A probabilistic alternative to regression suites
Shady Copty, Shai Fine, Shmuel Ur, Elad Yom-Tov, Avi Ziv *

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

A R T I C L E   I N F O

Keywords:
Regression suites
Probabilistic regression suites
Coverage analysis
Greedy algorithms
Future-aware greedy algorithms
Hardware verification
Software testing

A B S T R A C T

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

We describe techniques for optimizing random generated test suites. We first show 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. We then introduce a new class of greedy algorithms, referred to as future-aware greedy algorithms. The algorithms are computationally efficient and generate more effective regression suites.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Test suites are often saved to be later reused when the software evolves. This reuse, called regression testing [2,12,14,19,24,25], is pervasive in the software industry. Regression testing is needed primarily to check that no new bugs were introduced to the existing code base. If a test has been executed, and all the bugs that it uncovered were fixed, the only reason for it to fail again is if new bugs were introduced to the existing code. New bugs may get introduced into existing code base as a side effect of bug fixes and refactoring. Running the entire regression suite can require a large amount of effort and there are cases when executing all the tests can take over a month. Therefore, in practice regression suites are created in different sizes. Small suites, sometimes called smoke testing, are used after every build to check for fatal flow. Large suites are used before releases and intermittent size suites are used for intermediate versions and periodic testing (e.g., during weekends). The problem of selecting regression suites of the right size and quality is a difficult one which has been researched extensively.

One approach for creating high coverage regression suites is to find a small set of tests that achieves high coverage, out of those tests that have been executed so far. This is an instance of the set cover problem, which is known to be NP-complete [11]. However, in practice, efficient approaches for solving this problem exist. In addition to finding small regression suites, there is a large body of work that focuses on finding tests that are relevant for changes in the code [18, 24]. Other work focuses on test-case prioritization, which is of importance when only a subset of the regression suite may be executed [14]. Empirical results [12] showed that the test prioritization techniques achieve code coverage at a faster rate and, more importantly, increase the rate of bug detection. It was also shown [25] that no single regression optimization technique is best for all scenarios. Each has its strengths and weaknesses, which depend on the individual use scenario.

* Corresponding author.
E-mail address: aziv@il.ibm.com (A. Ziv).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.03.020
Regression suites have several inherent problems as they are sensitive to changes in the application and its environment. This impacts both the coverage quality of the regression suite and the maintenance cost. These maintenance costs are high because every test in the suite has to be modified when the application is changed. Due to this sensitivity, it is not advisable to automate the regression suite before the application is stable. This limits the effectiveness of regression testing in finding newly introduced bugs in the early stages. In addition, a test that was executed correctly is less likely to find bugs than a new one [20]. Due to this limitation, many experts feel that test automation is sometimes overused [16,20].

In an environment where random test generators are available [3,7,23], these problems can be overcome using random regression suites. Random test generators are tools that generate executable tests, together with expected results. Whenever regression is performed, the random test generator is used to generate tests out of these specifications. Random regression suites do not have the same guarantee with respect to measurable criteria, such as coverage, because tests generated from the same specification can cover different coverage tasks (e.g., they may execute different instructions in the program). However, random regression suites are less sensitive to changes in the application or its environment, and contain new and different tests each time they run. The test suites themselves do not have to be maintained; only the test generator requires maintenance. As test generation specifications are much terser than the test suite, keeping them up-to-date is easier than keeping the tests updated.

Creating random 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 hardware [1,15] and are fairly common in software testing [27]. The most common test generators in software are test data generators, which select new data at random for each test, biased by the specification [8,13,17,21,23]. An interesting example of a test data generator was used to look for date bugs using an aging technique. A test would run, and would then be modified to run with different, randomly chosen, time values. Noise generators of various kinds, which impact the environment in which the tests are executed, are also common. For example, tools are used to simulate network traffic, change the apparent network performance, and cause many other changes that stress the system under test. In the testing of multi-threaded or parallel applications, tools known as noise makers are used to change the apparent behavior of the scheduler [7]. In sequential domains, the input of a test dictates the output. For multi-threading applications, the input as well as the interleaving dictates the outcome. One may think of the noise makers as random test generators, because the noise is randomly biased using specific parameters, in order to impact the interleaving, which impacts the results.

A commonly used method for generating random regression suites is to choose a few specifications (or test parameters) 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 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 in covering each of the coverage tasks. This information can come from previous executions of tests. Another source of information might be coverage-directed generation engines [10] that provide estimates of these probabilities.

Given a set of reliable probability estimates, we showed in [9] how the construction of efficient random regression suites can be formalized as an optimization problem. We addressed two variants of the problem. First, we showed how to construct a 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.

Statistical software testing [28] is a similar method that has different objectives. Unlike the work presented in this paper, statistical software testing tries to change the distribution of the inputs so the least likely tasks will become more common. We have reasons to believe that when the required coverage is high, the results presented by the two methods will be similar.

In this paper, we focus on the problem of creating regression suites that maximize coverage, given limited resources. This paper contains two important contributions. The first contribution is a definition and modeling of measuring the quality of generation algorithms for random regression suites. While random regression suites have been used in the past, and each suite created at random could be evaluated using the same criteria for regular regression suites, there was no measure as to the quality of the algorithm that generates the random regression suites each time from scratch. Section 2 of this paper lays down the formalism needed to evaluate such algorithms.

The second contribution is a new type of future-aware greedy algorithm. The greedy algorithms suggested in [9], 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. We introduce a greedy algorithm that takes statistical expectations of the future into account. Specifically, tasks that are likely to be covered in the future are given less weight when choosing which test specifications to use. We also show that this algorithm is an improvement over the greedy algorithm for the classical set cover problem. We expanded the work done in [9] 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 for on-the-fly selection. In the Appendix, we evaluate the future-aware greedy algorithm on two other problems. We try to predict the types of problems on which the future-aware greedy algorithm will be superior to the regular kind. The Appendix is included in the paper in order to demonstrate that the utility of the future-aware greedy algorithms is not restricted to the regression problem.

We present 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 Section 2, we show how to formalize the construction of random regression suites as optimization problems. In Section 3, we describe the algorithms we use to create random regression suites. Section 4 describes the on-line creation of random regression suites with feedback from coverage. In Section 5, we provide our experimental results. We conclude with a few summarizing remarks and leads for future study. The Appendix evaluates the future-aware greedy algorithm on two additional problems.

2. Constructing random regression suites

We begin by formulating the problem of constructing a random 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 \( t = \{t_1, \ldots, t_n\} \) the set of tasks to be covered. Test specifications are often sets of parameters that govern and bias the generation of tests by a random test generation tool. Thus, we use the term set as an abbreviation, and denote \( s = \{s_1, \ldots, s_m\} \) the repository of sets for which statistical coverage predictors exist. We assume that a single set (test specification) is used for a single test generation run, i.e., the test generator cannot dynamically switch sets or mix individual parameters.

**Definition 1 (The Probability Matrix).** An \( |s| \times |t| \) matrix, where the \((i, j)\) entry, denoted \( P_{ij} \), holds the probability of covering a task \( t_j \) with a test generated using the set (test specification) \( s_i \).

We make the simplifying assumption that \( P_{ij} \) are statistically independent. Furthermore, we assume that these statistical estimates are reliable and hence we do not address issues related to the accuracy and confidence of these predictors.\(^1\) The resulting regression suite is represented by the vector \( w = \{w_1, \ldots, w_m\} \), which specifies an activation policy, such that \( w_i \in \mathbb{N} \) 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 \( w \).

Given a policy \( w \), the probability of covering a task \( t_j \), denoted \( P_j \), is

\[
P_j = 1 - \prod_i \left(1 - P_{ij}\right)^{w_i}.
\] (1)

As 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 [9]:

**Problem 1 (Random Regression Suite).** Find the policy \( w \), which minimizes the number of test executions and provides a desired coverage with high probability

\[
\min_w \sum w_i \\
\text{s.t. } \forall j \quad P_j = 1 - \prod_i \left(1 - P_{ij}\right)^{w_i} \geq Ecnst \\
\forall i \quad N \ni w_i \geq 0.
\] (2)

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

Applying the log operation to both sides of Eq. (2), the constraints in the problem statement defined in Problem 1 can be re-written as linear constraints

\[
\sum_i w_i \cdot g_j \leq \log (1 - Ecnst)
\] (3)

where \( g_j = \log \left(1 - P_j\right) \).

The resulting optimization problem is linear

\[
\min_w \sum w_i \\
\text{s.t. } \forall j \quad \sum w_i \cdot g_j \leq \log (1 - Ecnst) \\
\forall i \quad w_i \geq 0.
\]

The formulation in Problem 1 does not take into consideration any limitations of the resources available for the coverage process. However, practical limitations do exist. They will probably 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

---

\(^1\) 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 test specification.
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 limitations of the batch scheduler in the site). However, resources may translate to other measurable quantities, such as memory consumption. Moreover, the constraints for resource usage may be defined per set, resources allocated by different sets may be charged differently, and there may be a restriction on the total cost of resources allocated to carry out the coverage process.

There is no definite meaning to the term “best” possible regression suite. In this paper, we work with the interpretation that focuses on the expected coverage probability as a quality measure, and thus the next problem definition follows.

**Problem 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 $\mathbf{w}$ that maximizes the expected coverage probability

$$
\max_{\mathbf{w}} \sum_{j} P_j \\
\text{s.t.} \quad \sum_{i} w_i \leq W \\
\sum_{i} \zeta_i w_i \leq C \\
\forall i \quad w_i \geq 0
$$

where $\zeta_i$ is the cost of the overall resource consumption while using test specification $s_i$.

The problem formulated in **Problem 2** is a nonlinear IP, which is quite difficult to handle. The problem is nonlinear because $\mathbf{w}$, the variables of the problem, appear in the exponent of $P_j$ (see Eq. (1)). One possible approach is to apply the annealed approximation

$$
\log \sum_{j} \prod_{i} (1 - P_{ij})^{w_i} \approx \sum_{i} w_i \sum_{j} \log (1 - P_{ij}) .
$$

This approximation, in turn, yields the following linear IP

$$
\max_{\mathbf{w}} \sum_{i} w_i \sum_{j} g_{ij} \\
\text{s.t.} \quad \sum_{i} w_i \leq W \\
\sum_{i} \zeta_i w_i \leq C \\
\forall i \quad w_i \geq 0
$$

where $g_{ij} = \log (1 - P_{ij})$.

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

Finally, the next problem represents a different perspective on the quality of coverage attained with limited resources and focuses on the least probable (or most difficult) tasks to cover.

**Problem 3 (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 covering the task with the lowest probability of being covered

$$
\max_{\mathbf{w}} \min_{j} \left[ 1 - \prod_{i} (1 - P_{ij})^{w_i} \right] \\
\text{s.t.} \quad \sum_{i} w_i \leq W \\
\sum_{i} \zeta_i w_i \leq C \\
\forall i \quad w_i \geq 0
$$

where $\zeta_i$ is the cost of the overall resource consumption while using the parameter set $s_i$.

### 3. Greedy algorithms for random regression suites

The three versions of the problem of creating a random regression suite presented in the previous section are optimization problems. One approach for solving these problems is to provide an analytic solution. However, because the optimization problems are formulated as integer, nonlinear problems, this analytic approach is often difficult and time consuming. In this section, we present a set of simple heuristics that provide a good approximation of the exact optimal solution that can be obtained using an analytical approach. These algorithms are based on a simple greedy algorithm that is often used in set cover and similar problems. The algorithms presented in this section address the random regression suite with limited resources problem (**Problem 2**), but the same algorithms with some minimal trivial changes can be used for the random regression suite problem.
3.1. Greedy algorithm

One approach for creating high coverage regression suites is to find the smallest set of tests that maximizes coverage, out of the tests that have been executed so far. This is an instance of the set cover problem, which is known to be NP-complete [11]. 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 [2].

A similar greedy approach can be used in the random regression suite problem presented in this paper. This greedy algorithm exploits the next simple observation relating the average coverage before and after a step in an iterative algorithm. Let $P_{\text{old}}$ be the probability of covering task $j$ before the step ($P_{\text{old}}$ can be calculated using Eq. (1)). The average coverage after the step, $\bar{C}_{\text{new}}$, can be expressed in terms of $\bar{C}_{\text{old}}$, the average coverage before the step, $P_{\text{old}}$, and $P_{i}$, in the following way:

$$\bar{C}_{\text{new}} = \sum_j \left[ 1 - \left( 1 - P_{i} \right) \left( 1 - P_{\text{old}} \right) \right]$$

where, without loss of generality, test specification $k$ is the one selected for the increment (i.e., $w_k = w_{k-1} + 1$). In other words, the average coverage after the step is equal to the average coverage before the step, plus a change in the coverage, which depends on the previous coverage and the test specification $k$ used in the step.

The pseudo-code of the greedy algorithm appears in Algorithm 1. The algorithm assumes that the limited resource is the number of simulation runs. The inputs to the algorithm are an initial policy (that does not use all the allocated simulation runs) and a limit on the number of simulation runs. During initialization, the coverage probability of each task, $P_{\text{old}}$, is calculated. In the execution phase, at each iteration, we calculate for each test specification $i$, $d_i$, the change in average coverage if this test specification is used. We then select the test specification that maximizes the change in coverage and update the regression suite. This selection is done using the arg max expression on line 12 in the algorithm.

```
Greedy(w₀, W)
Input: an initial policy w₀, resource limit W
Output: a regression suite w

1 // Initialization
2 foreach task j do
3     P_{i} = 1 - \prod_i (1 - P_{i})^{w_i}
4 end
5 w = w₀
6 A = \sum_i w_i
7 // Execution
8 for a ← A + 1 to W do
9     foreach test specification i do
10         d_i = \sum_j P_{i} (1 - P_{\text{old}})
11     end
12     k = arg max(d_i)
13     w_k = w_k + 1
14     foreach task j do
15         P_{i} = 1 - (1 - P_{i}) (1 - P_{\text{old}})
16     end
17 end
```

**Algorithm 1**: Greedy algorithm

3.2. Future-aware greedy algorithm

The greedy algorithm for building random regression suites with limited resources provides efficient and high quality regression suites. However, as the problems are NP-complete, these suites are usually not optimal. One reason for the suboptimality of the greedy algorithm is that it does not consider future steps in the algorithm. Specifically, the greedy algorithm ignores the contribution of the test specifications selected in future steps to the overall coverage. As a result, the
The greedy algorithm may select a test specification with a high probability of covering a given task, and ignore the fact that this task will be covered with high probability in the future, even without the selected test specification.

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 test specifications \( s_1 \) and \( s_2 \) and the coverage probability matrix shown in Table 1.

When the greedy algorithm presented in the previous section is used, test specification \( s_1 \) is used in the first step of the algorithm because of its contribution to the coverage of \( t_1 \). For the same reason, the greedy algorithm also selects \( s_1 \) in the second step. After the second step, the probability of covering \( t_1 \) is high enough (0.96), such that the expected contribution of \( s_1 \) to its coverage in future steps is small. Therefore, the contribution of \( s_2 \) to the coverage of \( t_2 \) is dominant in the next eight steps. The resulting regression suite created by the greedy algorithm is \( w = \{2, 8\} \), with an average coverage of 1.3343. The progress of the average coverage for the greedy algorithm is shown in Fig. 1.

The greedy algorithm ignores the fact that at each step the probability of covering \( t_1 \) increases regardless of the test specification used. Even if \( s_2 \) 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 benefit of selecting \( s_1 \) becomes much lower, and \( s_2 \) becomes the preferred test specification. The resulting regression suite in this case is \( w = \{0, 10\} \), with an average coverage of 1.3730 (see the dashed line in Fig. 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 was 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. 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 test specification that maximizes the coverage after the \((i + 1)\)th step, given the test specification used in the previous \( i \) steps. That is, in each step the goal is to maximize \( \sum_j P_j (1 - p_{old}^j) \), where \( p_{old}^j \) is the probability of covering task \( j \) in the previous steps and \( P_j \) is the probability of covering task \( t_i \) using test specification \( s_i \). The future-aware greedy algorithm replaces this goal function with \( \sum_j P_j (1 - p_{old}^j)(1 - F_j) \), where \( F_j \) is an estimation of the probability of covering task \( t_j \) in future steps. The pseudo-code of the future-aware greedy algorithm is identical to the pseudo-code of the greedy algorithm presented in Algorithm 1. The only difference is replacing the calculation of the change in coverage on line 10 of the greedy algorithm with the following equation that considers the future.

\[
d_i = \sum_j P_j (1 - p_{old}^j)(1 - F_j).
\]

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 solving the problem optimally. An optimistic estimation of \( F_j \) may degrade the quality of the solution, since it may unnecessarily punish good test specifications 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 test specifications.
When a good method for estimating $F_j$ is used, the future-aware algorithm should outperform the greedy algorithm, sacrificing current for future performance. 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. Fig. 1 illustrates this.

In our experiments (see Section 5), we examined several methods of estimating $F_j$. These estimation methods differ in their accuracy and computational complexity. Generally, however, 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:

- $F_j = 1 - (1 - M_j)^l$, where $l$ is the number of steps left in the algorithm and $M_j$ is the minimum probability of covering task $t_j$ over all test specification (i.e., $M_j = \min P_i^j$). This estimation is obviously pessimistic and it is useful only if the coverage probability matrix $\{P_i^j\}$ 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 $s_i$ be the test specification selected in step $l$ of the algorithm, then an estimation of the future probability of covering task $t_j$ after the $k$ step is given by

$$F_j = 1 - \prod_{i=k+1}^l (1 - P_{ij}).$$

Algorithm 2 provides the pseudo-code of this version of the future-aware greedy algorithm. At each iteration of the algorithm, and for each test specification, we first assume that this set is selected in the iteration. We call the normal greedy algorithm (of Algorithm 1) with this starting point to calculate the contribution of the future. Next, we calculate the contribution of the current step given the past and the future, using Eq. (6). The rest of the algorithm is identical to the one in Algorithm 1.

```
Algorithm 2: Future-aware greedy algorithm

FutureAwareGreedy($w^0, W$)
    Input: an initial policy $w^0$, resource limit $W$
    Output: a regression suite $w$
    // Initialization
    foreach task $j$ do
        $p_{ij}^{old} = 1 - \prod_i (1 - P_{ij})^{w_i^0} $
    end
    $w = w^0$
    $A = \sum_i w_i$
    // Execution
    for $a \leftarrow A + 1$ to $W$ do
        foreach test specification $i$ do
            $wt_i = w_i$
            $wg = \text{Greedy}(wt, W) - wt$
            foreach task $j$ do
                $F_j = 1 - \prod_m (1 - P_{mj})^{wg_m}$
            end
            $d_i = \sum_j P_i^j (1 - p_{ij}^{old}) (1 - F_j)$
        end
        $k = \arg \max(d_i)$
        $w_k = w_k + 1$
        foreach task $j$ do
            $p_{ij}^{old} = 1 - (1 - P_{ij}^k) (1 - P_{ij}^{old})$
        end
    end
```

Algorithm 2: Future-aware greedy algorithm
3.3. Reverse greedy algorithm

As stated in the previous subsection, one of the problems of the greedy algorithm is that it does not consider its future actions when selecting a test specification in the current step. As a result, the greedy algorithm first handles the easier parts of the problem (i.e., tasks that have high probability of being covered by some test specifications), where it can make the most progress. Only after the easier parts are covered enough, the greedy algorithm turns to the harder parts of the problem, where progress is slower. In doing so, the greedy algorithm ignores the contribution of the test specifications selected for the harder parts to the easier parts. This phenomena is illustrated in the example in Section 3.2. The future-aware algorithm addresses this issue by trying to estimate the contribution of future test specifications.

The Reverse Greedy (RGreedy) algorithm presented below is the next step in using the future. Instead of estimating the future, the Reverse Greedy algorithm chooses in the future. That is, in each step of the algorithm we execute the greedy algorithm from the current state and choose the last test specification selected by the greedy algorithm. By doing so, the reverse greedy algorithm starts with the hard parts of the problem, assuming that the easier parts will take care of themselves. If this does not happen, the last steps of the algorithm will be used for easy parts.

The pseudo-code for the reverse greedy algorithm appears in Algorithm 3. The input and output of the algorithm are the same as the greedy algorithm of Algorithm 1. We assume here that the greedy algorithm returns \( w_0 \), the selection order of the test specifications, instead of the number of times they should be used. Specifically, \( w_{0\text{last}} \) is the last test specification selected by the greedy algorithm. At each iteration of the algorithm, the reverse greedy algorithm calls the greedy algorithm to work from the state at the end of the previous step. Then, it selects the last test specification taken by the greedy algorithm and uses it as the next test specification in its execution. This process continues until all the resources are used.

\[
\text{ReverseGreedy}(w^0, W)
\]

Input: an initial policy \( w^0 \), resource limit \( W \)
Output: a regression suite \( w \)

1 // Initialization
2 foreach task \( j \) do
3 \quad \( p^\text{old}_j = 1 - \prod_i (1 - p^0_i) w_i \)
4 end
5 \( w = w^0 \)
6 \( A = \sum_i w_i \)
7 // Execution
8 for \( a \leftarrow A + 1 \) to \( W \) do
9 \quad \( w_0 = \text{Greedy}(wt, W) \)
10 \quad \( k = w_{0\text{last}} \)
11 \quad \( w_k = w_k + 1 \)
12 foreach task \( j \) do
13 \quad \( p^\text{old}_j = 1 - (1 - p_j^k) (1 - p^\text{old}_j) \)
14 end
15 end

Algorithm 3: Reverse greedy algorithm

The Reverse Greedy algorithm is fairly efficient. If the number of steps in the greedy algorithm is \( n \) and the cost of each step is constant (as in the greedy algorithm presented above), then the cost of RGreedy is at most \( n/2 \) times the cost of the greedy algorithm.

Note that the future-aware greedy and reverse greedy algorithms are not limited to the random regression suite problem. Appendix presents other cases where the future-aware and reverse greedy algorithms outperform a simple greedy algorithm.

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 covered in the step and start generating a "new" suite containing fewer tasks. In addition, we assume that in each step a single test specification is selected and executed. Under this assumption, the greedy algorithm elects the test specification \( s_i \) that maximizes the expression \( \sum_j P_{ij} \), where \( P_{ij} \) is the probability of covering \( t_j \) using \( s_i \) if \( t_j \) is not covered and 0 if \( t_j \) is covered.

The basic extension of this algorithm to the future-aware and reverse greedy algorithm is trivial. At each step of the algorithm we execute the appropriate algorithm from the current state (considering the tasks that are already covered) and select the test specification that is chosen first for execution. In other words, for the future-aware greedy algorithm we select the test specification \( s_k \), such that \( k = \arg \max_i \sum_j P_{ij}(1 - F_j) \), where \( F_j \) is the estimation on the probability of covering task \( t_j \) in the future. For the reverse greedy algorithm, we execute the normal greedy algorithm and select the last test specification selected by it (same as lines 9–10 in Algorithm 3).

The future-aware approach with intermediate measurements can be 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 test specifications and the coverage probability matrix shown in Table 2.

Both the greedy algorithm and the future-aware greedy algorithm presented in the previous section select \( s_1 \) (or \( s_2 \)) 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 \( s_3 \) in the first step can improve the solution, because after the coverage measurement, we know whether \( t_1 \) (or \( t_2 \)) is covered and \( s_1 \) (or \( s_2 \)) 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 test specification, but also the amount of information added when the test specification is executed (i.e., the amount of uncertainty in the test specification). For example, we can generate the entire suite (without measurement) and select from it the test specification that has the highest information (in the classical information theory sense \( I = - \sum_i p_i \log p_i \)).

### 5. Experimental results

To demonstrate the feasibility and applicability of the suggested formalisms and algorithms, we conducted several experiments on regression data collected from both software testing and hardware verification environments, using real-life applications and coverage models. In previous sections, we described two approaches for using our algorithm. In one approach, our policy is dynamic, in 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. We will discuss each approach separately with respect to specific examples in the following section.

#### 5.1. Regression suite for software testing

A test in the multi-threaded domain is a combination of inputs and interleaving, 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. Therefore, a regression suite does not have the same meaning as in the sequential domain, since the same inputs may cause different executions of the program.

ConTest [7] is a tool for generating different interleavings for the purpose of revealing concurrent faults. ConTest takes a heuristic approach of seeding the program with instrumentation in concurrently significant locations. At runtime, we make heuristically, possibly coverage-based, decisions regarding which noise (for example, `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 parameter. Low strength parameter means that `yield()` is executed just a few times and high strength means that `yield()` is executed many times. 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 nine synchronization events (e.g., a thread waiting for a specific sleep statement) as tasks.

During the testing process, we collected all the statistics needed to construct random 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.

Given the statistics collected for the 18 heuristics, we constructed dynamic policies designed to maximize the coverage of the nine events, using five test runs in the first experiment and ten test runs in the second one. These policies were executed 100,000 times each, and the results show the average coverage they yielded.

In Fig. 2 we show how the pessimistic future-aware greedy algorithm, the worst of our future-aware heuristics, is able to produce better results than the greedy algorithm. The graph shows curves corresponding to the difference between future-aware greedy and greedy for five and ten executions (respectively). In the figure, the X-axis denotes the number of tests; while the Y-axis denotes the difference in expected coverage (measured in number of covered tasks). The solid line in the figure represents the ten executions case, and the dotted–dashed line the five executions. We see how greedy achieves better results at the beginning (greedy is better and therefore the curve is negative), while future-aware bypasses it (becomes positive) at some point in the middle of the whole regression. As the future-aware greedy algorithm can predict which tasks will be covered regardless of its efforts, it invests in harder tasks. In the five test runs, we see how the future-aware starts to climb earlier and bypasses the greedy around the third test.

5.2. Regression suite for hardware verification

The goal of the experiments was to construct random regression suites paired with static activation policies that maximize 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 zSeries system [10]. 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 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 is 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 started by examining results from the static policies, and used the greedy algorithm and the reverse greedy algorithm described in Section 3. The graph in Fig. 3 shows the difference between reverse greedy and greedy. The data depicts the expected average results in each test. We show 10 test runs (dotted line) and 20 test runs (solid line). 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.

Finally, we show the results for the dynamic policies generated by greedy and future-aware greedy, which predicts the future using simple learning techniques. The results are the average of 100,000 test runs. Fig. 4 shows, for each step, the average number of tasks covered by runs of length 10 and 20 of the future-aware greedy, compared to runs of length 20 of the greedy algorithm. We do not show runs of length 10 of the greedy algorithm since as the greedy algorithm is future oblivious, running it for length 10 is exactly the same as the first 10 runs of length 20. The probability distribution over tasks also shows that future-aware greedy yields a higher probability than the greedy algorithm for tasks with lower coverage probability. This result is intuitive because future-aware greedy, when compared to simple greedy, tries first for the lower probability tasks.

In Table 3, we show the average number of covered tasks (averaged over 100,000 runs) for the 20 test runs for greedy, pessimistic future-aware greedy, and future-aware that uses learning techniques to approximate the future. One can clearly see that the pessimistic is not a good enough approximation. Deciding how to estimate the future, with future-aware greedy algorithms, is a very interesting research direction, especially looking for good algorithms with low computational costs.
The promise of the future-aware greedy approach is not limited to optimizing regression suites. We are in the first stages of investigating the use of hybrid methods that combine the building of future-aware random regression suites with snapshots of the current coverage state. We need to decide the frequency of the checkpoints at which coverage is measured. 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 should see when we reach the point of diminishing return.

We introduced the concept of future-aware greedy algorithm for random regression suites—the main contribution of this paper. Future-aware greedy is based on the observation that since the number of tests to be executed is known, the probability that tasks will be seen in the future can be estimated. This probability, 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 problems and saw 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 of little value. Learning algorithms show promise and there are probably many other heuristics 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 outperforms the best algorithm without feedback. The algorithms used in both are very similar; however, some heuristics 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 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 methods that combine the building of future-aware random regression suites with snapshots of the current coverage state. We need to decide the frequency of the checkpoints at which coverage is measured. 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 should see when we reach the point of diminishing return.

To execute future-aware greedy algorithms, one needs a future-aware evaluation function. Such functions are problem specific. We are currently working on such functions for other problems to see if the technique suggested can be generalized.

The promise of the future-aware greedy approach is not limited to optimizing regression suites. We are in the first stages of investigating the use of hybrid methods that combine the building of future-aware random regression suites with snapshots of the current coverage state. We need to decide the frequency of the checkpoints at which coverage is measured. 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 should see when we reach the point of diminishing return.

To execute future-aware greedy algorithms, one needs a future-aware evaluation function. Such functions are problem specific. We are currently working on such functions for other problems to see if the technique suggested can be generalized.

The promise of the future-aware greedy approach is not limited to optimizing regression suites. We are in the first stages of investigating the use of hybrid methods that combine the building of future-aware random regression suites with snapshots of the current coverage state. We need to decide the frequency of the checkpoints at which coverage is measured. 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 should see when we reach the point of diminishing return.

To execute future-aware greedy algorithms, one needs a future-aware evaluation function. Such functions are problem specific. We are currently working on such functions for other problems to see if the technique suggested can be generalized.

The problem of test selection and test prioritization for efficient regression testing has received a lot of theoretical and practical attention. Besides being theoretically interesting, and closely related to one of the best known NP-complete problems—the set cover problem [26]—it is also of immense practical value. The problem of creating good regression suites has many facets that are treated in the literature: minimizing 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 addresses a related problem of choosing input parameters for test generators in such a way that the suite of generated tests will have good expected coverage characteristics. We started by proposing a method to formulate as optimization problems the construction of random regression suites generated to achieve good coverage. Solving the optimization problem provides information about 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.

Not surprisingly, our first attempts [9] were an adaptation of the known algorithms for test selection and prioritization to our domain. The main difference between the domain of these known algorithms and our domain is that in our domain 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 test specifications to try and maximize the quality of the tests. Unlike the set cover problem, where the selected tasks are removed from the problem, in the random 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 at 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 than using all the resources on the best or distributing the resources evenly.

We compared the dynamic and static policies for the different algorithms for the UE problem, and as we expected the naive dynamic algorithm performed better than all the static algorithms (see Table 4).

### Table 4

<table>
<thead>
<tr>
<th></th>
<th>Worst</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic policy</td>
<td>94.9</td>
<td>95.6</td>
</tr>
<tr>
<td>Static policy</td>
<td>92.8</td>
<td>93.33</td>
</tr>
</tbody>
</table>

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. Besides being theoretically interesting, and closely related to one of the best known NP-complete problems—the set cover problem [26]—it is also of immense practical value. The problem of creating good regression suites has many facets that are treated in the literature: minimizing 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 addresses a related problem of choosing input parameters for test generators in such a way that the suite of generated tests will have good expected coverage characteristics. We started by proposing a method to formulate as optimization problems the construction of random regression suites generated to achieve good coverage. Solving the optimization problem provides information about 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.

Not surprisingly, our first attempts [9] were an adaptation of the known algorithms for test selection and prioritization to our domain. The main difference between the domain of these known algorithms and our domain is that in our domain 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 test specifications to try and maximize the quality of the tests. Unlike the set cover problem, where the selected tasks are removed from the problem, in the random 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 at 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 than using all the resources on the best or distributing the resources evenly.

We compared the dynamic and static policies for the different algorithms for the UE problem, and as we expected the naive dynamic algorithm performed better than all the static algorithms (see Table 4).
Appendix

In this section, we show two additional applications for the future-aware greedy algorithm. The suggested algorithm performs better on one of the applications, but fails on the other. We suggest that these applications typify the cases where the future-aware greedy algorithm will work, compared to problems where it will not.

Many problems exist, which contain hard and easy components. It is a common strategy to try to solve the hard parts first and then deal with the rest. For example, the greedy implementation of the set cover problem contains an initial step in which it selects all the subsets containing an element that only they cover. Another example is packing, in which items that are hard-to-fit are selected first. An even more extreme example is in [22], where two playing strategies for Othello were compared: a greedy strategy and an approach that tried to figure out the important aspects of the game and concentrate on them. In a game between the two, the latter will be in a losing position all the way to the end, when the situation will dramatically reverse. Pure greedy algorithms, due to their preference of quantity over quality, tend to miss those harder cases. Reverse greedy and future-aware greedy both give preference to the harder problems and start with them. Both contain a built-in mechanism, based on the cardinality of the solution set to be chosen, which calibrates the selection of solutions whose contribution is not too small for the selected cardinality. The very esthetically pleasing result is that the utility functions of the suggested solutions surpass those of the greedy algorithm, only close to the solution cardinality, for any given cardinality.

Our experiments, detailed below, suggest that the future-aware greedy algorithm will not perform well in cases where the application of small changes, such as those attempted by the suggested algorithm, make it impossible to approximate the large parts of the problem using the available components. This is apparent in the matching pursuit problem, where the future-aware greedy algorithm causes the solution to have non-smooth characteristics that cannot be approximated correctly. This is similar to packing the small items first and then being unable to pack the hard-to-fit items. On the set cover problem and the facility location problem, we found that choosing the harder solutions first improves the final result. It is our assumption that after choosing the hard solutions first, the choice between the easy solutions, of which more exist, is more efficient.

A.1. Facility locating problem

Given \( N \) cities, choose \( M \) cities (\( 0 < M < N \)) in which to build a facility such that the sum of all minimum distances between cities and facilities is minimized. It is assumed that the distance between every two cities is known. \textbf{Fig. 5} shows an example of such a problem with 12 cities and four facilities. This type of facility-locating problem is NP-hard [11].

The greedy algorithm for this problem is quite trivial: Keep choosing cities in which to place facilities until you reach \( M \) cities; at each step, given the preceding choices, choose the city that minimizes the sum of all minimum distances. The RGreedy algorithm for the problem is for each step to run the greedy algorithm with the number of facilities left to locate, and pick the greedy algorithms’ last facility location as the current chosen facility location. In the variations that try to overcome the over-approximation problem, we run a fraction of the way to the end, and then the last city, \( \text{RGreedy}(\kappa = 0.5) \) for example, runs the greedy algorithm half the way at each step and takes the last solution as the current solution.

Several experiments were conducted to show the advantage RGreedy has over the greedy algorithm for this problem. The number of cities was chosen to be 50 for all the experiments. The number of facilities chosen was 5, 10, 15, 20, 25, 30, 35 and 45. Since choosing where to place \( M \) facilities is like choosing where not to place \( N-M \) facilities, the problems increase in complexity as they get closer to \( M = 25 \), and then decrease as they climb to 50. The random maps generated satisfy the triangle inequality. In this problem, we examine a different set of RGreedy variations, with \( \kappa \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1\} \). Again, we evaluate the performance of each of our algorithms in comparison to other algorithms.
Fig. 6 shows the performance of each algorithm. Since the performance is relative to other algorithms’ results, they should be examined as such. We clearly see that for the different number of facilities, $R\text{Greedy}(\kappa \leq 1.0)$ is better than that of the greedy algorithm. We again notice that the best performing $R\text{Greedy}$ is different for different settings. For example, the problem with $M = 25$ is best handled by $R\text{Greedy}(\kappa = 0.8)$ while $M = 15$ is best handled by $R\text{Greedy}(\kappa = 0.9)$ and $M = 10$ by $R\text{Greedy}(\kappa = 1)$. The behavior in $M = 5$ originates from the small difference between the different fractions for a small horizon such as 5. As for the loss of significance for problems with $M > 25$, we relate it to the problem becoming easier and for the long horizon where over-approximation is unavoidable.

A.2. Matching pursuit

Matching pursuit (MP) is a method for the suboptimal expansion of a signal in a redundant signal dictionary [6]. This algorithm, combined with a dictionary of Gabor functions, defines a time–frequency transformation. The matching pursuit algorithm works by iterative subtraction of the best matching dictionary functions (known as atoms) from the signal, with the appropriate amplitude and phase. Since at each iteration the best matching function is subtracted, this is a greedy algorithm.

Matching pursuit was used to expand a one-dimensional chirp signal of length 64, over a dictionary that contained approximately 21,000 Gabor functions. We ran the MP algorithm (denoted by Greedy MP) for 20 iterations. At each iteration, after finding the best fitting atom to the signal, its parameters were fine-tuned using 20 iterations of Nelder–Mead multidimensional unconstrained nonlinear minimization [4]. This atom was then subtracted from the remaining signal.

The $R\text{Greedy}$ version of MP was also used for approximating the same chirp signal. The $R\text{Greedy}$ version of MP is straightforward. Instead of using the best-fitting atom for subtraction, it uses the atom used by Greedy MP after, for example, 50% of the iterations (in the case of $R\text{Greedy}(\kappa = 0.5)$). The amplitude and phase used for subtraction are those found for Greedy MP.

The test was repeated with various levels of white Gaussian noise added to the chirp signal. Each variant of the algorithm was run against 10 realizations of noise at each noise level.
Fig. 7 demonstrates the average energy (over 10 realizations) of the residual signal, when approximating a signal with no noise added. As can be expected, Greedy MP (Reverse-Greedy with $\kappa = 0$) reduces the residual energy more rapidly, compared to RGreedy with $\kappa > 0$, which reduces most of the energy at later iterations.

As Fig. 8 shows, at a given noise level, Greedy MP attains a smaller residual energy level compared to any variant of RGreedy MP. This effect is more prominent at lower noise levels where the difference in the residual energy of Greedy MP and RGreedy MP is larger than an order of magnitude.

This is most likely due to the fact that the dictionary contains smooth Gabor functions. Thus, Greedy MP first finds a fit for the global features of the signal and gradually progresses to fitting more localized features. By using the atoms found in later iterations as the first atoms to be removed, the global features (that contain the most energy in the signal) are removed, and thus it is difficult for RGreedy MP to remove energy as efficiently as Greedy MP. This explains why, at high noise levels, where the smoothness of the signal is lost, the difference between Greedy MP and RGreedy MP is less pronounced as compared to the difference at low noise levels.

References


