heuristic is selected and executed. Under this assumption the greedy algorithm selects the heuristic H_i that maximizes the expression $\sum_j P^i_j$, where P^i_j is the probability of covering T_j using H_i if T_j is not covered and 0 if T_j is covered.

The future-aware approach with intermediate measurements is more complex than the future-aware approach without measurements because the algorithm needs to consider the size of the current step (1 in our case) in addition to the size of the entire regression suite. To illustrate this, consider a simple case, where the goal is to maximize the coverage of three tasks (T_1, T_2 and T_3) with three test cases selected from the three heuristics and the coverage probability matrix shown in Table 2.

Both the greedy algorithm and the future-aware greedy algorithm presented in the previous step select H_1 (or H_2) at the first step of the algorithm on their way to generating the optimal solution $\{1,1,1\}$. This selection is not very useful when intermediate measurements are used, since its outcome (after measurement) is known and the measurement does not help to improve the quality of the solution. On the other hand, executing H_3 in the first step can improve the solution, because after the coverage measurement, we know if T_1 (or T_2) is covered and H_1 (or H_2) is no longer needed.

Still, as the experimental results in the next section show, using the first step of the future-aware algorithm with any of the methods of estimating the future and measuring coverage after each test case, improves the quality of the regression suite over the simple greedy algorithm. One possible method to improve this simple algorithm is to consider not only the average contribution to coverage of each heuristic, but also the amount of information added when the heuristic is executed (from the knowledge that is will be executed). For example, we can generate the entire suite (without measurement) and select from it the heuristic that has the highest variance, because the outcome of this heuristic adds the most information. The topic of how to combine the average contribution and the variance is part of our future research in this area.

5. EXPERIMENTAL RESULTS

To demonstrate the feasibility and applicability of the suggested formalisms, we conducted several experiments on regression data collected from both software testing and hardware verification environments, using real-life applications and coverage models.

5.1 Experimental approach

We described two approaches for using our algorithm, in one approach our policy is dynamic in the sense that we measure coverage after each execution and decide on the next activation. The other approach is static, we create an activation policy and execute it. Our measurements have shown that creating a dynamic activation policy with a naive greedy algorithm yields better results than the best static activation policy that our algorithms were able to create (see Table 3 for example) therefore we will discuss each approach separatley.

	Best	Worst
Dynamic Policy	95.6	94.9
Static Policy	93.33	92.8

Table 3: Average coverage for the UE problem with 20 tests

Our primary measurement of an algorithm's quality is its average number of covered tasks, and our secondary measurement is its distribution over hard tasks. We show that the future aware greedy algorithm not only yields a better average but also invests its resources in harder tasks.

5.2 Regression Suite for Software Testing

A test in the multi-threaded domain is a combination of inputs and interleavings, where an interleaving is the relative order in which the threads were executed. Running the same input 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 [5] 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. 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 impact the length of the sleep. Some heuristics have additional features such as limiting the location of the noise to variables that are shared between threads or having additional types of noise primitives. 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.

The tested program in the experiment is a crawling engine for a large web product. For the experiment, we used 18 different heuristics as the test specifications and 9 synchronization events as tasks.

During the testing process, we collected 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 demonstrate the benefit of such tuning.

Given the statistics collected for the 18 heuristics, we constructed dynamic policies designed to maximize the coverage of the nine events, using 10 test runs and five test runs. these policies were executed 100,000 times and the results show the average coverage they yielded.

We show in Figure 2 how the pessimistic future aware greedy, the worst of our future aware heuristics, is able to produce better results than greedy. The graph shows the difference between future aware greedy and greedy. We see how greedy achieves better results at the beginning, while future aware bypasses it at some point in the middle of the whole regression. This is because it can predict which tasks will be covered regardless of its efforts, and so it invests in harder tasks. We also see how in the five test runs the future aware starts to climb earlier and also bypasses the greedy at around the

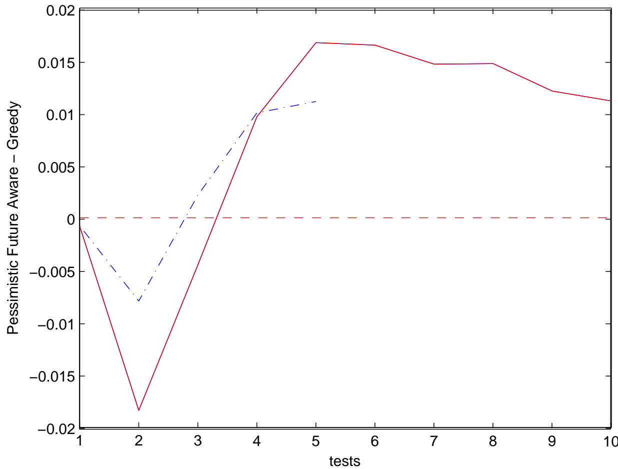

Figure 2: Greedy Vs Pessimistic Future Aware Greedy

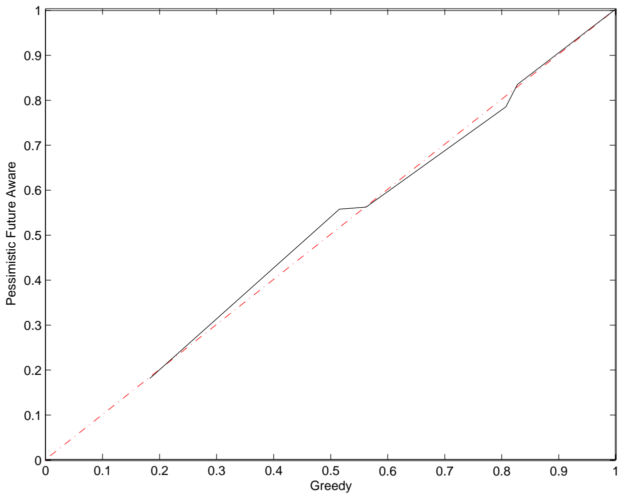

Figure 3: Greedy Vs Pessimistic Future Aware Greedy: Distribution over tasks

third test. Figure 3 shows the probability for future aware greedy to cover a task compared to greedy. This shows how our algorithm invests more resources in harder tasks - tasks with a probability of (0.2 - 0.55) and fewer resources in easier ones.

5.3 Regression Suite for Hardware Verification

The goal of the experiments was to construct random regression suites paired with static activation policies that maximizes the average number of covered tasks

These experiments were conducted 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 4. The environment and coverage model used in the experiments are similar to those used in [8]. 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 generate independently 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.

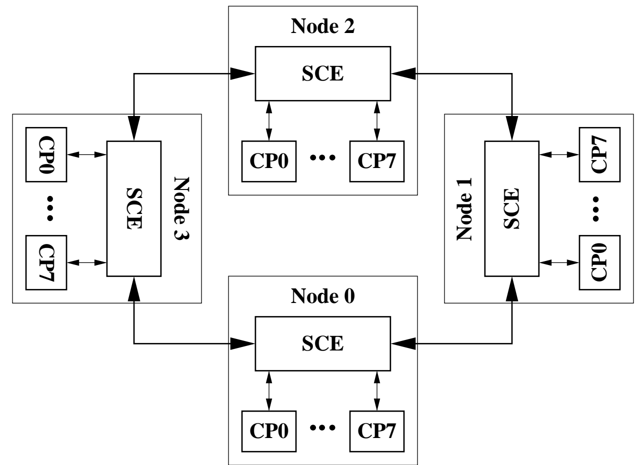

Figure 4: Structure of the SCE simulation environment of an IBM z-series system

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 this experiment, we concentrate 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.

The repository we used consisted of 98 sets of test specifications, each of which was designed to provide the best possible configuration to cover one of the events. The inherent randomness in the test generation mechanism enables the coverage of 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 both the static and dynamic activation policies, we start with examining results from the static policies. We use the greedy algorithm and the reverse greedy algorithm described in Section 3. The graph in Figure 5 shows the difference between the reverse greedy and greedy. The data depicts the expected average results in each test. We show 10 test runs and 20 test runs, and once again we see that the future aware algorithm achieves better results at the end of the planned test and worse intermediate results. We also see how it depends on the number of tests in the run.

In Figure 6 we compare the reverse greedy algorithm's distribution to the greedy algorithm's distribution over the tasks. We can see that reverse greedy finds greedy's hard tasks easier and greedy's easy tasks just as easy. the comparison is done for the case with 20 test runs, a comparison for the 10 test runs case yields similar results

Finally, show the results for the dynamic policies generated by greedy and the future aware greedy, which predicts the future using simple learning techniques. The results are the average of 100,000 regression runs. Figure 7 shows for each step, the average tasks covered for 10 test runs and 20 test runs future aware greedy and the 20 test runs greedy. (It gives the same first 10 test results as the 10 test runs greedy so adding it to the graph doesn't add information). The distribution over tasks also shows that future aware attributes more importance to the weak tasks. In Table 4, we show

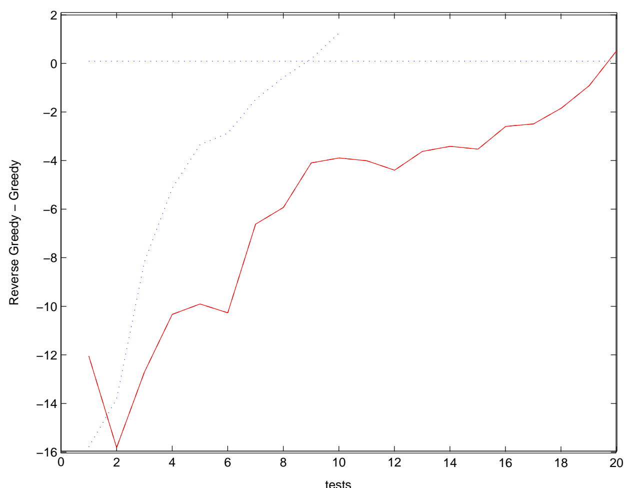

Figure 5: Coverage progress of the SCE model

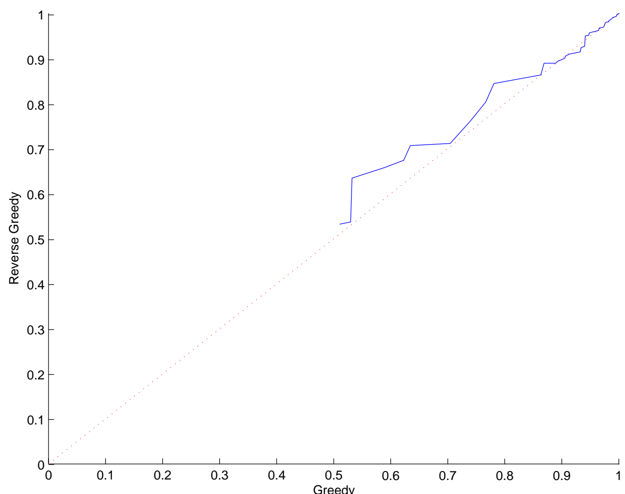

Figure 6: Coverage progress of the SCE model distribution over tasks

the average number of covered tasks (averaged over 100,000 runs) for the 20 test run for greedy, pessimistic future aware greedy and the future aware that uses learning techniques to approximate the future. One can clearly see that the pessimistic is not a good enough approximation.

Greedy	94.94
Pessimistic	94.96
Learning	95.61

Table 4: Average coverage for the UE problem with 20 tests

6. CONCLUSIONS AND FUTURE WORK

The problem of test selection and test prioritization for efficient regression testing has received a lot of theoretical and practical attention. Beside being theoretically interesting, closely related to one of the best known NP-C problems – the set cover problem

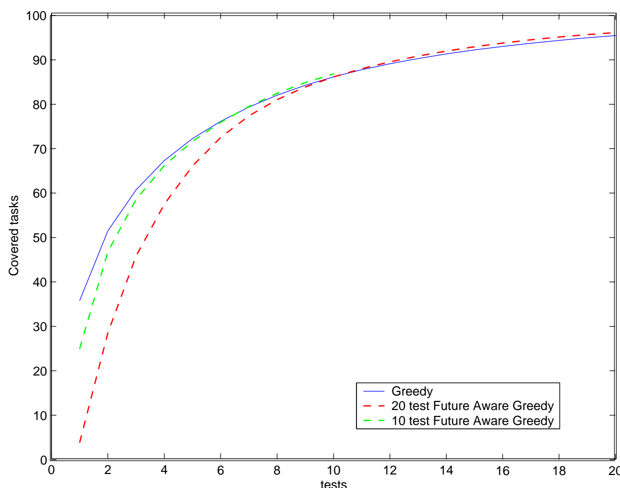

Figure 7: Coverage progress of the SCE model distribution over tasks

[25] – it is also of immense practical value. The problem of creating good regression suites has many facets that are treated in the literature: minimize the number of tests while maintaining good coverage; evaluating which tests are relevant to which code change using impact analysis; choosing tests that are generated by test generators on the fly; and many others.

This paper deals with a related problem of choosing input parameters for test generators in such a way that the suite of generated tests will have good expected characteristics. When we started this research, we were not aware of any work on input parameter selection/prioritization. We started by proposing a method to formulate the construction of random regression suites as optimization problems. solving 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 maximize coverage with a limited number of tests. The quality of the regression suite becomes a known quality and informed decisions can be made regarding the size and distribution of the regression suites.

Not surprisingly, our first attempts [7] were an adaptation of the known algorithms for test selection and prioritization to our domain. The main difference is that a given set of input parameters impacts the probability of having certain characteristics in the tests. The work is not deterministic but probabilistic. We created a new greedy algorithm that chooses parameters to try to maximize the quality of the tests. Unlike the set cover problem, where the selected tasks are removed from the problem, in the probabilistic case, the probability of the tasks being observed is impacted. Experimental results show that the impact is large as the regression suites created are expected to be much better than those created in random. Specifically, we show 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 then came up with the concept of the future-aware greedy for probabilistic regression suites, which is the main contribution of this paper. Future-aware greedy is based on the observation that the amount of resources used to solve the problem is known. Since the number of tests to be executed is known the probability that tasks will be seen in the future can be estimated. This probabil-

ity, together with the knowledge of the past, is used to impact the choice of input parameter. After convincing ourselves that this is useful on artificial problems, we tried the idea on real world problem and showed that it works. The interesting problem, we are just starting to investigate, is how to evaluate the future. "Safe" evaluation turned out to be too pessimistic and not of much value. Learning algorithms show promise and there are probably many other heuristic one can devise, which may perform better in some scenarios.

The work has been executed in two similar domains: one in which the policies use no feedback from the execution and another in which feedback, in the form of coverage, is used. The two domains should not be mixed, as a simple algorithm that uses feedback outperform the best algorithm without feedback. The algorithms used in both are very similar;, however, some heuristic work better in one domain than in the other. An obvious difference between the two is that when feedback is used, it replaces the evaluation of the past. We have not been able to ascertain, or even get good intuition, how this impacts the heuristics. However, we have shown in both domains that it pays to be aware of the future. Further future awareness is shown to yield good results in the deterministic set cover problem as well.

We have shown that future aware heuristics and the use of coverage are both beneficial. This research is in its early stage. We are currently investigating the use of hybrid schemes that combine the building of future aware probabilistic regression suites with snapshots of the current coverage state. We need to decide how frequent checkpoints at which coverage is measured should be. At each checkpoint, re-generation of the regression suite is calculated for the non-covered tasks. As the cost of re-generating the suite is small compared to the cost of testing, we expect that an optimal solution will have several such checkpoints. We also need to improve our prediction of the future. We expect that some heuristics will be better in some domains. As we are discussing testing with limited resources, we have to decide how much of the testing resources should be dedicated to deciding which heuristic to run and how much to running the heuristics. We should see when we reach the point of diminishing and actually negative returns. As all the algorithms discussed in this paper are very efficient, we assume that they will be of benefit but still we can not execute them without regard to resources.

Special attention should be given to tasks that are covered with a very low probability. With our current cost functions, these tasks have low impact on the selection process as trying to reach them is not efficient (unless many tests will are executed). Care needs to be taken so that this task gets proper attention. When regressions are executed regularly, we need to use historical data to decide about which regression to choose after these tasks. As regressions are not independent, we should favor heuristics that reach different tasks in different regressions even if their total is higher – a very similar idea to the future-aware greedy used in a single regression explained in this paper.

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

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

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

[6] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[7] S. Fine, S. Ur, and A. Ziv. A probabilistic regression suite for functional verification. Submitted to DAC043, 2004.

[8] 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.

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

[10] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th international conference on Software engineering*, pages 188–197. IEEE Computer Society, 1998.

[11] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244. ACM Press, 1998.

[12] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 312–326. ACM Press, 2001.

[13] 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.

[14] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th international conference on Software engineering*, pages 119–129. ACM Press, 2002.

[15] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th international conference on Software engineering*, pages 71–80. IEEE Computer Society, 1996.

[16] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th international conference on Software engineering*, pages 308–318. IEEE Computer Society, 2003.

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

[18] B. Marick. When should a test be automated, 1998.

[19] C. C. Michael, G. McGraw, M. Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. In *Automated Software Engineering*, pages 307–308, 1997.

[20] M. J. Moshkov. Greedy algorithm for set cover in context of knowledge discovery problems. In A. Skowron and

M. Szczuka, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[21] K. M. Nigel Tracey, John Clark and J. McDermid. Automated test-data generation for exception conditions. 30(1):61–79, 2000.

[22] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th international conference on Software engineering*, pages 201–210. IEEE Computer Society Press, 1994.

[23] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[24] A. G. M. S. Elbaum and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(2):159–182, 2002.

[25] P. Slavik. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 435–441. ACM Press, 1996.