SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Prioritizing the Creation of Unit Tests in Legacy Software Systems

Emad Shihab[†*], Zhen Ming Jiang[†], Bram Adams[†], Ahmed E. Hassan[†] and Robert Bowerman[§]

Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, ON, Canada[†]
Research In Motion, Waterloo, ON, Canada[§]

SUMMARY

Test-Driven Development (TDD) is a software development practice that prescribes writing unit tests before writing implementation code. Recent studies have shown that TDD practices can significantly reduce the number of pre-release defects. However, most TDD research thus far has focused on new development. We investigate the adaptation of TDD-like practices for already-implemented code, in particular legacy systems. We call such an adaptation "Test-Driven Maintenance" (TDM).
In this paper, we present a TDM approach that assists software development and testing managers to use the limited resources they have for testing legacy systems efficiently. The approach leverages the development history of a project to generate a prioritized list of functions that managers should focus their unit test writing resources on. The list is updated dynamically as the development of the legacy system progresses. We evaluate our approach on two large software systems: a large commercial system and the Eclipse Open Source Software system. For both systems, our findings suggest that heuristics based on the function size, modification frequency and bug fixing frequency should be used to prioritize the unit test writing efforts for legacy systems. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Unit testing, Testing legacy systems, Test driven maintenance

1. INTRODUCTION

Test-Driven Development (TDD) is a software development practice where developers write and run unit tests that would pass once the requirements are implemented. Then they implement the requirements and re-run the unit tests to make sure they pass [1, 2]. The unit tests are generally written at the granularity of the smallest separable module, which is a function in most cases [3].

Previous empirical studies have shown that TDD can reduce pre-release defect densities by as much as 90%, compared to other similar projects that do not implement TDD [4]. In addition, other studies show that TDD helps produce better quality code [5, 6], improve programmer productivity [7] and strengthen the developer confidence in their code [8].

Most of the previous research to date studied the use of TDD for new software development. However, prior studies show that more than 90% of the software development cost is spent on maintenance and evolution activities [9, 10]. Other studies showed that an average Fortune 100 company maintains 35 million lines of code and that this amount of maintained code is expected to double every 7 years [11]. For this reason, we believe it is extremely beneficial to study the

*Correspondence to: 156 Barrie Street, Kingston, Ontario, K7L 3N6, email: emads@cs.queensu.ca

adaptation of TDD-like practices for the maintenance of already implemented code, in particular for legacy systems. In this paper we call this "Test-Driven Maintenance" (TDM).

Applying TDM to legacy systems is important because legacy systems are often instilled in the heart of newer, larger systems and continue to evolve with new code [12]. In addition, due to their old age, legacy systems lack proper documentation and become brittle and error-prone over time [13]. Therefore, TDM should be employed for these legacy systems to assure quality requirements are met and to reduce the chance of failures due to evolutionary changes.

However, legacy systems are typically large and writing unit tests for an entire legacy system at once is time consuming and practically infeasible. To mitigate this issue, TDM uses the same divide-and-conquer idea of TDD. However, instead of focusing on a few tasks from the requirements documents, developers that apply TDM isolate functions of the legacy system and individually unit test them. Unit tests for the functions are incrementally written until a desired quality target is met.

The idea of incrementally writing unit tests is very practical and has three main advantages. First, it gives resource-strapped managers some breathing room in terms of resource allocation (i.e., it alleviates the need for long-term resource commitments). Second, developers can get more familiar with the legacy code through the unit test writing efforts [14], which may ease future maintenance efforts. Third, unit tests can be easily maintained and updated in the future to assure the high quality of the legacy system [3].

The major challenge for TDM is determining how to *prioritize* the writing of unit tests to achieve the best return on investment. Do we write unit tests for functions in a random order? Do we write unit tests for the functions that we worked on most recently? Often, development and testing teams end up using ad hoc practices, based on gut feelings, to prioritize the unit test writing efforts. However, using the right prioritization strategy can save developers time, save the organization money and increase the overall product quality [15, 16].

This paper extends an earlier conference paper [46], in which we presented an approach that prioritizes the writing of unit tests for legacy software systems, based on different history-based heuristics. We evaluated our approach on a commercial system, determined the most effective heuristics and investigated the effect of various simulation parameters on our study results. In this paper, we extend our previous work by conducting our study on an additional Open Source system. Doing so reduces the threat to external validity and improves the generalizability of our findings since both systems follow different development practices (i.e., commercial vs. open source), come from different domains (i.e., communication system vs. integrated development environment) and are written in different programming languages (i.e., C/C++ vs. Java). In addition, instead of only considering heuristics individually, we combine them in two ways and compare the performance using the combined heuristics and the individual heuristics.

For both systems, our results show that the proposed heuristics significantly improve the testing efforts, in terms of potential bug detection, when compared to random test writing. Heuristics that prioritize unit testing effort based on function size, modification frequency and bug fixing frequency are the best performing for both systems. Combining the heuristics improves the performance of some heuristics, but did not outperform our best performing heuristics.

Organization of Paper. Section 2 provides a motivating example for our work. Section 3 details our approach. Section 4 describes the simulation-based case study. Section 5 presents the results of the case study. Section 6 details two techniques used to combine the heuristics and presents their results. Section 7 discusses the effects of the simulation parameters on our results. Section 8 presents the list of threats to validity and Section 9 discusses the related work. Section 10 concludes the paper.

2. MOTIVATING EXAMPLE

In this section, we use an example to motivate our approach. Lindsay is a software development manager for a large legacy system that continues to evolve with new code. To assure a high level of quality for the legacy system, Lindsay's team employs TDM practices. Using TDM, the team isolates functions of the legacy system and writes unit tests for them. However, deciding which functions to write unit tests for is a challenging problem that Lindsay and his team must face.

DOI: 10.1002/spe

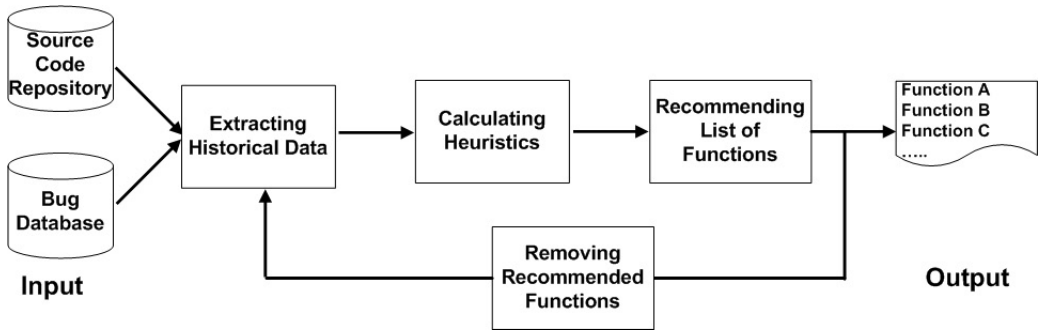


Figure 1. Approach overview

Writing unit tests for all of the code base is nearly impossible. For example, if a team has enough resources to write unit tests to assess the quality of 100 lines of code per day, then writing unit tests for a 1 million lines of code (LOC) system would take over 27 years. At the same time, the majority of the team is busy with new development and maintenance efforts. Therefore, Lindsay has to use his resources effectively in order to obtain the best return on his resource investment.

A primitive approach that Lindsay tries is to randomly pick functions and write unit tests for them or write tests for functions that have been recently worked on. However, he quickly realizes that such an approach is not very effective. Some of the recently worked on functions are rarely used later, while others are so simple that writing unit tests for them is not a priority. Lindsay needs an approach that can more effectively assist him and his team prioritize the writing of unit tests for a legacy system.

To assist development and testing teams like Lindsay's, we present an approach that uses the history of the project to prioritize the writing of unit tests. The approach uses heuristics extracted from the project history to recommend a prioritized list of functions to write unit tests for. The size of the list can be customized based on the amount of available resources at any specific time. The approach updates its recommended list of functions as the project progresses.

3. APPROACH

In this section, we detail our approach, which is outlined in Figure 1. In a nutshell, the approach extracts a project's historical data from its code and bug repositories, calculates various heuristics and recommends a prioritized list of functions to write unit tests for. Once the unit tests are written, we remove the recommended functions, re-extracts new data from the software repositories to consider new development activity and repeats the process of calculating heuristics and generating a list of functions. In the next four subsections, we describe each phase in more detail.

3.1. Extracting Historical Data

The first step of the approach is to extract historical data from the project's development history. In particular, we combine source code modification information from the source code control system (e.g., SVN [17] and CVS [18]) with bug data stored in the bug tracking system (e.g., Bugzilla [19]). Each modification record contains the time of the modification (day, month, year and local time), the author, the changed files, the version of the files, the changed line numbers and a modification record log that describes the change.

In order to determine whether a modification is a bug fix, we used a lexical technique to automatically classify modifications [20–22]. The technique searches the modification record logs, which are stored in the source code repository, for keywords, such as "bug" or "bugfix", and bug identifiers (used to search the bug database) to do the classification. If a modification record contains a bug identifier or one of the keywords associated with a bug, then it is classified as a bug fixing modification. In some cases, the modification record logs contain a bug identifier only. In this case,

Prepared using speauth.cls
DOI: 10.1002/spe

we automatically fetch the bug report's type and classify the modification accordingly. We check the bug report's type, because in certain cases bug reports are used to submit feature enhancements instead of reporting actual bugs. Eventually, our technique groups modification records into two main categories: bug fixing modifications and general maintenance modifications.

The next step involves mapping the modification types to the actual source code functions that changed. To achieve this goal, we identify the files that changed and their file version numbers (this information is readily available in the historical modification log). Then, we extract the source code versions of the files that changed and their previous version, parse them to identify the individual functions, and compare the two consecutive versions of the files to identify which functions changed. Since we know which files and file versions were changed by a modification, we can pinpoint the functions modified by each modification. We annotate each of the changed functions with the modification type.

To better illustrate this step, we use the example shown in Figure 2. Initially, change 1 commits the first version of files X and Y. There are 3 bugs (italicized) in the committed files, one in each of the functions `add`, `subtract` and `divide`. Change 2 fixes these bugs. We would determine that change 2 is a bug fix from the change log message and comparing versions 1 and 2 of files X and Y would tell us that functions `add`, `divide` and `subtract` changed. Therefore, we would record 1 bug fix change to each of these functions. Thus far, function `multiply` did not change, therefore it will not have anything recorded against it. In change 3, comments are added to functions `subtract` and `multiply`. By the end of change 3, function `add` would have 1 bug fixing change, function `subtract` will have 1 bug fix change and 1 enhancement change, function `divide` will have 1 bug fix change, and function `multiply` will have 1 enhancement change.

Although the use of software repositories (i.e., source code control systems and bug tracking systems) is becoming increasingly popular in software projects, there still exist some issues with using data from such repositories. For example, developer may forget to mention the bug number that a change fixes. And even if they do include the bug numbers, in certain cases, the bug mentioned in the change description could refer to a bug that was created after the change itself or the bug mentioned is missing from the bug database altogether [49]. These issues may introduce bias in the data [50], however, recent work showed that the effect of such bias does not significantly affect the outcome of our findings [51].

3.2. Calculating Heuristics

We use the extracted historical data to calculate various heuristics. The heuristics are used to generate the prioritized list of functions for testing. We choose to use heuristics that can be extracted from a project's history for two main reasons: 1) legacy systems ought to have a very rich history that we can use to our advantage and 2) previous work in fault prediction showed that history based heuristics are good indicators of future bugs (e.g., [23, 24]). We conjecture that heuristics used for fault prediction will perform well, since ideally we want to write unit tests for the functions that have bugs in them.

The heuristics fall under four main categories: modification based heuristics, bug fix based heuristics, size based heuristics and risk based heuristics. The heuristics are listed in Table I. We also include a random heuristic that we use as a baseline heuristic to compare the aforementioned heuristics to. For each heuristic, we provide a description, our intuition for using the heuristic and related work that influenced our decision to study the heuristics.

The heuristics listed in Table I are a small sample of the heuristics that can be used to generate the list of functions. We chose to use these heuristics since previous work on fault prediction has proven their ability to perform well. However, any metric that captures the characteristics of the legacy system and can be linked to functions may be used to generate the list of functions.

3.3. Generating a List of Functions

Following the heuristic calculation phase, we use the heuristics to generate a prioritized list of functions that are recommended to have unit tests written for. Each heuristic generates a different prioritized list of functions. For example, one of the heuristics we use (i.e., MFM) recommends that

DOI: 10.1002/spe

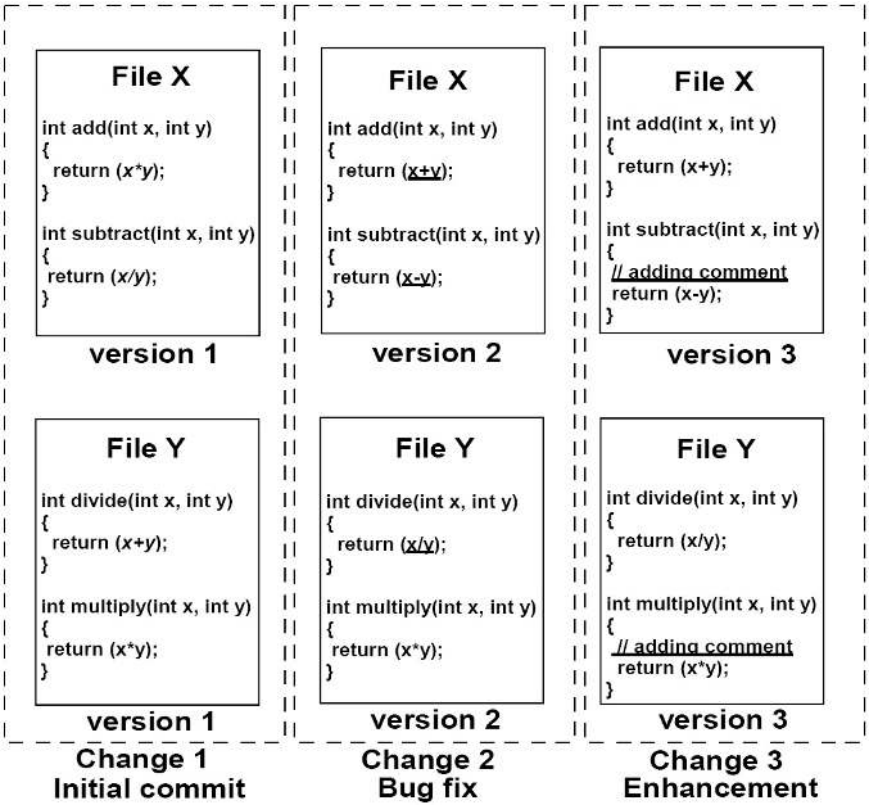


Figure 2. Linking changes to functions

we write tests for functions that have been modified the most since the beginning of the project. Another heuristic recommends that we write tests for functions that are fixed the most (i.e., MFF).

Then, we loop back to the historical data extraction phase, to include any new development activity and run through the heuristic calculation and list generation phases. Each time, a new list of functions is generated for which unit tests should be written.

3.4. Removing Recommended Functions

Once a function is recommended to have a test written for it, we remove it from the pool of functions that we use to generate future lists. In other words, we assume that once a function has had a unit test written for it, it will not need to have any additional new test written for it in the future; at most the test may need to be updated. We make this assumption for the following reason: once the function is recommended and the initial unit test is written, then this initial unit test will make sure all of the function's current code is tested. Also, since the team adopts TDM practices any future additions/changes to the function will be accompanied with changes to the associated unit tests.

4. SIMULATION-BASED CASE STUDY

To evaluate the performance of our approach, we conduct a simulation-based case study on two large software systems: a commercial system and the Eclipse OSS system. The commercial software system is a legacy system, written in C and C++, which contains tens of thousands of functions totalling hundreds of thousands of lines of code. We used 5.5 years of the system's history to conduct our simulation, in which over 50,000 modifications were analyzed. We cannot disclose any more details about the studied system for confidentiality reasons.

DOI: 10.1002/spe

Table I. List of all heuristics used for prioritizing the test writing efforts for legacy systems

Category	Heuristic	Order	Description	Intuition	Related Work
Modifications	Most Frequently Modified (MFM)	Highest to lowest	Functions that were modified the most since the start of the project.	Functions that are modified frequently tend to decay over time, leading to more bugs.	The number of prior modifications to a file is a good predictor of its future bugs [23, 25–27].
	Most Recently Modified (MRM)	Latest to oldest	Functions that were most recently modified.	Functions that were modified most recently are the ones most likely to have a bug in them (due to the recent changes).	More recent changes contribute more bugs than older changes [25].
Bug Fixes	Most Frequently Fixed (MFF)	Highest to lowest	Functions that were fixed the most since the start of the project.	Functions that are frequently fixed in the past are likely to be fixed in the future.	Prior bugs are a good indicator of future bugs [28].
	Most Recently Fixed (MRF)	Latest to oldest	Functions that were most recently fixed.	Functions that were fixed most recently are more likely to have a bug in them in the future.	The recently fixed heuristic has been used to prioritize buggy subsystems [22], files and functions [45].
Size	Largest Modified (LM)	Largest to smallest	The largest modified functions, in terms of total lines of code (i.e, source, comment and blank lines of code).	Large functions are more likely to have bugs than smaller functions.	The simple lines of code metric correlates well with most complexity metrics (e.g., McCabe complexity) [25, 27, 29, 30].
	Largest Fixed (LF)	Largest to smallest	The largest fixed functions, in terms of total lines of code (i.e, source, comment and blank lines of code).	Large functions that need to be fixed are more likely to have more bugs than smaller functions that are fixed less.	The simple lines of code metric correlates well with most complexity metrics (e.g., McCabe complexity) [25, 27, 29, 30].
Risk	Size Risk (SR)	Highest to lowest	Riskiest functions, defined as the number of bug fixing changes divided by the size of the function in lines of code.	Since larger functions may naturally need to be fixed more than smaller functions, we normalize the number of bug fixing changes by the size of the function. This heuristic will mostly point out relatively small functions that are fixed a lot (i.e., have high defect density).	Using relative churn metrics performs better than using absolute values when predicting defect density [31].
	Change Risk (CR)	Highest to lowest	Riskiest functions, defined as the number of bug fixing changes divided by the total number of changes.	The number of bug fixing changes normalized by the total number of changes. For example, a function that changes 10 times in total and out of those 10 times 9 of them were to fix a bug should have a higher priority to be tested than a function that changes 10 times where only 1 of those ten is a bug fixing change.	Using relative churn metrics performs better than using absolute values when predicting defect density [31].
Random	Random	Random	Randomly selects functions to write unit tests for.	Randomly selecting functions to test can be thought of as a base line scenario. Therefore, we use the random heuristic's performance as a base line to compare the performance of the other heuristics to.	Previous studies on test prioritization use a random heuristic to compare their performance [15, 16, 32].

On the other hand, the Eclipse OSS system is an Integrated Development Environment (IDE), written in Java. We analyzed a total of 48,718 files, which contained 314,910 functions over their

DOI: 10.1002/spe

history. Again, we used 5.5 years of the system's history to conduct our simulation, in which 81,208 modifications were analyzed, of which 12,332 were buggy changes.

In this section, we detail the steps of our case study and introduce our evaluation metrics.

4.1. Simulation study

The simulation ran in iterations. For each iteration we: 1) extract the historical data, 2) calculate the heuristics, 3) generate a prioritized list of functions, 4) measure the time it takes to write tests for the recommended list of functions and 5) remove the list of functions that were recommended. Then, we advance the time (i.e., we account for the time it took to write the unit tests) and do all of the aforementioned steps over again.

Step 1. We used 5.5 years of historical data from the commercial and OSS systems to conduct our simulation. The first 6 months of the project were used to calculate the initial set of heuristics and the remaining 5 years are used to run the simulation.

Step 2. To calculate the heuristics, we initially look at the first 6 months of the project. If, for example, we are calculating the MFM heuristic, we would look at all functions in the first 6 months of the project and rank them in descending order based on the number of times they were modified during that 6 month period. The amount of history that we consider to calculate the heuristics increases as we advance in the simulation. For example, an iteration 2 years into the simulation will use 2.5 years (i.e., the initial 6 months and 2 years of simulation time) of history when it calculates the heuristics.

Step 3. Next, we recommend a prioritized list of 10 functions that should have unit tests written for them. One list is generated for each of the heuristics. The list size of 10 functions is an arbitrary choice we made. If two functions have the same score, we randomly choose between them. We study the effect of varying the list size on our results in detail in Section 7. Furthermore, in our simulation, we assume that once a list of 10 functions is generated, tests will be written for all 10 functions before a new list is generated.

Step 4. Then, we estimate the time it takes to write tests for these 10 functions. To do so, we use the size of the recommended functions and divide by the available resources. The size of the functions is used as a measure for the amount of effort required to write unit tests for those 10 functions [33, 34]. Since complexity is highly correlated with size [25], larger functions will take more effort/time to write unit tests for.

The number of available resources is a simulation parameter, expressed as the number of lines of code that testers can write unit tests for in one day. For example, if one tester is available to write unit tests for the legacy system and that tester can write unit tests for 50 lines of code per day, then a list of functions that is 500 lines will take him 10 days. In our simulation, we set the total test writing capacity of the available resources to 100 lines per day. We study the effect of varying the resources available to write unit tests in more detail in Section 7.

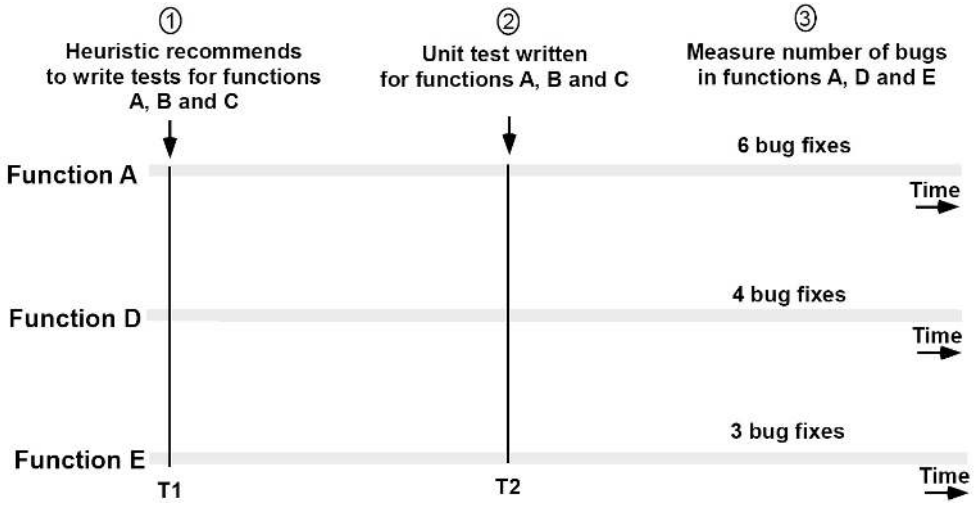
After we calculate the time it takes to write unit tests for the functions that we recommend, we update the heuristics with the new historical information. We do this to account for the continuous change these software systems undergo. For example, if we initially use six months to calculate the heuristics and the first list of recommended functions takes one month to write unit tests for, then we update the heuristics for the next iteration to use seven months of historical information. We use the updated heuristics to calculate the next set of functions to write tests for and so on.

Step 5. Once a function is recommended to have a test written for it, we remove it from the pool of functions that we use to generate future lists. In other words, we assume that once a function has had a unit test written for it, it will not need to have a new test written for it from scratch in the future; at most the test may need to be updated.

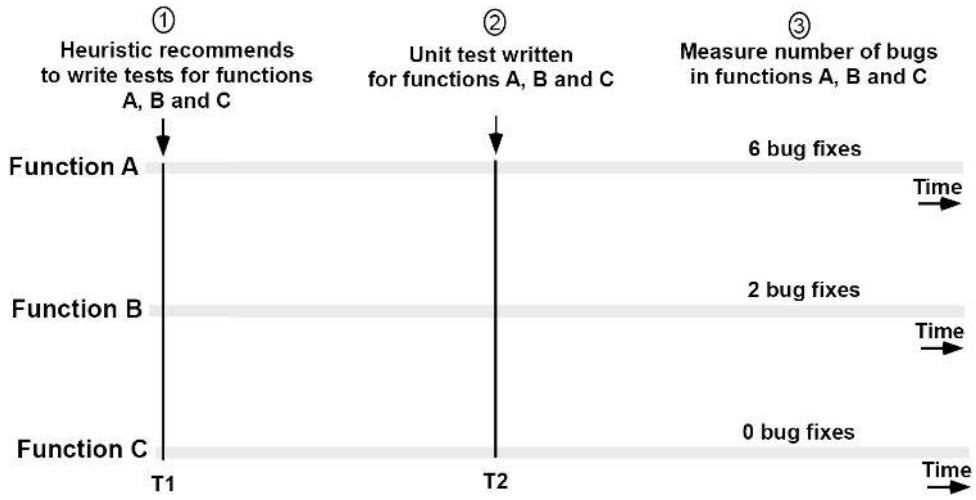
Once the parameters are set, the simulation is entirely automated. Any of the parameters can be modified at any time, however, there is no need for any manual work after the initial setup is done.

We repeat the 5-step process mentioned above for a period of 5 years. To evaluate the performance of the different heuristics, we periodically (every 3 months) measure the performance using two metrics: Usefulness and Percentage of Optimal Performance (POP), which we describe next.

DOI: 10.1002/spe



(a) Usefulness evaluation example



(b) POP example

Figure 3. Performance evaluation example

4.2. *Performance Evaluation Metrics*

Usefulness: The first question that comes up after we write unit tests for a set of functions is - was writing the tests for these functions worth the effort? For example, if we write unit tests for functions that rarely change and/or have no bugs after the tests are written, then our effort may be wasted. Ideally, we would like to write unit tests for the functions that end up having bugs in them.

We define the usefulness metric as *the percentage of functions for which we write unit tests that catch at least one bug after the tests are written*. The usefulness metric indicates how much of our effort on writing unit tests is actually worth the effort. This metric is similar to the hit rate metric used by earlier dynamic defect prediction studies [22, 45].

We use the example in Figure 3(a) to illustrate how we calculate the usefulness. Functions A and B have more than 1 bug fix after the unit tests were written for them (after point 2 in Figure 3(a)). Function C did not have any bug fix after we wrote the unit test for it. Therefore, for this list of three functions, we calculate the usefulness as $\frac{2}{3} = 0.666$ or 66.6%.

DOI: 10.1002/spe

Percentage of Optimal Performance (POP): In addition to calculating the usefulness, we would like to measure how well a heuristic performs compared to the optimal (i.e., the best we can ever do). Optimally, one would have perfect knowledge of the future and write unit tests for functions that are the most buggy. This would yield the best return on investment.

To measure how close we are to the optimal performance, we define a metric called Percentage of Optimal Performance (POP). To calculate the POP, we generate two lists: one is the list of functions generated by a heuristic and the second is the optimal list of functions. The optimal list contains the functions with the most bugs from the time the unit tests were written till the end of the simulation. Assuming that the list size is 10 functions, we calculate the POP as the number of bugs in the top 10 functions (generated by the heuristics), divided by the number of bugs contained in the top 10 optimal functions. Simply put, the POP is *the percentage of bugs we can avoid using a heuristic compared to the best we can do if we had perfect knowledge of the future*.

We illustrate the POP calculation using the example shown in Figure 3(b). At first, we generate a list of functions that we write unit tests for using a specific heuristic (e.g., MFM or MFF). Then, based on the size of these functions, we calculate the amount of time it takes to write unit tests for these functions (point 2 in Figure 3(b)). From that point on, we calculate the number of bugs for all of the functions and rank them in descending order. For the sake of this example, let us assume we are considering the top 3 functions. Assuming our heuristic identifies functions A, B and C as the functions for which we need to write unit tests, however, these functions may not be the ones with the most bugs. Assuming that the functions with the most bugs are functions A, D and E (i.e., they are the top 3 on the optimal list). From Figure 3(a) , we can see that functions A, B and C had 8 bug fixes in total after the unit tests were written for them. At the same time, Figure 3(b) shows that the optimal functions (i.e., functions A, D and E) had 13 bug fixes in them. Therefore, the best we could have done is to remove 13 bugs. We were able to remove 8 bugs using our heuristic, hence our POP is $\frac{8}{13} = 0.62$ or 62%.

It is important to note that the key difference between the usefulness and the POP values is that usefulness is the percentage of *functions* that we found useful to write unit tests for. On the other hand, POP measures the percentage of *bugs* that we could have avoided using a specific heuristic.

5. CASE STUDY RESULTS

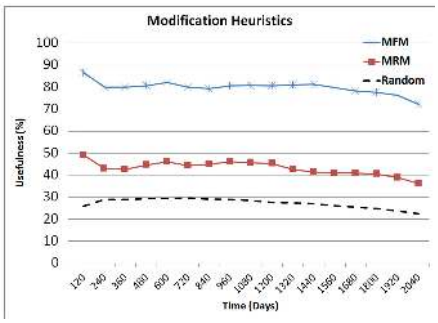
In this section, we present the results of the usefulness and POP metrics for the proposed heuristics. Ideally, we would like to have high usefulness and POP values. To evaluate the performance of each of the heuristics, we use the random heuristic as our baseline [15, 16, 32]. If we cannot do better than just randomly picking functions to add to the list, then the heuristic is not that effective. Since the random heuristic can give a different ordering each time, we use the average of 5 runs, each of which uses different randomly generated seeds.

5.1. Usefulness

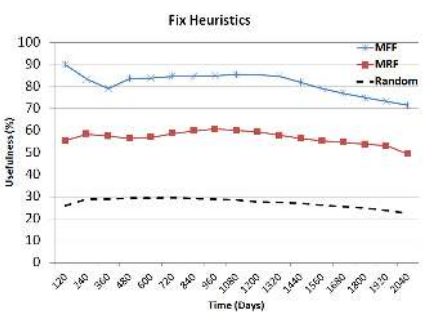
We calculate the usefulness for the heuristics listed in Table I and plot it over time for the commercial system and the Eclipse OSS system in Figures 4 and 5, respectively. The dashed black line in each of the figures depicts the results of the random heuristic. From the figures, we can observe that in the majority of the cases, the proposed heuristics outperform the random heuristic. However, our top performing heuristics (e.g., LF and LM) substantially outperform the random heuristic, in both projects.

The median usefulness values for each of the heuristics in the commercial system and the Eclipse OSS system are listed in Tables II and III, respectively. Since the usefulness values change over the course of the simulation, we chose to present the median values to avoid any sharp fluctuations. The last row of the table shows the usefulness achieved by the random heuristic. The heuristics are ranked from 1 to 9, with 1 indicating the best performing heuristic and 9 the worst.

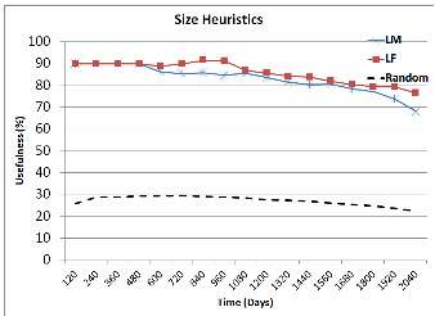
DOI: 10.1002/spe



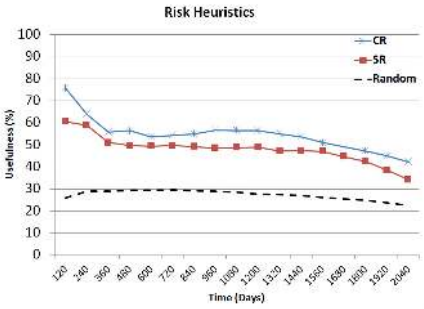
(a) Usefulness of modification heuristics



(b) Usefulness of fix heuristics

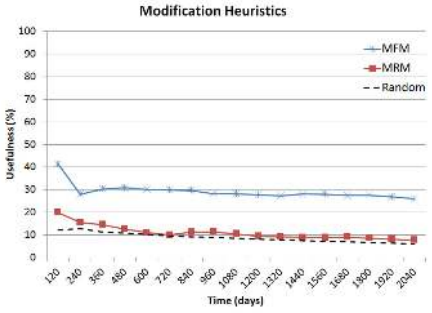


(c) Usefulness of size heuristics

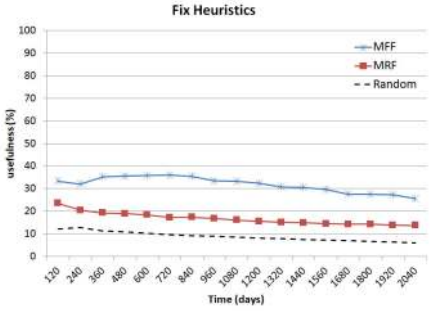


(d) Usefulness of risk heuristics

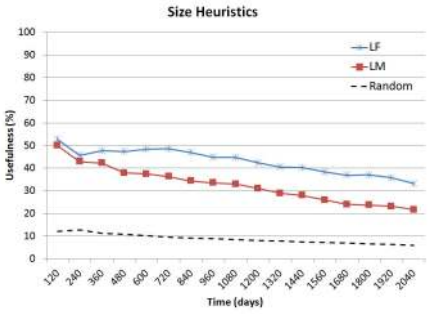
Figure 4. Usefulness of heuristics compared to the random heuristic for the commercial system



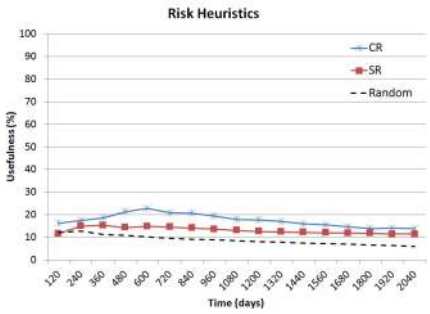
(a) Usefulness of modification heuristics



(b) Usefulness of fix heuristics



(c) Usefulness of size heuristics



(d) Usefulness of risk heuristics

Figure 5. Usefulness of heuristics compared to the random heuristic for the Eclipse OSS system

DOI: 10.1002/spe

Table II. Usefulness Results of the commercial system

Heuristic	Median Usefulness	Improvement over random	Rank
LF	87.0%	3.1 X	1
LM	84.7%	3.1 X	2
MFF	83.8%	3.0 X	3
MFM	80.0%	2.9 X	4
MRF	56.9%	2.1 X	5
CR	55.0%	2.0 X	6
SR	48.8%	1.8 X	7
MRM	43.1%	1.6 X	8
Random	27.7%	-	9

Table III. Usefulness Results of the Eclipse OSS system

Heuristic	Median Usefulness	Improvement over random	Rank
LF	44.7%	5.3 X	1
LM	32.9%	3.9 X	2
MFF	32.3%	3.8 X	3
MFM	28.1%	3.3 X	4
CR	17.4%	2.0 X	5
MRF	16.0%	1.9 X	6
SR	12.6%	1.5 X	7
MRM	9.9%	1.2 X	8
Random	8.5%	-	9

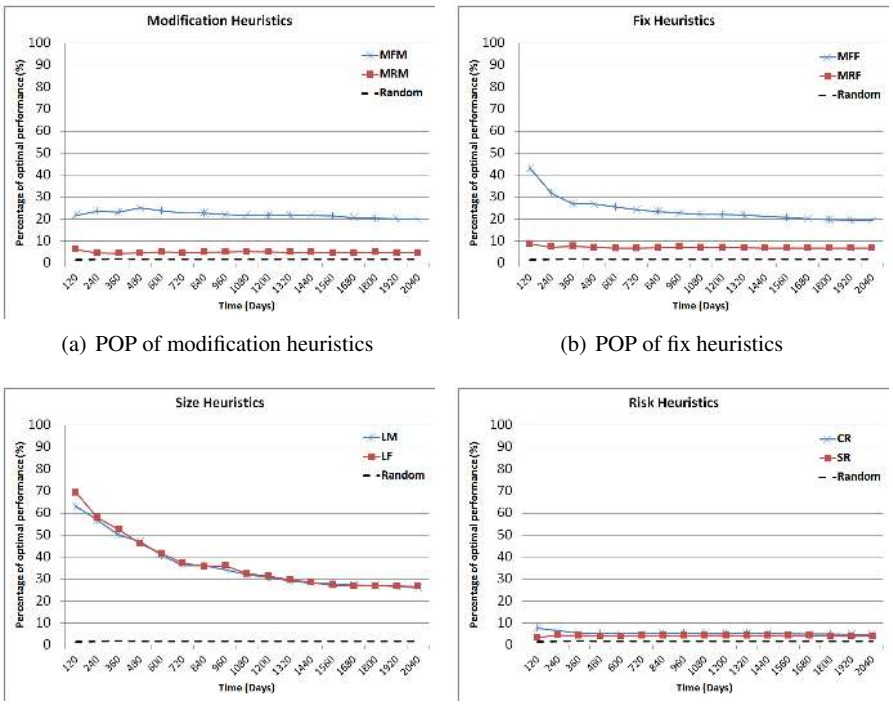
Commercial system: Table II shows that the LF, LM, MFF and MFM are the top performing heuristics, having median values in the range of 80% to 87%. The third column in Table II shows that these heuristics perform approximately 3 times better than the random heuristic.

A strategy that developers may be inclined to apply is to write tests for functions that they worked most recently on. The performance of such a strategy is represented by the recency heuristics (i.e., MRM and MRF). We can observe from Figures 4(a) and 4(b) that the recency heuristics (i.e., MRM and MRF) perform poorly compared to their frequency counterparts (i.e., MFM and MFF) and the size heuristics.

Eclipse OSS system: Similar to the commercial system, Table III shows that the LF, LM, MFF and MFM heuristics are the top performing heuristics. These heuristics achieve median values in the range of 28% to 44%. Although these usefulness values are lower than those achieved in the commercial system, they perform 3 to 5 times better than the random heuristic. The 3 to 5 times improvement is consistent with the results achieved in the commercial system.

Again, we can see that the frequency based heuristics (i.e., MFM and MFF) substantially outperform their recency counterparts (i.e., MRM and MRF). Examining the median values in Tables II and III, we can see that for MFM we were able to achieve approximately 30% median usefulness for the Eclipse OSS system (80% for the commercial system). This means that approximately 3 (8 for the commercial system) out of the 10 functions we wrote unit tests for had one or more bugs in the future. Therefore, writing the unit tests for these functions was useful. On the contrary, for the random heuristic, approximately 1 (3 for the commercial system) out of every 10 functions we wrote unit tests for had 1 or more bugs after the unit tests were written.

DOI: 10.1002/spe



(a) POP of modification heuristics

(b) POP of fix heuristics

(c) POP of size heuristics

(d) POP of risk heuristics

Figure 6. POP of heuristics compared to the random heuristic for the commercial system

Size, modification frequency and fix frequency heuristics should be used to prioritize the writing of unit tests for software systems. These heuristics achieve median usefulness values between 80–87% for the commercial system and between 28–44% for the Eclipse OSS system.

5.2. Percentage of Optimal Performance (POP)

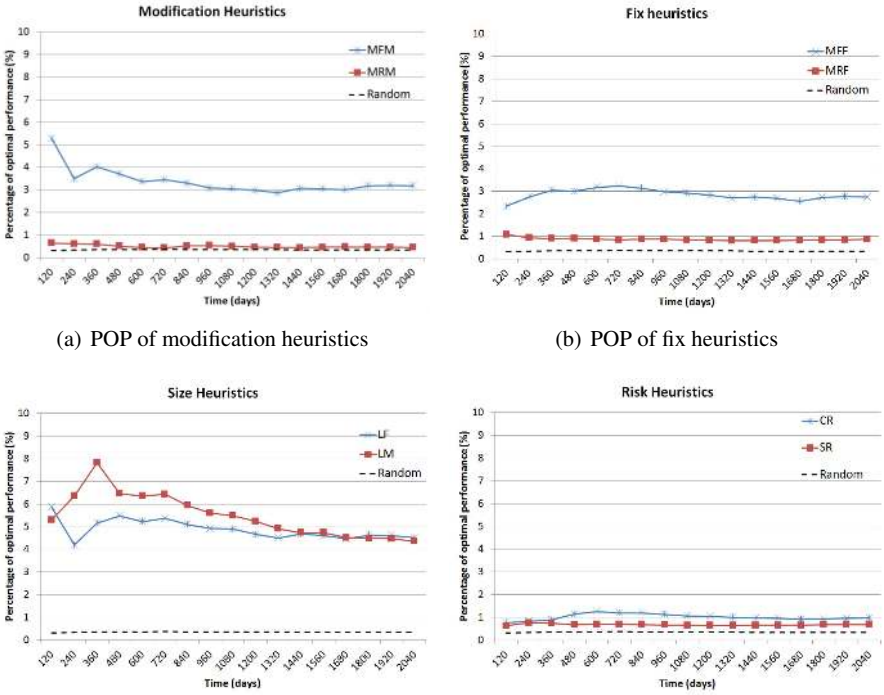
In addition to calculating the usefulness of the proposed heuristics, we would like to know how close we are to the optimal list of functions that we should write unit tests for if we have perfect knowledge of the future. We present the POP values for each of the heuristics in Figures 6 and 7. The performance of the random heuristic is depicted using the dashed black line. The figures show that in all cases, and for both the commercial and the Eclipse OSS system, the proposed heuristics outperform the random heuristic.

Commercial system: The median POP values are shown in Table IV. The POP values for the heuristics are lower than the usefulness values. The reason is that usefulness gives the percentage of functions that have one or more bugs. However, POP measures the percentage of bugs the heuristics can potentially avoid in comparison to the best we can do, if we have perfect knowledge of the future.

Although the absolute POP percentages are lower compared to the usefulness measure, the ranking of the heuristics remains quite stable (except for the SR and MRM, which exchanged 7th and 8th spot). Once again, the best performing heuristics are LF, LM, MFF and MFM. The median values for these top performing heuristics are in the 20% to 32.4% range. These values are 12 to 19 times better than the 1.7% that can be achieved using the random heuristic.

Eclipse OSS system: The median POP values are shown in Table V. The LM, LF, MFM and MFF are also the top performing heuristics for the Eclipse OSS system. Their median values are

DOI: 10.1002/spe



(a) POP of modification heuristics

(b) POP of fix heuristics

(c) POP of size heuristics

(d) POP of risk heuristics

Figure 7. POP of heuristics compared to the random heuristic for the Eclipse OSS system

Table IV. Percentage of Optimal Performance Results of the commercial system

Heuristic	Median POP	Improvement over random	Rank
LF	32.4%	19.1 X	1
LM	32.2%	18.9 X	2
MFF	22.2%	13.1 X	3
MFM	21.8%	12.8 X	4
MRF	7.0%	4.1 X	5
CR	5.5%	3.2 X	6
MRM	4.9%	2.9 X	7
SR	4.3%	2.5 X	8
Random	1.7%	-	9

between 2.76% to 5.31%. Although these values are low, they are much higher than the value of the random heuristic.

The median Eclipse POP values are considerably lower than the median POP values of the commercial system. A preliminary examination into this issue shows that the distribution of changes and bugs in the Eclipse OSS system is quite sparse. For example, in a certain year a set of files changed and then those files do not change for a while. This sort of behavior affects our simulation results, since we use the history of the functions to prioritize. One possible solution is to restrict how far into the future the simulator looks at when calculating the POP values, however, since our goal is to compare the heuristics, our current simulation suffices. In the future, we plan to investigate different strategies to improve the performance of the proposed heuristics.

DOI: 10.1002/spe

Table V. Percentage of Optimal Performance Results in the Eclipse OSS system

Heuristic	Median POP	Improvement over random	Rank
LM	5.31%	15.2 X	1
LF	4.68%	13.4 X	2
MFM	3.18%	9.1 X	3
MFF	2.76%	7.9 X	4
CR	0.97%	2.8 X	5
MRF	0.85%	2.4 X	6
SR	0.66%	1.9 X	7
MRM	0.46%	1.3 X	8
Random	0.35%	-	9

Regardless, our top performing heuristics performed approximately 8 to 15 times better than the random heuristic. The 8 to 15 times improvement is consistent with the finding in the commercial system.

Finally, we can observe a decline in the usefulness and POP values at the beginning of the simulation, shown in Figures 4, 5 , 6 and 7. This decline can be attributed to the fact that initially, there are many buggy functions for the heuristics to choose from. Then, after these buggy functions have been recommended, we remove them from the pool of functions that we can recommend. Therefore, the heuristics begin to recommend some functions that are not or less buggy. Previous studies by Ostrand *et al.* [29] showed that the majority of the bugs (approximately 80%) are contained in a small percentage of the code files (approximately 20%). These studies support our findings.

Size, modification frequency and fix frequency heuristics should be used to prioritize the writing of unit tests for software systems. These heuristics achieve median POP values between 21.8–32.4% for the commercial system and 2.76–5.31% for the Eclipse OSS system.

6. COMBINING HEURISTICS

Thus far, we have investigated the effectiveness of prioritizing the unit test creation using each heuristic individually. Previous work by Kim *et al.* showed that combining heuristics yields favorable results [45]. Therefore, we investigate whether or not combining the heuristics can further enhance the performance.

To achieve this goal, we use two combining strategies which we call COMBO and WEIGHTED COMBO. With COMBO, we generate a list of functions for each heuristic. Then, we take the top ranked function from each heuristic and write a test for them. The WEIGHTED COMBO heuristic on the other hand gives higher weight to functions put forward by higher ranked heuristics.

To illustrate, consider the example in Table VI. In this example, we have 3 heuristics: HA, HB and HC ranked 1, 2 and 3, respectively. Each of the heuristics lists 3 functions: f1, f2 and f3 ranked 1, 2 and 3, respectively.

In this case, the COMBO heuristics would recommend f1A from HA, f1B from HB and f1C from HC. The WEIGHTED COMBO heuristic uses the weights assigned to each function to calculate the list of recommended functions. The weight for each function is based on the rank of the heuristic relative to other heuristics and the rank of that function relative to other functions within its heuristic. Mathematically, the weight is defined as:

$$Function\ Weight = \frac{1}{Heuristic\ rank} * \frac{1}{Function\ rank} \quad (1)$$

DOI: 10.1002/spe

Table VI. Combining heuristics example

	Heuristic A (HA) Rank 1	Heuristic B (HB) Rank 2	Heuristic C (HC) Rank 3
1.	f1A	f1B	f1C
2.	f2A	f2B	f2C
3.	f3A	f3B	f3C

Table VII. Median Usefulness of combined heuristics

	Commercial system				**Eclipse OSS system**			
	LF	**COMBO**	**WEIGHTED COMBO**	**MRM**	**LF**	**COMBO**	**WEIGHTED COMBO**	**MRM**
Usefulness (%)	87.0	67.5	64.5	43.1	44.7	32.8	34.1	9.9

Table VIII. Median POP of combined heuristics

	Commercial system				**Eclipse OSS system**			
	LF	**COMBO**	**WEIGHTED COMBO**	**SR**	**LF**	**COMBO**	**WEIGHTED COMBO**	**MRM**
POP (%)	32.4	18.9	14.7	4.3	5.31	5.03	3.75	0.46

For example, in Table VI HA has rank 1. Therefore, f1A would have weight 1 (i.e. $\frac{1}{1} * \frac{1}{1}$), f2A would have weight 0.5 (i.e., $\frac{1}{1} * \frac{1}{2}$) and f3A would have weight 0.33 (i.e., $\frac{1}{1} * \frac{1}{3}$). Heuristic HB on the other hand has rank 2. Therefore, the weight for f1B is 0.5 (i.e., $\frac{1}{2} * \frac{1}{1}$), for f2B is 0.25 (i.e., $\frac{1}{2} * \frac{1}{2}$) and for f3B is 0.17 (i.e. $\frac{1}{2} * \frac{1}{3}$). Following the same method, the weight for f1C is 0.33, for f2C is 0.17 and for f3C is 0.11. In this case, if we were to recommend the top 5 functions, then we would recommend f1A, f2A, f1B, f3A, and f1C. In the case of a tie in the function weights (e.g., f2A and f1B), we choose the function with the higher ranked heuristic first (i.e., f2A).

We obtain the heuristic rankings from Tables II , III , IV, V and run the simulation using the combined heuristics. The Usefulness and the POP metrics were used to evaluate the performance of the COMBO and WEIGHTED COMBO metrics. Tables VII and VIII present the Usefulness and POP results, respectively. The best performing heuristics (i.e., LF and LM) outperform the COMBO and WEIGHTED COMBO heuristics in all cases. However, the COMBO and WEIGHTED COMBO heuristics provide a significant improvement over the worst performing heuristic (i.e., MRM and SR). The COMBO heuristic outperforms the WEIGHTED COMBO heuristic in most cases (except for Usefulness of the Eclipse OSS system). Taking into consideration the amount of time and effort required to gather and combine all of the different heuristics, and the performance of the combined heuristics, we suggest using only the top performing heuristic.

Although, combining heuristics did not yield a considerable improvement in our case, previous work by Kim *et al.* [45] showed favorable improvements in performance when heuristics are combined. We conjecture that these differences in performance are attributed to a few differences between the two approaches: 1) we recommend 10 functions in each iteration, while Kim *et al.'s* [45] the cache is made of 10% of the functions in the system; 2) in our approach, once a function is recommended once, it is removed from the pool of functions to be recommended in the future. In Kim *et al.'s* [45] the same function can be added and removed from the cache multiple times; 3) we consider the effort required to write a test for a function (depending on its size), while in [45] the effort is not considered.

Prepared using speauth.cls
DOI: 10.1002/spe

7. DISCUSSION

During our simulation study, we needed to decide on two simulation parameters: list size and available resources. In this section, we discuss the effect of varying these simulation parameters on our results. It is important to study the effect of these simulation parameters on our results because it helps us better understand the results we obtain from the simulation. In addition, it provides some insight into ways that could lead to more effective approaches.

7.1. Effect of List Size

In our simulations, each of the heuristics would recommend a list of functions that should have unit tests. Throughout our study, we used a list size of 10 functions. However, this list size was an arbitrary choice. We could have set this list size to 5, 20, 40 or even 100 functions. The size of the list will affect the usefulness and POP values.

To analyze the effect of list size, we vary the list size and measure the corresponding POP values at a one particular point in time. We measure the median POP, from the 5 year simulation, for each list size and plot the results in Figure 8(a). The y-axis is the log of the median POP value and the x-axis is the list size. We observe a common trend - an increase in the list size increases the POP for all heuristics. Once again, our top performing heuristics are unchanged with LF, LM, MFF and MFM scoring in the top for all list sizes. We performed the same analysis for usefulness and obtain similar results.

This trend can be explained by the fact that a bigger list size will make sure that more functions have unit tests written for them earlier on in the project. Since these functions are tested earlier on, we are able to avoid more bugs and the POP increases.

The results for the Eclipse OSS system are consistent with the findings in Figure 8(a).

7.2. Effect of Available Resources

A second important simulation parameter that we needed to set in the simulations is the effort available to write unit tests. This parameter determines how fast a unit test can be written. For example, if a function is 100 lines of code, and a tester can write unit tests for 50 lines of code per day, then she will be able to write unit tests for that function in 2 days.

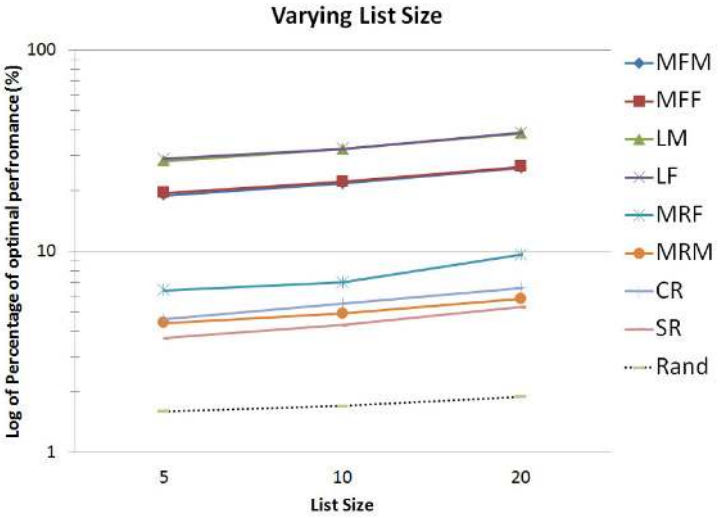
Throughout our study, we set this value to 100 lines per day. If this value is increased, then testers can write unit tests faster (due to an increase in man power or due to more efficient testers) and write tests for more functions. On the other hand, if we decrease this value, then it will take longer to write unit tests.

We varied this value from 50 to 200 lines per day (assuming the same effort is available each day) and measured the median POP, from the entire 5 year simulation. The results are plotted in Figure 8(b). We observe three different cases:

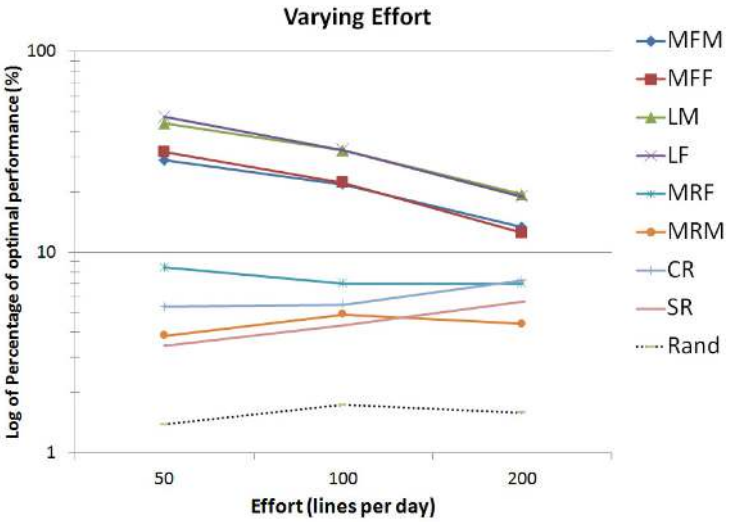
1. POP decreases as we increase the effort for heuristics LF, LM, MFF and MFM.
2. POP increases as we increase the effort for heuristics CR and SR.
3. POP either increases, then decreases or decreases, then increases as we increase the effort for heuristics MRF, MRM and Random.

We examined the results in more depth to try and explain the observed behavior. We found that in case 1, the POP decreases as we increase the effort because as we write tests for more functions (i.e., increasing effort available to 200 lines per day), we were soon writing tests for functions that did not have bugs after the tests were written. Or in other words, as we decrease the effort, less functions have unit tests written for them, which reduces the chance of prioritizing functions that do not have as many (or any) bugs in the future. In case 2, we found that the risk heuristics by themselves mostly identified functions that had a small number of bugs. Since an increase in effort means more functions can have unit tests written for them, therefore, we see an increase in the POP as effort is increased. In case 3, the MRF and MRM heuristics identify functions that are most recently modified or fixed. Any change in the effort will change the time it takes to write unit tests. This change in time will change the list of functions that should have unit tests written for

DOI: 10.1002/spe



(a) Effect of varying list size on POP



(b) Effect of varying effort on POP

Figure 8. Effect of simulation parameters for the commercial system

them. Therefore, an increase or decrease in the effort randomly affects the POP. As for the random heuristic, by definition, it picks random functions to write unit tests for.

The results for the Eclipse OSS system are consistent with the findings in Figure 8(b).

8. THREATS TO VALIDITY

This section discusses the threats to validity of our study.

Construct validity

We used the POP and Usefulness measures to compare the performance of the different heuristics. Although POP and Usefulness provide effective means for use to evaluate the proposed heuristics, they may not capture all of the costs associated with creating the unit tests, maintaining the test

Prepared using speauth.cls
DOI: 10.1002/spe

suites and managing the cost of different kinds of bugs (i.e., minor vs. major bugs).

Internal validity

In our simulations, we used 6 months to calculate the initial set of heuristics. Changing the length of this ramp-up period may effect the results from some heuristics. In the future, we plan to study the effect of varying this initial period in more detail.

Our approach assumes that each function has enough history such that the different heuristics can be calculated. Although our approach is designed for legacy systems, in certain cases new functions may be added, in which case little or no history can be found. In such cases, we ask practitioners to carefully examine and monitor such functions manually until enough history is accumulated to use our approach.

Throughout our simulation study, we assume that all bug fixes are treated equally. However, some bugs have a higher severity and priority than others. In the future, we plan to consider the bug severity and priority in our simulation study.

External validity

Performing our case studies on a large commercial software system and the Eclipse OSS system with a rich history significantly improves the external validity of our study. However, our findings may not generalize to all commercial or open source software systems since each project may have its own specific circumstances.

Our approach uses the comments in the source control system to determine whether or not a change is a bug fix. In some cases, these comments are not properly filled out or do not exist. In this case, we assume that the change is a general maintenance change. We would like to note that at least in the case of the commercial system, the comments were very well maintained.

When calculating the amount of time it takes to write a the unit test for a function in our simulations, we make the assumption that all lines in the function will require the same effort. This may not be true for all functions.

Additionally, our simulation assumes that if a function is recommended once, it needs not be recommended again. This assumption is fueled by the fact that TDM practices are being used and after the initial unit test, all future development will be accompanied by unit tests that test the new functionality.

We assume that tests can be written for individual functions. In some cases, functions are closely coupled with other functions. This may make it impossible to write unit tests for the individual functions, since unit tests for these closely coupled functions need to be written simultaneously.

In our study of the effect of list size and available resources (Section 7), we set the time to a fixed point and varied the parameters to study the effect on POP. Using a different time point may lead to steeper/flatter curves. However, we believe that the trends will still be the same.

9. RELATED WORK

The related work can be categorized into two main categories: test case prioritization and fault prediction using historical data.

Test Case Prioritization

The majority of the existing work on test case prioritization has looked at prioritizing the execution of tests during regression testing to improve the fault detection rate [15, 32, 36–38].

Rothermel *et al.* [38] propose several techniques that use previous test execution information to prioritize test cases for regression testing. Their techniques ordered tests based on the total coverage of code components, the coverage of code components not previously covered and the estimated ability to reveal faults in the code components that Rothermel *et al.* cover. They showed that all of their techniques were able to outperform untreated and randomly ordered tests. Similarly, Aggrawal *et al.* [37] proposed a model that optimizes regression testing while achieving 100% code coverage.

DOI: 10.1002/spe

Elbaum *et al.* [15] showed that test case prioritization techniques can improve the rate of fault detection of test suites in regression testing. They also compared statement level and function level techniques and showed that at both levels, the results were similar. In [32], the same authors improved their test case prioritization by incorporating test costs and fault severities and validated their findings using several empirical studies [32, 39, 40].

Kim *et al.* [41] used historical information from previous test suite runs to prioritize tests. Walcott *et al.* [42] used genetic algorithms to prioritize test suites based on the testing time budget.

Our work differs from the aforementioned work in that we do not assume that tests are already written, rather, we are trying to deal with the issue of which functions we should write tests for. We are concerned with the prioritization of unit test writing, rather than the prioritization of unit test execution. Due to the fact that we do not have already written tests, we have to use different heuristics to prioritize which functions of the legacy systems we should write unit tests for. For example, some of the previous studies (e.g., [41]) use historical information based on previous test runs. However, we do not have such information, since the functions we are trying to prioritize have never had tests written for them in the first place.

Fault Prediction using Historical Data

Another avenue of closely related work is the work done on fault prediction. Nagappan *et al.* [24, 31] showed that dependency and relative churn measures are good predictors of defect density and post-release failures. Holschuh *et al.* [43] used complexity, dependency, code smell and change metrics to build regression models that predict faults. They showed that these models are accurate 50-60% of the time, when predicting the 20% most defect-prone components. Additionally, studies by Arisholm *et al.* [23], Graves *et al.* [25], Khoshgoftaar *et al.* [26] and Leszak *et al.* [27] have shown that prior modifications are a good indicator of future bugs. Yu *et al.* [28], and Ostrand *et al.* [29] showed that prior bugs are a good indicator of future bugs. In their follow-up work, Ostrand *et al.* [47] showed that 20% of the files with the highest number of predicted faults contain between 71-92% of the faults, for different systems that may follow different development processes [48]. Hassan [44] showed that the complexity of changes is a good indicator of potential bugs.

Mende and Koschke [34] examined the use of various performance measures of bug prediction models. They concluded that performance measures should always take into account the size of source code predicted as defective, since the cost of unit testing and code reviews is proportional to the size of a module.

Other work used the idea of having a cache that recommends buggy code. Hassan and Holt [22] used change and fault metrics to generate a Top Ten list of subsystems (i.e., folders) that managers need to focus their testing resources on. Kim *et al.* [45] extended Hassan and Holt's work [22] and use the idea of a cache that keeps track of locations that were recently added, recently changed and where faults were fixed to predict where future faults may occur (i.e., faults within the vicinity of a current fault occurrence). They performed their prediction at two levels of granularity: file- and method/function-level.

There are some key differences between our work and the work on fault prediction:

1. Our work prioritizes functions at a finer granularity than most previous work on fault prediction (except for *Kim et al.'s* approach [45] which predicts at the file and function/method level). Instead of identifying buggy files or subsystems, we identify buggy functions. This difference is critical since we are looking to write unit tests for the recommended functions. Writing unit tests for entire subsystems or files may be wasteful, since one may not need to test all of the functions in the file or subsystem.
2. Our work considers the effort required to write the unit tests for the function/method. Furthermore, since our approach is concerned with the unit test creation, we removed functions/methods after they are recommended once.
3. Fault prediction techniques provide a list of potentially faulty components (e.g., faulty directories or files). Then it is left up to the manager to decide how to test this directory or file. Our work puts forward a concrete approach to assist in the prioritization of unit test writing, given the available resources and knowledge about the history of the functions.

DOI: 10.1002/spe

10. CONCLUSIONS AND FUTURE WORK

In this paper we present an approach to assist practitioners applying TDM, prioritize the writing of unit tests for legacy systems. Different heuristics are used to generate lists of functions that should have unit tests written for them. To evaluate the performance of each of the heuristics, we perform a simulation-based case study on a large commercial legacy software system and the Eclipse OSS system. We compared the performance of each of the heuristics to that of a random heuristic, which we use as a base line comparison. All of the heuristics outperformed the random heuristic in terms of usefulness and POP. Our results showed that, in both systems, heuristics based on the function size (i.e., LF and LM), modification frequency (i.e., MFM) and bug fixing frequency (i.e., MFF) perform the best for the purpose prioritization of unit test writing efforts. Furthermore, we studied whether we can enhance the performance by combining the heuristics. The results showed that combining the heuristics does not improve the performance when compared to the best performing heuristic (i.e., LF). Finally, we examine the effect of varying list size and the resources available to write unit tests on the simulation performance.

Although the approach presented in this paper assumed legacy systems that did not have any unit tests written for them in the past, we would like to note that this is not the only use case for this approach. In the future we plan to adapt our approach to work for legacy systems that may have some unit tests written for them already (since this might be commonly encountered in practice). In addition, we plan to extend the approach to leverage any other historical data (e.g., systems tests or domain knowledge) when recommending functions to write unit tests for. Domain knowledge information and existing system tests can help guide us toward specific parts of the legacy that might be more (or less) important to test. In that case, we can use this information to know where we should start our prioritization.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their input and feedback. We also thank Jim Whitehead for his feedback and assistance in clarifying the cache history work [45].

In addition, we are grateful to Research In Motion for providing data that made this study possible. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of Research In Motion. Moreover, our results do not in any way reflect the quality of Research In Motion's software products.

REFERENCES

1. L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.
2. K. Beck and M. Fowler, *Planning extreme programming*. Addison-Wiley, 2001.
3. P. Runeson, "A survey of unit testing practices," *IEEE Softw.*, vol. 23, no. 4, 2006.
4. N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Softw. Engg.*, vol. 13, no. 3, 2008.
5. B. George and L. Williams, "An initial investigation of test driven development in industry," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, 2003.
6. ——, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, 2003.
7. H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transaction on Software Engineering*, vol. 31, no. 3, 2005.
8. M. M. Müller and W. F. Tichy, "Case study: extreme programming in a university environment," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.
9. L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, 2000.
10. J. Moad, "Maintaining the competitive edge," *Datamation*, vol. 64, no. 66, 1990.
11. H. Muller, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology," 1994.
12. K. Bennett, "Legacy systems: Coping with success," *IEEE Softw.*, vol. 12, no. 1, 1995.
13. J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: Issues and directions," *IEEE Software*, vol. 16, no. 5, 1999.
14. E. R. Harold, "Testing legacy code," http://www.ibm.com/developerworks/java/library/j-legacytest.html.

DOI: 10.1002/spe

15. 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*, 2000.
16. H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.
17. M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, 1975.
18. W. F. Tichy, "Rcs - a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, 1985.
19. "Bugzilla," http://www.bugzilla.org/.
20. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.
21. A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM '00: Proceedings of the International Conference on Software Maintenance*, 2000.
22. A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.
23. E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006.
24. N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, 2007.
25. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions of Software Engineering*, vol. 26, no. 7, July 2000.
26. T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 5, 1999.
27. M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *J. Syst. Softw.*, vol. 61, no. 3, 2002.
28. T.-J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, 1988.
29. T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004.
30. I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.
31. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.
32. S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.
33. E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, 2007.
34. T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 2009.
35. R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.
36. W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore, "A study of effective regression testing in practice," in *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.
37. K. K. Aggrawal, Y. Singh, and A. Kaur, "Code coverage based technique for prioritizing test cases for regression testing," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, 2004.
38. G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, 2001.
39. S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, 2002.
40. S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Control*, vol. 12, no. 3, 2004.
41. 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*, 2002.
42. K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *ISSTA '06: Proceedings of the international symposium on Software testing and analysis*, 2006.
43. T. Holschuh, M. Puser, K. Herzig, T. Zimmermann, P. Rahul, and A. Zeller, "Predicting defects in sap java code: An experience report." in *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*, 2009.
44. A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009.
45. S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007.
46. E. Shihab, Z. Jiang, B. Adams, A. E. Hassan, R. Bowerman, "Prioritizing Unit Test Creation for Test-Driven Maintenance of Legacy Systems," in *QSIC '10: Proceedings of the 10th International Conference on Quality Software*, 2010.
47. T.J. Ostrand, E. J. Weyuker, and R.M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, 2005.
48. R.M. Bell, T.J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in *ISSTA '06: Proceedings of the International Symposium on Software Testing and Analysis*, 2006.
49. T. Zimmermann, R. Premraj and A. Zeller, "Predicting Defects for Eclipse," in *PROMISE'07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

DOI: 10.1002/spe

50. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Datasets," in *FSE'09: Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.
51. T. H. D. Nguyen, B. Adams, A. E. Hassan, "A Case Study of Bias in Bug-Fix Datasets," in *WCRE'10: Proceedings of the 17th Working Conference on Reverse Engineering*, 2010.

DOI: 10.1002/spe