Regression Test Selection Techniques for Test-Driven Development

Hagai Cibulski[†]
The Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
Email: hagaicib@tau.ac.il

Amiram Yehudai
The Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
Email: amiramy@tau.ac.il

***Abstract*—Test-Driven Development (TDD) is characterized by repeated execution of a test suite, enabling developers to change code with confidence. However, running an entire test suite after every small code change is not always cost effective. Therefore, *regression test selection* (RTS) techniques are important for TDD. Particularly challenging for TDD is the task of selecting a small subset of tests that are most likely to detect a regression fault in a given small and localized code change.**

We present cost-bounded RTS techniques based on both dynamic program analysis and natural-language analysis. We implemented our techniques in a tool called TestRank, and evaluated its effectiveness on two open-source projects. We show that using these techniques, developers can accelerate their development cycle, while maintaining a high bug detection rate, whether actually following TDD, or in any methodology that combines testing during development.

***Index Terms*—Development Tools, Test coverage of code, Test execution.**

I. Introduction

The Test-Driven Development (TDD) methodology involves small cycles of writing a test, implementing that test and refactoring the code [1]. A regression test suite is executed after each code change, to guard against regression bugs. Over time the test suite grows larger and slower, while on the other hand, TDD relies on rapid response to small code changes, in the form of executing the regression test suite after each such change. For example, JUnit FAQ [2] recommends to run tests "ideally every time the code is changed", but then immediately adds "For larger systems, you may just run specific test suites that are relevant to the code you're working on.". Indeed, TDD developers look for ways to spend less time rerunning their test suite, and one of the most effective ways to speed up retesting, is to select and run only the tests that are relevant to the latest code change.

How does one find the relevant tests for a given code change? In the particular case of using strict unit tests, finding the relevant test for a given unit is straight forward (i.e., using naming conventions). However, many TDD tests are not unit tests. A TDD test, written before implementing some functionality, may be called a functional programmer test. The functionality tested by a single TDD test is implemented in code that is scattered among many units. TDD also makes heavy use of refactoring [3], leading to highly factored code where each unit has tangled functionality defined by many tests.[1] Functionality scattering and tangling mean that it is not immediately clear which tests cover which units of code. This results in developers either wasting a lot of time running irrelevant tests, or trying to reduce the retesting time using strategies such as procrastination and/or guessing. The result of either is less rigorous practice of TDD and poorer code quality. While this problem is particularly evident in TDD, it also exists to some degree in any setting where frequent testing is performed during development, which is a common theme in many agile techniques.

Our goal is to automatically find, for a given small and localized code change, a small subset of tests, with a bounded cost of execution, that provides a high bug detection rate, close enough to that of full retesting. This bounded *regression test selection* (RTS) problem requires a different approach from that of traditional RTS techniques. Instead of finding an unbounded subset of tests that detect faults 100% of the time, we want to find for example 20% of the tests that will reveal a failure with 80% probability. Detecting a failure 80% of the time is justified by the assumption that the developer will eventually run all tests — at least once per day, in order to secure a 100% safety net. This is different from safe RTS techniques [4]–[6], which do not guarantee a bound on the selection size, or general *test case prioritization* (TCP) techniques [7], [8] that are generally not change sensitive.

More formally, given a program P, a test suite T and a code change c, we define $fail(T, P, c)$ as the subset of tests in T that fail on program P after change c. We also define the cost of executing a subset of the test suite, as a function $cost : \mathcal{P}(T) \to \mathbb{R}$. We now define the Bounded RTS problem as follows:
Given an upper bound $\alpha < 0.5$, find $T' = selection(T, P, c)$, s.t:

$$cost(T') \leq \alpha \cdot cost(T)$$

[†]*Supported in part by the Deutsch Institute*

[1]For example, suppose that initially two tests were implemented by two units. After factoring out the common code from the two units, the common unit now participates in implementing both tests. We say that the common unit has tangled functionality defined by the two tests.

and if $t \in fail(T, P, c)$, then:

$$Pr(t \in fail(T', P, c)) \geq 1 - \alpha$$

The bound α is equal to the worst case false positive rate. The *fault recall rate* is equal to $1 - \alpha$. In practice, we will often discuss bounded RTS results in terms of (α, β) pairs, where β is the *fault recall rate* corresponding to the bound α.

Our approach to bounded RTS is reducing it to a prioritization problem. We implement bounded RTS by applying *change-sensitive test case prioritization* on top of a safe RTS filter, and selecting the top α test cases from the prioritized list.

Like previous works on test selection and prioritization [8], we will assume that all test cases in the test suite are independent of each other. That is, a test case does not depend on the state of the system that was established by previous test cases. This assumption guarantees that a test's result (pass or fail) does not depend on what other test cases exist in the selection, or on the ordering of test cases.

In the next section we will discuss dynamic program analysis techniques used in RTS and TCP. We will also examine a useful technique of *natural language program analysis*, which was found effective at solving similar problems in the past. Section III presents our techniques that include novel dynamic program analyses and natural-language analysis. We describe a tool called TestRank that uses a combination of these techniques in order to implement bounded RTS. We evaluated our techniques against safe selection (section IV), and validated their effectiveness. Section V presents conclusions, and discusses ideas for future research and applications.

II. Related Work

There are many existing works on regression test selection (RTS) [4]–[6]. However, these works focus on system validation testing, which is done after all the developers checked in their changes, and a new version of the system was built. The input in this case is not a program P and an isolated change c, but rather two versions of the program P and P', with many changes between them. Moreover, in the system retesting scenario there is no need for immediate feedback to the developer, so such techniques can incur significant overhead [9]. Most important, the goal of system validation is to assure the quality of the system as much as possible. Consequently, almost all past RTS systems are "safe" RTS systems, meaning if a test t should fail it must be included in the selection T'. We note that this "safety" is never absolute due to possible nondeterminism in the system under test. Handling nondeterminism in a conservative manner requires further static analysis, incurring higher overhead and poor precision [4]. In the TDD context, we defined the bounded RTS problem in which high precision is the goal, and sometimes not reporting a test that actually will fail may be acceptable. A "safe" RTS filter is a good starting point, but a bounded RTS technique would still need to reduce the selection size.

Ren et al. [10] described a tool that can conservatively report affected tests, given a a changed version of an object-oriented program, while handling object-oriented programming difficulties. This tool, called Chianti, uses static analysis to identify code changes, such as overriding method introduction, that might have a non-local effect.

A different approach to test suite optimization is test case prioritization (TCP) [7], [8], [11], [12]. TCP's goal is to schedule the test cases in an order that increases the rate of fault detection (*fail fast*). For example, a good heuristic for this is ordering the tests by decreasing additional coverage. In TCP safety is guaranteed because all tests are eventually run. General TCP is "global", and not change sensitive. That is, it does not consider the code change that we wish to test. Different code changes may cause different tests to fail, but general TCP would always order the tests the same. This might be a problem for bounded RTS, especially if the bound α is very small.

Srivastava and Thiagarajan [13] described a tool that performs version specific TCP. The tool, named Echelon, compares two versions of a program, and orders the given tests by decreasing additional coverage of changed basic blocks in the program. The underlying assumption in this case is that the program version P' contains multiple code changes. However, in a typical short TDD cycle, the only input available might be a single changed code fragment. As we shall see, our solution ranks tests based on their relevance to a single changed code fragment.

Elbaum et al. (2000) [7] suggested that version specific TCP techniques (as in Srivastava and Thiagarajan [13]) were found to improve the rate of fault detection of test suites in system regression testing. The same paper describes a test case prioritization technique based on mutation analysis [14]. However, the authors acknowledge that mutation analysis is an extremely slow process.

Historically, most work on TCP was done on code bases written in C. Do et al. [15] studied how TCP techniques generalize to Java programs and the JUnit testing framework [16]. They found additional-coverage version-specific TCP to be the most effective (as in as in Srivastava and Thiagarajan [13] and Elbaum et al. (2000) [7]). However, a surprising result was that finer granularity analysis (block level vs. method level) has no effect on TCP of Java programs. This is contrary to previous work on TCP of C programs [11], and is probably a results of small constructors and methods (i.e., getters and setters) typical of object oriented programs.[2] Such small methods usually have only one or two basic blocks. In the next section, we will present our solution for Java programs, which uses method-level analysis that may be justified given these findings.

Continuous Testing (CT) [17] is another approach to speed up retesting during development. CT uses spare CPU cycles to continuously run tests in the background, providing rapid feedback about test failures, as source code is edited.[3] In practice, CT works best in combination with TCP. Therefore

[2]Since refactoring [3] is integral to TDD, it is expected that small methods are even more common in TDD code bases.

[3]JUnit FAQ [2] even recommends Saff's continuous testing plug-in.

introducing the notion of change sensitive TCP, may further improve the effectiveness of CT tools.

CT techniques, as well as RTS and TCP, in the context of testing during development, have recently achieved some penetration into the software development industry. Infinitest [18] is a CT runner. Clover coverage tool now features Test Optimization [19]. Both Infinitest and Clover combine safe RTS with simple global TCP (fast-tests- and recently-failed-tests- first). Kent Beck, the "father" of TDD, developed JUnitMax [20], which also features CT and similar simple TCP. Other similar industry tools include Google Testar [21], JTestMe [22] and ProTest [23]. None of these tools offer to bound the cost of running the test suite. Nor does any of these tools order tests by any criteria related to the current code edit. We believe that such tools are an important step forward towards accelerating TDD retesting, and that combining bounded RTS and change-sensitive TCP with any of these tools may further improve their effectiveness.

Pollock et al. [24] have observed strong indicators that there are many natural language clues in program literals, identifiers and comments that could be leveraged to increase the effectiveness of many software tools. The CodePsychologist, which was previously developed in our group [25], is a tool which assists the programmer to locate source code segments that caused a given regression bug. The CodePsychologist uses affinity ranking to estimate how close a segment of code is to a given test case. Affinity between groups of words is calculated based on the semantic similarity between pairs of words from each group, measured as the inverse path length in a WordNet taxonomy [26], [27]. We use a similar algorithm for estimating how close a test case is to a given segment of code (see section III-C).

III. The TestRank Technique

In this section we will present a tool called TestRank that implements our change-sensitive TCP technique. The overall structure of the tool is illustrated in Fig. 1. The input to TestRank is a program P and its related test suite T. Both P and T are assumed to be written in the Java programming language. An initial preprocessing phase creates the TestRank database. This preprocessing can be performed off-line each night, in order to synchronize the system with the latest baseline version. Any change during the day is then queried in the context of that baseline version. The TestRank query engine accepts queries $Q(file : line)$, and outputs a list of tests from T ranked by relevance to changes in Q. The query is very simple and efficient (In our experiments in the next section it took less than one second).

We shall now explain these steps in more detail. After setting up the system, a preprocessing stage is executed. During preprocessing the entire test suite T is run under tracing. We trace the tests through the production code, currently using AspectJ [28]. AspectJ allows us to define a *pointcut* to identify each method call and execution in the application under test, and an *advice* to automatically intercept these execution points with our tracing. As a result of AspectJ's join-point model,

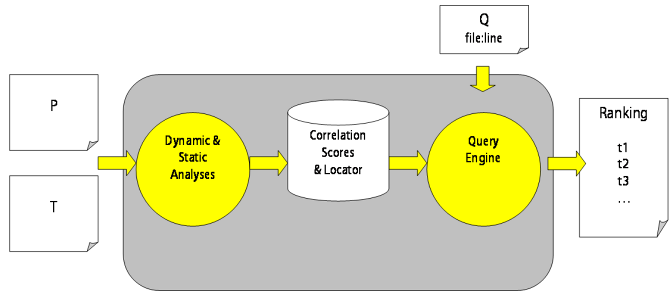

Fig. 1. TestRank Architecture

the basic traced unit is at the *method* level granularity. Finer granularity techniques do not necessarily yield significant improvement [7], especially in Java programs [15]. This is significant since coarse granularity analysis is much faster, especially for large or CPU intensive code bases.

During tracing we collect the coverage data, which is used as a basic safe-RTS filter. We also compute various metrics that serve to estimate the correlation between test cases and the units under test. These metrics are used to prioritize the tests which passed the safe-RTS filter. We trace all test executions, and method executions and calls. For each execution stemming from test t, we record the number of times each method m was executed, the number of distinct calls to that method (how many methods call this method?), and the depth of the stack during these calls. These dynamic analyses serve as the basic prioritization heuristics, or *predictors*. We also record the values of arguments flowing out of the test and those reaching the method under test. These are used by the Value Propagation predictor. We explain these dynamic predictors in section III-B.

Another type of heuristics that we set out to explore are natural language based. As was mentioned in the previous section, past results [25] suggest that these types of heuristics may increase the effectiveness of our tool. After tracing is finished, we statically analyze the code looking for natural language clues in the sources text, for further natural language analysis, which will be described in Section III-C.

All scores for all pairs from the dynamic predictors and the affinity based predictors are stored in the TestRank database files. We do not persist any execution traces (The call graph is temporarily created during preprocessing of Call Count and Affinity Propagation correlation scores). We also store for each file all methods' start and end lines. The query engine's *locator* component uses this data to locate a method given a query $Q(file : line)$.

The TestRank query engine loads this database. It then uses the correlation data to show the developer a list of tests which might conflict with the block of code currently being edited. This list of tests is sorted in descending order of correlation scores, so the developer can run just the top tests that are most likely to be specific to the current code change. We will next provide the details of the heuristics used by each predictor.

TABLE I
TESTRANK PREDICTORS

Category	Mnemonic	Description
dynamic	EXCNT	execution count
	CLCNT	call count
	STKEXCN	stack depth sum
	VALPROP	value propagation
	ICOV	inverse coverage
natural language	AFFIN	simple affinity
	AFFPROP	affinity propagation
meta	AVG	linear combination

A. Predictors

This section describes the predictors that we have used. Table I summarized these predictors.

We denote the score of a predictor p for the relevance of a test t to a method m as $score_p(t, m)$. Coverage is used as a basic soundness filter for all predictors, meaning that if method m is never called during the execution of test t, then $score_p(t, m) = 0$. The rest of this section describes in detail the heuristics used by each predictor.

B. Dynamic Predictors

These predictors track test metrics during test suite execution.

1) Execution Count Predictor: This predictor counts the number of times that each method was executed, during the execution stemming from test t. The more often the method was executed, the more likely it is important in the flow of this test. Let $executions(t, m)$ be the set of executions of method m during test t. We define the execution count score as:

$$score_{excnt}(t, m) = |executions(t, m)|$$

2) Call Count Predictor: This predictor counts the number of distinct calls to each method, during the execution stemming from test t. For example if $f()$ is called five times, twice from g() and three times from h(), then f's execution count is 5 and its call count is 2. Distinct calls are assumed to be more important than repeated calls from the same place, which could be caused by loops. Let $distinct\_calls(t, m)$ be the set of distinct calls to method m during test t. We define the call count score as:

$$score_{clcnt}(t, m) = |distinct\_calls(t, m)|$$

3) Stack Depth Sum Predictor: This predictor sums the inverse depth of the call stack, at the executions of each method, during the execution stemming from test t. The shallower the stack, the closer the method is to the test, hence it is assumed to be more directly related to the test. Let $depth(execution_i(t, m))$ be the depth of the stack during the i'th execution of method m during test t, and let n be the number of executions of method m during test t. We define the stack depth sum score as:

$$score_{stkexcn}(t, m) = \sum_{i=1}^{n} \frac{1}{depth(execution_i(t, m))}$$

4) Value Propagation Predictor: The idea behind the Value Propagation predictor is that different tests might cover the same method with different data values. Therefore, a test passing similar values to those received in the method is assumed to be related to the method, and the same is true for the return value going out of the method and received back in the test. This predictor compares values of primitive types, wrapper types and strings. It computes the size of intersection between the two sets of values, and for each test/method pair, finds the maximum intersection c out of all the method's executions.

For each execution $e \in executions(t, m)$ of method m during test t, let $vals_e(m)$ be the set of values reaching method m, and $vals_e(t)$ be the last set of values to flow out of test t before m was called, and let $vals'_e(m)$ and $vals'_e(t)$ be the corresponding sets after adding the return value of m and the returned value to t. We define the value propagation score as:

$$score_{valprop}(t, m) = \max_{e \in executions(t,m)} \{|vals'_e(t) \cap vals'_e(m)|\}$$

5) Inverse Coverage Predictor: The idea behind the Inverse Coverage Predictor is that if test t covers method m, then the relative "importance" of m for t depends on how many other methods t covers. The fewer methods t covers, the more significant is the fact that it covers m. Let $trace(t)$ be the set of methods executed during test t. This predictor simply ranks t by increasing order of the number of methods which t covers:

$$score_{icov}(t, m) = \begin{cases} \frac{1}{|trace(t)|} & \text{if } m \in trace(t); \\ 0 & \text{otherwise.} \end{cases}$$

C. Natural Language based Predictors

Natural language program analysis techniques use Information Retrieval similarity measures, to estimate the affinity or similarity between two given segments of code [24]. The underlying assumption behind *natural language program analysis* techniques is that programmers use meaningful identifiers, in order to create a readable program. An identifier can be broken into a concatenation of meaningful words, using code conventions such as *under_lining*, or *CamelCase*. In addition English words are extracted from comments, again assuming programmers indeed write comments in their code.

TestRank uses the WordNet lexicon [26] and an adapted version of the CodePsychologist's affinity score algorithm [25] to assign each (test,method) pair a score between 0 and 1, based on the textual similarity of the word groups from the source code of the test and the method.

In contrast with TestRank, CodePsychologist is designed for use with regression testing in a system testing environment, where the test cases are not programmer tests but rather textual test plans or scripts for system test automation tools. CodePsychologist's input is not a single code change, but rather a history of the source repository. Another difference is CodePsychologist's dependence on manual configurations, and in particular relying on user supplied coverage filters, instead of automatically discovering tests coverage. In spite of these differences, TestRank borrows the same basic approach as the CodePsychologist but for a different purpose, working in "reverse psychology" mode — locating tests "looking after" methods, instead of methods that fail tests.

We made the following adaptations and improvements to CodePsychologist:

- Coverage is used as a basic soundness filter, meaning if method m is not covered by test t, then $score_{\mathit{affin}}(t, m) = 0$
- In addition to English words in the text compared using the word affinity measure, we also extracted *literals*. These include string literals, numbers and whole identifiers made of several words. Also, any word not found in WordNet is considered a literal. For literals we used simple string comparison. We also experimented with comparing literals using edit distance, as in Simpson and Dao [27]. However, the results were not conclusive so we do not show them here.
- Words appearing in the method name are given extra weight. For a constructor, which is anonymous in Java, the class name is given extra weight. The weight factor $\dot{\omega}_m(w)$ is one if the word w does not appear in the name of method m. Otherwise it is set to constant value which was determined in the tuning phase.
- We filter out stop words such as "to", "the", etc., These are extremely common words that are not significant for the affinity and only add noise and load to the system.
- In order to further reduce uninformative noise and load, we filter out the most common words in the text. The specific percentage of words left out was determined in the tuning phase.
- We give each word in a word group a weight, according to its relative importance in its context, using the Information Retreival measure TF-IDF [29]. Term frequency (TF) measures the importance of a word in a particular method (or test). Suppose a word w occurs $n_{w,m}$ times in a method m, and that there are a total of N_m distinct words in the method, then $tfidf(w, m) = \frac{n_{w,m}}{N_m}$. The inverse document frequency (IDF) measures the importance of the word among all methods. Suppose w occurs in d_w methods, and there are a total of D methods in all the traces, then $idf(w) = \log \frac{D}{d_w}$. TF-IDF balances the relative frequency of the word in a particular method with its overall frequency:

$$tfidf(w, m) = tf(w, m) \cdot idf(w) = \frac{n_{w,m}}{N_m} \cdot \log \frac{D}{d_w}$$

1) Simple Affinity Predictor: Given two collections of words: $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$, the weighted word group affinity $\mathit{GrpAff}(A, B)$ is calculated as follows:

$$\mathit{MaxAff}(\omega, B) = \max_{b \in B}\{\, \mathit{WrdAff}(\omega, b)\}$$

$$bMax_{\omega,B} = \operatorname*{argmax}_{b \in B}\{\, \mathit{WrdAff}(\omega, b)\}$$

$$\begin{aligned} \ddot{W}_A(\omega, B) \quad &= \quad tfidf(\omega, A)\dot{\omega}_A(\omega) \;\cdot \\ &\qquad tfidf(bMax_{\omega,B}, B)\dot{\omega}_B(bMax_{\omega,B}) \end{aligned}$$

$$\mathit{AsyGrpAff}(A, B) = \frac{\sum_{\omega \in A} \mathit{MaxAff}(\omega, B)\ddot{W}_A(\omega, B)}{\sum_{\omega \in A} \ddot{W}_A(\omega, B)}$$

The final group affinity and affinity score are defined the same as in the CodePsychologist [25]:

$$\mathit{GrpAff}(A, B) = [\mathit{AsyGrpAff}(A, B) + \mathit{AsyGrpAff}(B, A)]/2$$

$$score_{\mathit{affin}} = \mathit{GrpAff}(words(method), words(test))$$

2) Affinity Propagation Predictor: Suppose that test case t has high affinity to some method m. Sometimes t might fail due to a change in a method p, which calls m, even though m's code was not changed. For example, changing p might cause m to be called with different arguments, or in a different state, or even cause m not to be called at all. We may say then that p is correlated with t because it calls a method that is correlated with t. The Affinity Propagation predictor processing is done after all simple affinities have been assigned. It runs in n iterations, updating the current affinity propagation of each method p with that of its "children":

$$\begin{aligned} \mathit{affprop}_0(p) \quad &= \quad \mathit{affinity}(p) \\ \mathit{affprop}_{i+1}(p) \quad &= \quad (1 - f)\mathit{affprop}_i(p) \;+ \\ &\qquad \frac{f}{|children(p)|} \sum_{c \in children(p)} \mathit{affprop}_i(c) \end{aligned}$$

Where f is a weight factor determining how much propagation is used at each iteration. n and f were determined in the tuning phase. We define for fixed t:

$$score_{\mathit{affprop}}(p) = \mathit{affprop}_n(p)$$

D. Linear Combination

The relevance of each test t to the code change in method m is given a score by each of the predictors described above. The Linear Combination predictor attempts to combine the scores from the different predictors into one meta predictor.

We first must make sure that the scores are normalized for all predictors. In particular, the Execution Count, Call Count, Stack Depth Sum and Value Propagation predictors give nominal scores in whole numbers $0, 1, 2, 3, \cdots$. We first normalize these scores to the range $[0, 1]$, by taking:

$$s' = \frac{s}{s + 1}$$

where s is the original nominal score. Note that this normalization function is monotonic, which is necessary in order to use the scores for ranking.

We also experimented with normalizing scores using the alternative function:

$$s'(m, t) = \frac{s}{\sum_j s(m_j, t)}$$

This alternative normalization function attempts to consider $s(m, t)$ in relation to all methods m_j covered by t. However, the results were not conclusive, and due to lack of space we omit them here, and provide their details in [30].

The final score is a linear combination of all the predictor-scores:

$$score(m,t)_{avg} = \sum_{i=1}^{P} \omega_i \cdot score'_i(m)$$

where ω_i is the coefficient of the predictor i, and P is the number of predictors the algorithm uses. The values of the coefficients are based on experimental tuning described in Section IV-B.

E. Implementation

TestRank is implemented in Java and AspectJ. It uses the Java API for WordNet Searching (JAWS) [31]. TestRank can be used in any IDE and with any test runner. We used Eclipse IDE with JUnit [16].

IV. Empirical evaluation on open-source projects

The aim of this section is to evaluate the effectiveness of our techniques. The effectiveness of the change-sensitive TCP process is measured by the rank of the actual failed test in the ordered list of tests. In order to obtain an effective bounded RTS process we need to be able to set the selection size bound α low, and get a high *fault recall rate* β.

In the rest of this section we will first explain our experimental method, and then we will present results from two experiments. In order to evaluate our techniques, we used two *Apache* open source projects. The first project $Log4J$ was used for experimenting and tuning TestRank. We will also use $Log4J$ as a running example to illustrate our method. The second project *Commons Math* was used to test TestRank's effectiveness on unseen data. We conclude this section with a discussion of the results and what expected *fault recall rate* can be inferred from them.

A. Method

In both experiments we mutated methods in order to simulate a programmer accidentally introducing a bug to the code. In order to perform a meaningful comparison to safe RTS, we are most interested in the cases where safe-RTS would select a large portion of the test cases. We used TestRank's option "$-core$" to find which methods have the highest test coverage, and chose these methods as potential mutation methods (*core methods*).

In order to find suitable mutations, we used the Jumble mutation testing tool [32] on each potential class using all the tests in the suite, obtaining a list of potential mutations. The Jumble mutation operators used are detailed in table II. Further details about Jumble's mutation operators are available at the Jumble web site [33].

We also created mutations manually, based on our experience with real programs, in order to create more realistic faults, and also in cases where Jumble missed an obvious mutation, and in order to save time and increase the overall number of experiments.

TABLE II
JUMBLE MUTATION OPERATORS USED

default	Negate conditionals
default	Replace Binary Operation
-r	Mutate return values
-k	Mutate inline consts
-i	Mutate increments
-w	Mutate constant pool entries

For each method m with mutations, we determined the *fail set* — $fail(m)$, the tests that fail as a result of any mutation in the method. We then selected the methods whose *fail set* is positive and yet small as our *mutation methods*. We chose methods with high *test coverage* and a small positive *fail set*, to test that our technique performs beyond what might be expected at random. (If *test coverage* is low then safe RTS provides a small precise selection, and if *fail set* is high then safe RTS is expected to fail fast.)

Since in most cases $|fail(m)| > 1$, and in order to evaluate how high each failing test $t \in fail(m)$ is ranked independently of other failing tests in $fail(m)$, we treated each failing test separately as follows: We define a *detected mutation* as a pair (m,t), where t is a test that fails as a result of mutating the method m. After running preprocessing in order to create the TestRank database, we executed a script to perform TestRank's "-query" option on each *mutation method* m and produced $ranks(m)$ — a list of all tests in T covering m ranked by their relevance to m, as determined by a TestRank predictor. The script also records for each $t \in fail(m)$ its relative rank in $ranks(m)$ as follows: Let $ranks(m,t)$ be the same ordered list as $ranks(m)$, but excluding all failing tests except t, and let $position(m,t)$ be the index of the failing test t in this list. then the relative rank of t in the context of m is:

$$RR_m(t) = \frac{position(m,t)}{|ranks(m,t)|}$$

Thus, we obtained for each *detected mutation* (m,t) its relative rank $RR_m(t)$, the smaller—the better, but of course $\frac{1}{|ranks(m,t)|}$ is a lower bound on $RR_m(t)$. The validation script repeats this for each predictor p to obtain $RR^p_m(t)$, so we can compare the effectiveness of the different predictors by comparing the different values of $RR^p_m(t)$. In the next section, we will demonstrate our method by walking through how we used it for testing during the development process.

B. Verification During Development

In order to tune TestRank, and test it during development, we used the open source project $Log4J$, the Apache Java logging service project [34]. This project contains 32713 lines of code and 252 test cases. For convenience, we used the project's test suite *CoreTestSuite*, which contains 202 out of these test cases. The size of the TestRank database file was ~ 580 KB.

As was explained in section IV-A, methods with high *test coverage* and a small positive *fail set* are good candidates for mutation. In the case of Log4J, out of all methods m covered by at least 15 tests (*core methods*), we chose 8 *mutation methods* satisfying $0 < fail(m) < 6$. For these 8 method we obtained 24 $(method, test)$ *detected mutations*.

Table III shows the absolute rank $position(m, t)$ that each predictor prescribed to each test failure for each bug. Since each bug may cause several tests to fail, each row represents a test failure, and the rows are grouped by bugs. The *Safe RTS* column records the number of tests needed to cover each mutation using a safe RTS approach, which equals $|ranks(m, t)|$.

Looking at the table we can observe several things. Different heuristics predicted different failures. No predictor was always the best. The highest effectiveness was achieved by Affinity, Affinity Propagation and Linear Combination.

The dynamic predictors generally produced the worst results, out of which Call Count predictor was the best and Inverse Coverage was the worst. We may also notice that Execution Count was always worse than Call Count. Some predictors, which were generally not highly effective, beat all the others in a few cases. For example, Stack Depth was the only predictor to rank test #1 of bug #1 at rank #2 out of 29, and Value Propagation ranked test #1 of bug #6 at expected rank #1 or #2 out of 20 tests.[4]

Another kind of analysis we performed was relative ranks analysis. As an example, the Affinity predictor ranked the second test that detected bug #1 as #3 out of 29 covering tests for the mutated method. This means a relative rank of 10.3%. The same test was ranked at the top 22.4% by Execution Count Predictor, and its expected absolute rank is #6.5 out of 29.

Fig. 3 shows the results for the Log4J experiment in a box plot.[5] The diagram shows for each predictor, the distribution of the predictor's detection-effectiveness scores. On each *detected mutation* (m, t) the detection-effectiveness score is defined as $1 - RR_m(t)$.

The linear combination was tuned at this stage to maximize the results. The optimal combination found is 85% Affinity Propagation, 8% Call Count, 5% Value Propagation and 2% Stack Depth. Out of all the *covering* tests, TestRank's Linear Combination predictor (AVG) ranked the failing test in the top 23.8%, in 70% of the cases (out of 24 detected mutations), and in the top 30.8% in 80% of the cases.

Fig. 2.A shows the results of the Linear Combination predictor for the Log4J experiment in a Pareto chart. For each relative rank cutoff, the chart shows in how many cases the predictor ranked the detecting test lower than that cutoff value. In bounded RTS terms, the chart shows the fault detection rate β as a function of the selection-size bound α.[6]

C. Validation Experiment

In order to evaluate TestRank effectiveness on unseen test data, we used the open source project *Commons Math*, the *Apache Commons* mathematics library [35]. This project contains 42212 lines of code and 1882 test cases (in 152 test classes). The size of the TestRank database file was ~ 11.4 MB.

As was explained in section IV-A, methods with high *test coverage* and a small positive *fail set* are good candidates for mutation. In the case of Commons Math, out of all 61 methods m covered by at least 90 tests (*core methods*), we chose 8 *mutation methods* satisfying $0 < fail(m) < 10$. For these 8 method we obtained 38 $(method, test)$ *detected mutations*.

Fig. 4 shows the results for the Commons Math experiment in a box plot.

Comparing the results to those of the previous experiment, we can observe that Affinity and Affinity Propagation effectiveness is now worse than on the tuning data, while all the dynamic predictors, except Call Count, have improved. The Value Propagation predictor was the best dynamic predictor, ranking the failed tests in its top 45%, in 80% of the cases.

Further analysis reveals that the Linear Combination predictor (AVG), as tuned in the previous experiment, performs as the best predictor in the current experiment, on several statistics including average, median and 70% percentile. Out of all the *covering* tests, TestRank's Linear Combination predictor (AVG) ranked the failing test in the top 30.5%, in 70% of the cases (out of 38 detected mutations), and in the top 51.5% in 80% of the cases. Fig. 2.B shows the results of the Linear Combination predictor for the Commons Math experiment in a Pareto chart.

D. Threats to Validity

Like any empirical evaluation, this work has limitations that must be considered when interpreting the results. Threats to internal validity mostly concern possible errors in the way we implemented our techniques that could affect outcomes. To control for this threat, we verified parts of the implementations on synthetic examples, and performed several sanity checks. For the purpose of initial evaluation, we approximated a test suite's execution cost by the number of test cases in the test suite. The more realistic cost measure is of course the running time of the test suite. We expect to collect the running time for each test case and improve the accuracy of our results in the near future, but we expect that on average this approximation should not bias the results much.

[4] It is possible for a predictor to give the same score to more than one test, which is often the case with dynamic predictors. In this case, the internal order between tests with the same rank would be arbitrary. In order to deal with this phenomenon, we designed the validation script to order the actual failing test according to the expected-case of a random internal order. The validation script thus calculates the rank of a test as the average between the two extreme cases, where the test is first or last in its "equivalence class" of tests. That is why we might often see non integer numbers like 6.5 in TestRankValidator's relative ranks output.

[5] Box plots provide a concise display of a distribution. The boxes display the median, first and third quartiles. The whiskers stretch to the min and max values within 1.5 box-ranges from the box. Min and max outliers are shown as '$*$'.

[6] This is somewhat similar to average precentage of faults detected, or APFD [7]. The difference is that an APFD graph describes a single experiment, showing how fast one general prioritization is detecting multiple bugs, whereas our graph describes a distribution over many experiments, showing the accumulated effectiveness of many change-specific prioritizations in detecting a single bug each (In how many experiments the bug was detected in the top x%).

TABLE III
LOG4J EXPERIMENT RESULTS — EXPECTED ABSOLUTE RANKS[A]

Bug	Safe RTS	EXCNT	CLCNT	STKEXCN	VALPROP	ICOV	AFFIN	AFFPROP	AVG
1	29	6.5	2.5	2	15	29	3	3	3
		20	16	21.5	15	29	3	3	4
2	21	5	4.5	3.5	7	5	8	6	5
		5	4.5	3.5	3.5	5	8	6	5
3	17	14	9	16	2.5	13.5	2	2	2
		14	9	13	10	9	3	3	3
		14	9	13	2.5	9	2	2	2
		14	9	13	10	9	3	3	3
		14	9	13	10	9.5	2	2	2
4	17	9.5	9	9.5	9	17	1	1	1
5	17	9.5	9	9.5	9	17	2	2	2
6	20	16.5	10.5	14.5	1.5	16.5	9	9	6
		16.5	10.5	14.5	10.5	18	9	9	10
		16.5	10.5	14.5	10.5	11	10	10	12
		16.5	10.5	14.5	10.5	11.5	10	10	11
		16.5	10.5	14.5	10.5	12.5	9	9	10
7	12	10.5	6.5	10.5	10.5	12	4	4	8
		10.5	6.5	10.5	5	7.5	1	1	1
		10.5	6.5	10.5	5	7	1	1	1
		10.5	6.5	10.5	5	7	1	1	1
8	13	6.5	3.5	4.5	9	13	3	3	4
		3	1.5	2.5	9	7.5	2	2	3
		1.5	1	1	9	7	2	2	1
		3	1.5	2.5	9	7	3	3	3

[A] Rows represent ranks of failed tests, grouped by the bugs causing the tests to fail.

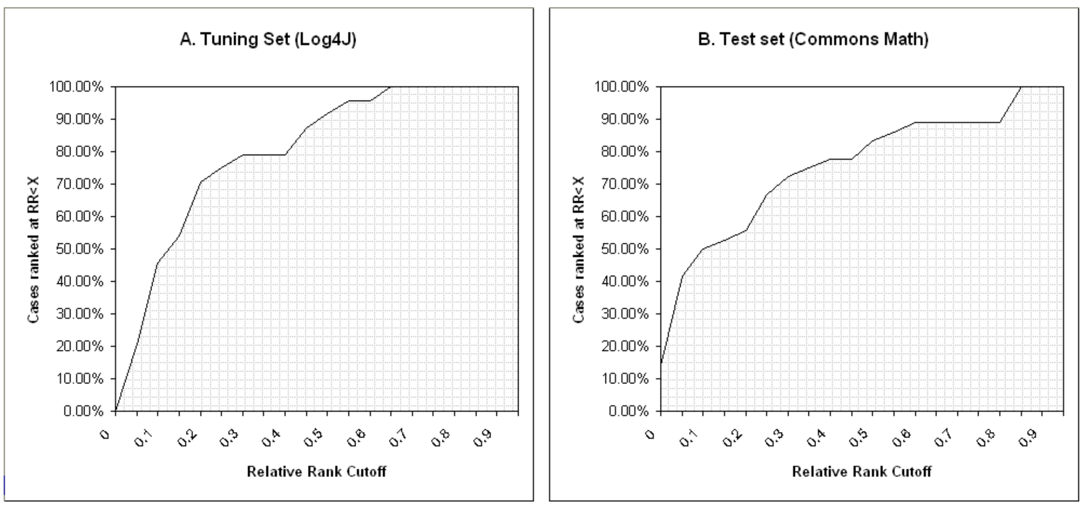

Fig. 2. Fault-detection rate vs. fraction of test suite used

The main threat to external validity is that our case study included only two programs, and therefore we cannot claim that these results generalize to other programs. Like many other projects, which utilize automated tests, it is hard to say how much the studied projects follow the TDD philosophy. However, the programs are real software systems, and the test suites used are the ones actually used by the developers of the considered systems.

Another threat to the ability to use our tool in various code bases is that, in our current implementation, preprocessing of CPU intensive code is slow. This slowness is an artifact of a design decision we made, which simplified the construction of the prototype implementation, and we may improve on it in the future. In any event, preprocessing is performed off-line, so developers using the tool should not experience any overhead.

E. Discussion

Our validation experiment (*Commons Math* experiment) confirms that the results obtained during the first experiment (*Log4J* experiment) do not deteriorate too much when validating the system on an unseen code base. The improvement we can expect in relation to safe-RTS, is about 70/30 or 80/50.

What are the expected improvements in a typical TDD environment, in relation to not using any selection technique? In order to answer this question, we need to asses the typical distribution of test coverage in TDD projects. This data is a subject for further research. For the purpose of this discussion, we will just illustrate using a hypothetical example: Suppose we have an application with 1000 tests, in which 60% of the methods are covered by 100 tests each, 25% of the methods are

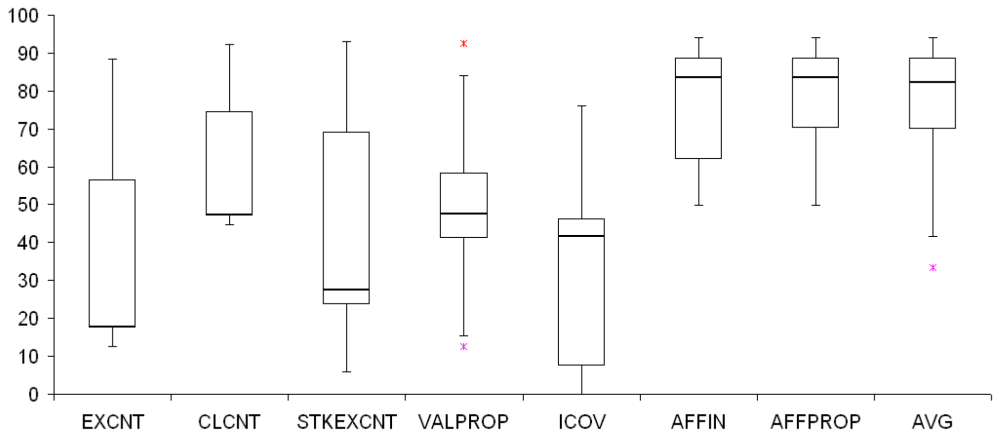

Fig. 3. Log4J Experiment Results boxplots — detection-effectiveness by predictor

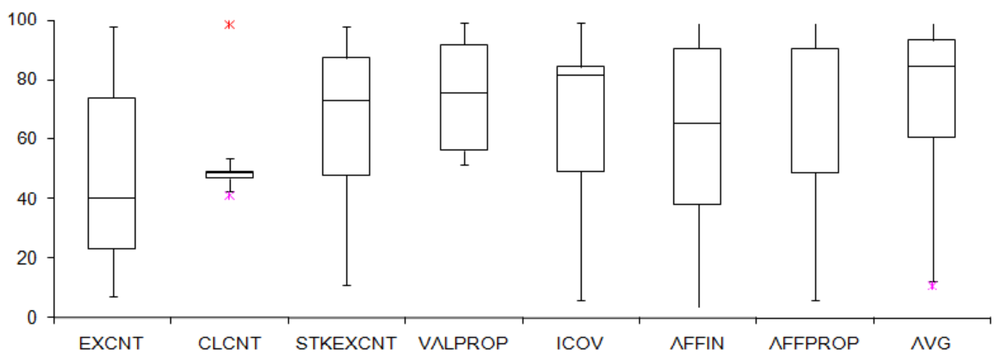

Fig. 4. Commons Math Experiment Results boxplots — detection-effectiveness by predictor

covered by 200 tests each, and another 15% of the method by 300 tests each. Suppose further that we configure TestRank's bound to the top 100 tests. Then for 60% of the methods we run all their covering tests and hence get 100% bug detection, for 25% of the methods we select 50% of their covering tests and expect to get 80% bug detection, and for another 15% of the method we select 33% of their covering tests each, and expect to get 70% bug detection. We obtain the following expected fault recall rate:

$$0.6 \cdot 1.0 + 0.25 \cdot 0.8 + 0.15 \cdot 0.7 = 0.905$$

So, in this case TestRank is expected to find 90% of the bugs, while running only 10% of the tests after any code change.[7]

V. Conclusions and Future Work

We have introduced TestRank, a bounded regression test selection tool, using a combination of several change-sensitive test case prioritization techniques. We based our techniques on dynamic and natural language analyses of a code base and its test suite. Our experimental results with Log4J and Commons Math support our assumption that the technique works, and gives added value to developers in accelerating retesting during development, while maintaining a high bug detection rate. TestRank gives the TDD developer a good bug detection tool, while considerably accelerating TDD testing. Finally, while our approach is particularly suitable for TDD, we believe it may be beneficial in any setting which emphasizes frequent testing during development, a practice that is becoming more widespread in the software industry.

This paper presented the first experience with TestRank, which was limited to two medium sized projects. We would like to do further empirical evaluation on larger projects, ideally in a real world TDD work environment. Further research is still needed into the alternative normalization scheme presented in section III-D, and how to apply it to affinity ranking. Combining the different predictors in an optimal manner is still another subject that has room for improvement. In addition, in the future we want to combine some improvements, part of which were found to be effective in previous works:

- Handling changes outside methods, such as overriding method introduction, having a non-local effect [10].
- Ranking tests by relevance to multiple code changes [13], expecting to improve the ranking precision.
- Using fine granularity tracing, i.e., basic blocks instead

[7] Of course, in order to confirm this hypothetical result, further validation is needed on various code bases.

of methods, thus potentially improving precision.

- Combining global ranking. Prefer tests that are shortest, most recently failed or frequently failed, least recently executed [12] or that are complex.[8]
- Setting in each rank a test that gives *additional* value to whatever value was covered by the preceding tests.
- Reinforce a correlation score when a test actually fails after a method changed, and vice versa.
- Use mutation analysis to record which tests fail after mutating a code fragment [7].
- Annotation syntax for the developer to provide hint tags to tests and code segments.
- Continuing the work on comparing *literals*, using edit distance measures. Handling abbreviations.
- Consider changes in resources outside code (e.g., XML).
- Integration with real world environments like Eclipse.
- Filtering changes in comments, refactoring, dead code, etc.

On a more general level, we introduced a technique for estimating test-to-code correlation in TDD code bases. We have focused on the application for test selection, but once the test-to-code correlation is revealed, there can be further interesting applications such as root cause analysis of bugs ("CodePsychologist for TDD"), code comprehension (understanding a piece of code's functionality, by finding the test that it implements), change sensitivity analysis, test maintenance (find tests that will need to change) and TDD implementation assistance (find where to implement a changed test).

In addition, combining text-to-code or text-to-test correlation for free text, can yield such applications as finding relevant code to ticket in bug tracking system, finding potential regression to known past tickets and feature location by keywords.

REFERENCES

[1] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[2] "JUnit FAQ," http://junit.sourceforge.net/doc/faq/faq.htm.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, July 1999. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201485672

[4] Y. F. Chen, D. S. Rosenblum, and K. P. Vo, "Testtube: A system for selective regression testing," in *International Conference on Software Engineering*, 1994, pp. 211–220. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.353

[5] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, April 1997. [Online]. Available: http://dx.doi.org/10.1145/248233.248262

[6] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2004, pp. 241–251. [Online]. Available: http://dx.doi.org/10.1145/1029894.1029928

[7] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2000, pp. 102–112.

[8] But Elbaum et al. (2000) [7] suggests that complexity ranking does not significantly improve prioritization effectiveness.

[8] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2006, pp. 1–12.

[9] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 149–183, 2001.

[10] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: a change impact analysis tool for Java programs," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 664–665. [Online]. Available: http://dx.doi.org/10.1145/1062455.1062598

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.5403

[12] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 119–129.

[13] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2002, pp. 97–106.

[14] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34 –41, apr. 1978.

[15] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," *Fifteenth International Symposium on Software Reliability Engineering*, pp. 113–124, 2004.

[16] "JUnit," http://www.junit.org.

[17] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *Fourteenth International Symposium on Software Reliability Engineering*, Denver, CO, November 17–20, 2003, pp. 281–292.

[18] "Infinitest," http://improvingworks.com/products/infinitest.

[19] "Clover test optimization," http://www.atlassian.com/software/clover/tour/code-coverage-test-optimization.jsp.

[20] K. Beck, "JUnit Max," http://www.junitmax.com.

[21] "Google Testar," http://google-testar.sourceforge.net.

[22] "JTestMe," http://xircles.codehaus.org/projects/jtestme.

[23] "ProTest," http://sourceforge.net/projects/protest.

[24] L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor, "Introducing natural language program analysis," in *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2007, pp. 15–16.

[25] D. Nir, S. S. Tyszberowicz, and A. Yehudai, "Locating regression bugs," in *Haifa Verification Conference*, 2007, pp. 218–234.

[26] C. D. Fellbaum, *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press, 1998.

[27] T. Simpson and T. Dao, "WordNet-based semantic similarity measurement," http://www.codeproject.com/KB/string/semanticsimilaritywordnet.aspx.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.

[29] G. Salton and M. Mcgill, *Introduction to Modern Information Retrieval (Computer Science S.)*. McGraw-Hill Education, August 1983. [Online]. Available: http://www.worldcat.org/isbn/0070665265

[30] H. Cibulski, "Regression Test Selection Techniques for Test-Driven Development," Master's thesis, Tel-Aviv University, Israel, 2011, in preparation.

[31] B. Spell, "Java API for WordNet Searching (JAWS)," http://lyle.smu.edu/~tspell/jaws.

[32] S. A. Irvine, P. Tin, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble Java byte code to measure the effectiveness of unit tests," in *Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques*, 2007, pp. 169–175.

[33] "Jumble Mutations," http://jumble.sourceforge.net/mutations.html.

[34] "Apache Log4J," http://logging.apache.org/log4j/1.2/.

[35] "Apache Commons Math," http://commons.apache.org/math.