Advances in Software Engineering

Software Test Automation

Guest Editors: Philip Laplante, Fevzi Belli, Jerry Gao, Greg Kapfhammer, Keith Miller, W. Eric Wong, and Dianxiang Xu

Software Test Automation

Advances in Software Engineering

Software Test Automation

Guest Editors: Philip Laplante, Fevzi Belli, Jerry Gao, Greg Kapfhammer, Keith Miller, W. Eric Wong, and Dianxiang Xu

Editorial Board

Contents

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 163746, 2 pages
doi:10.1155/2010/163746

Editorial

Software Test Automation

Phillip Laplante,[1] Fevzi Belli,[2] Jerry Gao,[3] Greg Kapfhammer,[4] Keith Miller,[5] W. Eric Wong,[6] and Dianxiang Xu[7]

[1] *Engineering Division, Great Valley School of Graduate Professional Studies, Penn State, 30 East Swedesford Road, Malvern, PA 19355, USA*
[2] *Department of Electrical Engineering and Information Technology, University of Paderborn, 33095 Paderborn, Germany*
[3] *Computer Engineering Department, San Jose State University, One Washington Square, San Jose, CA 95192-0180, USA*
[4] *Department of Computer Science, Allegheny College, Meadville, Pennsylvania, USA*
[5] *Department of Computer Science, University of Illinois at Springfield, One University Plaza, UHB 3100, Springfield, IL 62703, USA*
[6] *Department of Computer Science, The University of Texas at Dallas, 800 West Campbell, Richardson, TX 75080, USA*
[7] *National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD 57042, USA*

Correspondence should be addressed to Phillip Laplante, plaplante@psu.edu

Received 31 December 2009; Accepted 31 December 2009

Software testing is an essential, yet time-consuming, and expensive activity. Therefore, automation of any aspect of software test engineering can reduce testing time and, in the long-run, reduce costs for the testing activity. While there are many research directions in testing automation, from theory through application, the main focus of this special issue is on partially or fully validated tools, techniques, and experiences.

In response to the call for papers a broad range of submissions were received. Consistent with the standards of a highly regarded journal, all submissions received several rounds of review and revision and only outstanding papers were selected, yielding an acceptance rate of 60%. The resultant collection provides a number of important and useful results.

For example, in "A tester-assisted methodology for test redundancy detection" Koochakzadeh and Garousi propose a semiautomated methodology based on coverage metrics to reduce a given test suite while keeping the fault detection effectiveness unchanged. They then validate their approach on Allelogram, an open source Java program used by biological scientists for processing genomes. The results of the experiments confirm that the semiautomated process leads to a reduced test suite with the same fault detection ability as the original test suite.

In "A strategy for automatic quality signing and verification processes for hardware and software testing," Younis and Zamli give a technique for optimizing the test suite required for testing both hardware and software in a production line. Their strategy is based a "Quality Signing Process" and a "Quality Verification Process." The Quality Signing Process involves parameter interaction while the Quality Verification Process is based on mutation testing and fault injection. The novelty of the proposed strategy is that the optimization and reduction of test suite is performed by selecting only mutant killing test cases from cumulating t-way test cases. The results demonstrate that the proposed strategy outperforms traditional block partitioning with the same number of test cases.

There are considerable costs involved in regression testing, which often cause practitioners to short-circuit the activity. In "Automated test case prioritization with reactive GRASP" Mai et al. propose the use of the Reactive GRASP (Greedy Randomized Adaptive Search Procedures) metaheuristic for the regression test case prioritization problem. They compare this metaheuristics with five other search-based algorithms previously described in the literature. Five programs were used in the experiments and the results demonstrate good coverage performance with respect to many of the compared techniques and a high stability of the results generated by the proposed approach. Their work also confirms some of the previous results reported in the literature concerning the other prioritization algorithms.

Automated GUI test case generation is a highly resource intensive process. In “A proposal for automatic testing of GUIs based on annotated use cases,” Mateo Navarro et al. describe a new approach that reduces the effort by automatically generating GUI test cases. The test case generation process is guided by use cases describing behavior recorded as a set of interactions with the GUI elements. These use cases are then annotated by the tester to indicate interesting variations in widget values and validation rules with expected results. Once the use cases are annotated, this approach uses the new values and validation rules to automatically generate test cases and validation points, expanding the test coverage.

The next paper by Darwin, “AnnaBot: a static verifier for Java annotation usage,” describes one of the first tools permitting verification of correct use of annotation-based metadata in Java. This verification becomes especially important both as annotations become more widely adopted and as multiple implementations of certain standards become available. The author describes the domain-specific language and parser for AnnaBot, which are available for free from the author’s website

Finally a study of the software test automation practices in industry was conducted and also the results given in “Software test automation in practice: empirical observations” by Kasurinen et al.. Both qualitative interviews and quantitative surveys of 55 industry specialists from 31 organizational units across 12 software development organizations were conducted. The study revealed that only 26% of the organizations used automated testing tools, primarily in quality control and quality assurance. The results also indicated that adopting test automation in software organization is a demanding effort

We hope you enjoy this eclectic set of works relating to software testing automation. The editors also wish to thank the authors for their excellent contributions and the reviewers for their outstanding efforts in helping to select and improve all of the papers in this special issue.

Phillip Laplante
Fevzi Belli
Jerry Gao
Greg Kapfhammer
Keith Miller
Eric Wong
Dianxiang Xu

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 932686, 13 pages
doi:10.1155/2010/932686

Research Article

A Tester-Assisted Methodology for Test Redundancy Detection

Negar Koochakzadeh and Vahid Garousi

Software Quality Engineering Research Group (SoftQual), Department of Electrical and Computer Engineering, Schulich School of Engineering, University of Calgary, Calgary, AB, Canada T2N 1N4

Correspondence should be addressed to Negar Koochakzadeh, nkoochak@ucalgary.ca

Received 15 June 2009; Revised 16 September 2009; Accepted 13 October 2009

Academic Editor: Phillip Laplante

Test redundancy detection reduces test maintenance costs and also ensures the integrity of test suites. One of the most widely used approaches for this purpose is based on coverage information. In a recent work, we have shown that although this information can be useful in detecting redundant tests, it may suffer from large number of false-positive errors, that is, a test case being identified as redundant while it is really not. In this paper, we propose a semiautomated methodology to derive a reduced test suite from a given test suite, while keeping the fault detection effectiveness unchanged. To evaluate the methodology, we apply the mutation analysis technique to measure the fault detection effectiveness of the reduced test suite of a real Java project. The results confirm that the proposed manual interactive inspection process leads to a reduced test suite with the same fault detection ability as the original test suite.

1. Introduction

In today's large-scale software systems, test (suite) maintenance is an inseparable part of software maintenance. As a software system evolves, its test suites need to be updated (maintained) to verify new or modified functionality of the software. That may cause test code to erode [1, 2]; it may become complex and unmanageable [3] and increase the cost of test maintenance. Decayed parts of test suite that cause test maintenance problems are referred to as test *smells* [4].

Redundancy (among test cases) is a discussed but a seldom-studied test smell. A redundant test case is one, which if removed, will not affect the fault detection effectiveness of the test suite. Another type of test redundancy discussed in the literature (e.g., [5, 6]) is test code duplication. This type of redundancy is similar to conventional source code duplication and is of syntactic nature. We refer to the above two types of redundancy as semantic and syntactic test redundancy smells, respectively. In this work, we focus on the semantic redundancy smell which is known to be more challenging to detect in general than the syntactic one [5].

Redundant test cases can have serious consequences on test maintenance. By modifying a software unit in the maintenance phase, testers need to investigate the test suite to find all relevant test cases which test that feature and update them correctly with the unit. Finding all of the related test cases increases the cost of maintenance. From the other hand, if test maintenance (updating) is not conducted carefully, the integrity of the entire test suite will be under question. For example, we can end up in a situation in which two test cases test the same features of a unit, if one of them is updated correctly with the unit and not the other one, one test may fail while the other may pass, making the test results ambiguous and conflicting.

The motivation for test redundancy detection is straightforward. By detecting and dealing with redundant test case (e.g., carefully removing them), we reduce test maintenance cost and the risk of loosing integrity in our test suite, while fault detection capability of our test suite remains constant.

One of the most widely used approaches in the literature (e.g., [6–11]) for test redundancy detection, also referred to as test minimization, is based on coverage information. The rationale followed is that, if several test cases in a test suite execute the same program elements, the test suite can then be reduced to a smaller suite that guarantees equivalent test coverage ratio [6].

However, test redundancy detection based on coverage information does not guarantee to keep fault detection capability of a given test suite. Evaluation results from our previous work [12] showed that although coverage information can be very useful in test redundancy detection, detecting redundancy only based on this information may lead to a test suite which is weaker in detecting faults than the original one.

Considering fault detection capability of a test case for the purpose of redundancy detection is thus very important. To achieve this purpose, we propose a collaborative process between testers and a proposed redundancy detection engine to guide the tester to use valuable coverage information in a proper and useful way.

The output of the process is a reduced test suite. We claim that if testers play their role carefully in this process, fault detection effectiveness of this reduced test set would be equal to the original set.

High amount of human effort should be spent on inspecting a test suite manually. However, the proposed process in this paper tries to use the coverage information in a constructive fashion to reduce the required tester efforts. More automation can be added to this process later to save more cost and thus the proposed process should be considered as the first step to reduce required human effort for test redundancy detection.

To evaluate our methodology, we apply the mutation technique in a case study in which common types of faults are injected. Then original and reduced test set are then executed to detect faulty versions of the systems. The results show similar capability of fault detection for those two test sets.

The remainder of this paper is structured as follows. We review the related works in Section 2. Our recent previous work [12] which evaluated the precision of test redundancy detection based on coverage information is summarized in Section 3. The need for knowledge collaboration between human testers and the proposed redundancy detection engine is discussed in Section 4. To leverage and share knowledge between the automated engine and human tester, we propose a collaborative process for redundancy detection in Section 5. In Section 6, we show the results of our case study and evaluate the results using the mutation technique. Efficiency, precision, and a summary of the proposed process are discussed in Section 7. Finally, we conclude the paper in Section 8 and discuss the future works.

2. Related Works

We first review the related works on test minimization and test redundancy detection. We then provide a brief overview of the literature on semiautomated processes that collaborate with software engineers to complete tasks in software engineering and specifically in software testing.

There are numerous techniques that address test suite minimization by considering different types of test coverage criteria (e.g., [6–11]). In all of those works, to achieve the maximum possible test reduction, the smallest test set which covers the same part of the system was created [7]. The problem of finding the smallest test set has been shown to be NP-complete [13]. Therefore, in order to find an approximation to the minimum cardinality test set, heuristics are usually used in the literature (e.g., [7, 9]).

A few works have applied data flow coverage criteria (e.g., [7, 10]) while a few others have applied control flow criteria (e.g., [6, 9, 11]).

In [7], in addition to the experiment which was performed for all-definition-use coverage criterion on a relatively simple program (LOC is unknown), the authors mentioned that all the possible coverage criteria should be considered in order to detect redundant test cases more precisely. The authors were able to reduce 40% of the size of the test suite under study based on coverage information.

Coverage criteria used in [10] were predicate-use, computation-use, definition-use, and all-uses. The authors applied their approach on 10 Unix programs (with average LOC of 354) and 91% of the original test suites were reduced in total.

The control flow coverage criteria used in [6, 9, 11] are Branch [6], statement [9], and MC/DC [11]. In [9], mutation analysis was used to assess and evaluate the fault detection effectiveness of the reduced test suites. The ratios of reduction reported in these works were 50%, 34%, and 10%, respectively. The Systems Under Tests (SUTs) used in [6, 9] were small scale (avg. LOC of 29 and 231, resp.), while [11] used a medium size space program as its SUT with 9,564 LOC.

The need to evaluate test redundancy detection by assessing fault detection effectiveness was mentioned in [6, 11]. In those works, faults were manually injected into the SUTs to generate mutants. Then the mutation scores of original and reduced test sets were compared. Reference [6] concludes that test minimization based on coverage information can reduce the ability of fault detection, while [11] showed opposite conclusions.

In [6], faults were seeded to the SUTs manually by modifying mostly a single line of code (first order mutation), while in a few other cases, the authors modified between two and five lines of code (k-order mutation). As mentioned in [6], ten people (mostly without knowledge of each other's work) had tried to introduce faults that were as realistic as possible, based on their experience with real programs.

In [11], the manually injected faults (18 of them) were obtained from the error-log maintained during its testing and integration phase. Eight faults were in the "logic omitted or incorrect" category, seven faults belong to the type of "computational problems," and the remaining three faults had "data handling problems" [11].

In our previous work [12], an experiment was performed with 4 real Java programs to evaluate coverage-based test redundancy detection. The objects of study were JMeter, FitNesse, Lurgee and Allelogram with LOC of 69,424, 22,673, 7,050, and 3,296, respectively. Valuable lessons learned from our previous experiment revealed that coverage information cannot be the only source of knowledge to precisely detect test redundancy. Lessons are summarized in Section 3 of this paper.

To the best of the authors' knowledge, there has been no existing work to improve the shortcomings (imprecision) of coverage-based redundancy detection. In this paper, we are proposing a semiautomated process for this purpose.

Semiautomated decision supports systems leverage human-computer interaction which put together the knowledge of human users and intelligent systems to support decision-making tasks. Hybrid knowledge is very effective in such situations where the computational intelligence provides a set of qualified and diversified solutions and human experts are involved interactively in the decision-making process for final decision [14].

A logical theory of human-computer interaction has been suggested by Milner [15]. Besides, the ways in which open systems' behavior can be expressed by the composition of collaborative components is explained by Arbab [16]. There are various semiautomated systems designed for software engineering such as user-centered software design [17].

There have also been semiautomated systems used specifically in software testing. For instance, test case generation tools require tester's assistance in providing test oracles [18]. Another example of collaborative tool for testing is manual testing frameworks [19]. In these tools, testers perform test cases manually while system records them for later uses. The process proposed in this paper is a semiautomated framework with the purpose of finding test redundancy in software maintenance phase.

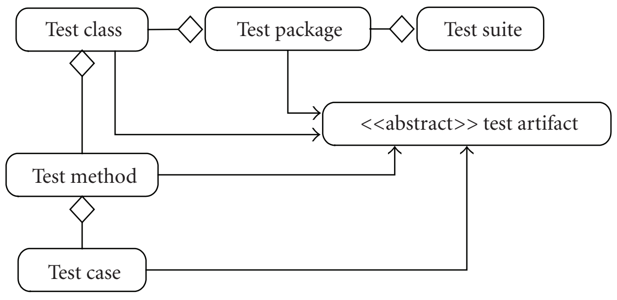


Figure 1: Test granularity in JUnit.

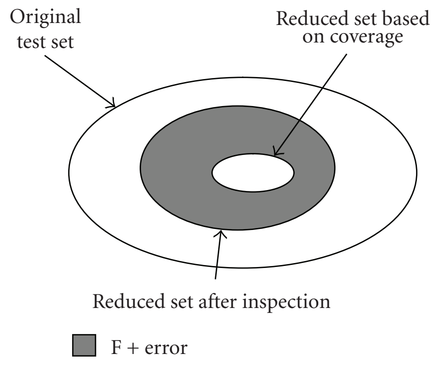


Figure 2: False-Positive Error in Test Redundancy Detection based on Coverage Information.

3. Coverage-Based Redundancy Detection Can Be Imprecise

In our previous work [12], we performed an experiment to evaluate test redundancy detection based only on coverage information. We formulated two experimental metrics for coverage-based measurement of test redundancy in the context of JUnit test suites. We then evaluated the approach by measuring the redundancy of four real Java projects (FitNesse, Lurgee, Allelogram, and JMeter). The automated test redundancy measures were compared with manual redundancy decisions derived from inspection performed by a human software tester.

In this paper, we use the term *test artifact* for different granularity levels supported in JUnit (Figure 1). Three levels of package, class, and methods are grouping mechanism for test cases that have been introduced in JUnit.

The results from that study [12] showed that measuring test redundancy based only on coverage information is vulnerable to imprecision given the current implementation of JUnit unit test framework and also coverage tools. The following discussion explains the root causes.

In the SUTs we analyzed in [12], about 50% of test artifacts, manually recognized as nonredundant, had been detected as redundant tests by our coverage-based redundancy metrics. In a Venn diagram notation, Figure 2 compares a hypothetical original test set with two reduced sets showing high number of false-positive errors. Three main reasons discovered in [12] to justify the errors are discussed next.

(1) Test redundancy detection based on coverage information in all previous works have been done by only considering limited number of coverage criteria. This fact that two test cases may cover the same part of SUT according to one coverage criterion but not the other one causes impreciseness in test redundancy detection only by considering one coverage criterion.

(2) In JUnit, each test case contains four phases: setup, exercise, verify, and teardown [4]. In the setup phase the required state of the SUT for the purpose of a particular test case is setup. In the exercise phase, the SUT is exercised. In the teardown phase the SUT state is rolled back into the state before running the test. In these three phases SUT is covered while in the verification phase only a comparison between expected and actual outputs is performed and SUT is not covered. Therefore, there might be some test cases with the same covered part of SUT with various verifications. In this case, coverage information may lead to detecting a nonredundant test as redundant.

(3) Coverage information is calculated only based on the SUT instrumented for coverage measurement. External resources (e.g., libraries) are not usually instrumented. There are cases in which two test methods cover different libraries. In such cases, the coverage information of the SUT alone is not enough to measure redundancies.

Another reason of impreciseness in redundancy detection based on coverage information mentioned in [12]

```
public void testAlleleOrderDoesntMatter () {
  Genotype g1 = new Genotype(new double [ ] {0,1});
  Genotype g2 = new Genotype(new double [ ] [13]);
  assertTrue (g1.getAdjustedAlleleValues (2).
          equals(g2.getAdjustedAlleleValues (2)));
}
public void testOffset (){
  Genotype g = new Genotype(new double [ ]{0,1});
  g.offsetBy (0.5);
  List<Double> adjusted =
  g.getAdjustedAlleleValues (2);
  assertEquals (2, adjusted.size ());
  assertEquals (0.5, adjusted.get (0));
  assertEquals (1.5, adjusted.get (1));
  g.clearOffset();
  adjusted = g.getAdjustedAlleleValues (2);
  assertEquals (0.0, adjusted.get (0));
  assertEquals (1.0, adjusted.get (1));
}
```

Algorithm 1: Source code of two test methods in the Allelogram test suite.

was some limitations in coverage tools implementation. For example, the coverage tool that we used in [12] was *CodeCover* [20]. The early version of this tool (version 1.0.0.0) was unable to instrument `return` and `throw` statements due to a technical limitation. Hence, the earlier version of the tool excluded covering of such statements from coverage information. This type of missing values can lead to detecting a nonredundant test as redundant. However, this limitation has now been resolved in the newest version of CodeCover (version 1.0.0.1 released on April 2009) and we have updated our redundancy detection framework by using the latest version of this tool. Since in [12] this problem was a root cause of false positive error, here we just report this as a possible reason of impreciseness in redundancy detection, while in this paper we do not have this issue.

Algorithm 1 shows the source code of two test methods from *Allelogram* test suite as an example of incorrect redundancy detection by only applying coverage information. In this example, test method *testAlleleOrderDoesntMatter* covers a subset of covered items by the test method *testOffset* both in setup and exercise phases. The setup phase includes calling *Genotype(new double)* constructor. The exercise phase contains calling *getAdjestedAlleleValues(int)* method by passing the created *Genotype* object, which both are called in the second test method as well. However, the assertion goal in the first test is completely different from the assertion goal in the second one. In the first test method, the goal is comparing the output value of *getAdjestedAlleleValues* method for two *Genotype* objects, while in second one, one of the goals is checking the size of output list from the *getAdjestedAlleleValues* method. Therefore, although according to coverage information the first test method is redundant, in reality it is nonredundant.

4. The Need for Knowledge Collaboration with Testers

Reduced test set based on coverage information contains those test artifacts that cover at least one coverable item not covered by any other test artifact. Therefore these test artifacts contribute to achieving more coverage and according to the concept of test coverage, they may increase the fault detection capability of the test suites.

Based on the above discussion, it is worthwhile to use coverage information for test redundancy detection to reduce the number of test artifacts that might be redundant.

On the other side, high ratio of false-positive errors shows that the coverage-based results alone are not reliable and we may inaccurately detect many nonredundant test artifacts as redundant ones.

The above advantages and disadvantages of coverage-based redundancy detection have motivated us to improve the test redundancy detection process by leveraging knowledge from human testers. The three main root causes of imprecision discussed in Section 3 should be considered in such a tester-assisted approach.

First, the more coverage criteria are applied, the more precise test redundancy will be detected. However, all of the existing test coverage tools support a limited number of coverage criteria. White-box criteria are more usually supported, while there are only a few tools supporting black-box criteria (e.g., JFeature [21]). In addition, usually there are no precise formal specifications for some units in some systems. Thus, automated measurement of black-box coverage is impossible in those cases. Also, there is a lack of coverage tools which automatically measure both white-box and black-box coverage criteria at the same time. Combing the coverage results from various coverage tools might be a solution. However, lack of formal specification for many real projects makes it very challenging for us testers to consider automated measurement of black-box coverage for the purpose of redundancy detection in this work []. For projects with full formal specifications, if test minimization is performed precisely with respect to all available coverage criteria, loss of fault detection ability can be minimized or eliminated altogether. However, since formal specifications are not available for many real projects, we propose to involve human testers in the process of test redundancy detection.

For this purpose, testers can use their knowledge to write formal specification for the SUT and use them in black-box coverage tools, or apply black-box coverage manually. For instance, if test t_1 covers a subset of covered items by t_2, and the main goal of t_1 is to check whether there is an exception thrown by the SUT while t_2 has a different goal, t_1 is not redundant. In other words, the inputs of two above tests are from different equivalence classes (i.e., a black-box coverage criterion should be applied).

Second, the verification phase of JUnit test methods should be analyzed separately. As explained in Section 3, this phase is independent of coverage information, and is thus a precision threat to redundancy detection. Assertion statements in JUnit tests should be compared to find if they cause redundancy or not. In some cases, the *actual* and *expected* values in assert statements have complicated data flow. In such cases, comparing assertions in verification phase would require sophisticated source code analysis (e.g., data flow analysis). For example, the actual outcomes of the two `assertEquals` statements (located in two test methods) in Figure 3 are the same: `adjusted.get()`. However, determining whether their expected outcomes (`a` and 1.5) have the same value or not would require data flow analysis in this example. Automating such an analysis is possible, but is challenging while in this step we use human tester for this purpose by leaving its automation as a future work.

Third, in addition to the SUT, all the external libraries used should be considered. However, as the source code of those libraries is not probably available, we need to instrument the class files in Java systems or to monitor coverage through the JVM. As per our investigations, automating this instrumentation and calculating coverage information for the external libraries and combining them with coverage information of the source code of the SUT is challenging and is thus considered as a future work. At this step, we propose the human tester to analyze the test code to find out how an external library affects test results and consider that in comparing test artifacts.

As explained previously, although it is possible to increase the degree of automation to cover the shortcoming of redundancy detection only based on limited number of coverage criteria, there is one main reason that does not allow full automation for this process, which is the lack of precise and formal specification for real world project. In other words, in the process of test redundancy detection the existence of human testers is necessary to confirm the real redundancy of those test artifacts detected as redundant by the system. The human tester has to conduct a manual inspection with guidelines proposed in this work and has to consider the three root causes to prevent false positive errors.

```
...
double a = getDefaultAdjusted(0);
...
assertEquals(a, adjusted.get(0));
...
```

```
...
assertEquals(1.5, adjusted.get(0));
...
```

Figure 3: The challenge of comparing assertions: excerpts from the test suite of Allelogram.

Using the three above guidelines helps testers to collaborate more effectively in the proposed redundancy detection process by analyzing test codes. Testers who have developed test artifacts are the best source of knowledge to decide about test redundancy by considering the above three lessons. However, other test experts can also use our methodology to find the redundancy of a test suite through manual inspection. For instance, in the experiment of this work, the input test suite was created by the developers of an open source project while the first author has performed the process of test redundancy detection.

5. A Collaborative Process for Redundancy Detection

To systematically achieve test redundancy detection with lower false-positive error, we propose a collaborative process between an automated redundancy detection system and human testers. The system will help the tester to inspect test artifacts with the least required amount of effort to find the actually redundant tests by using the benefits from coverage information while the fault detection capability of the reduced test suite is not reduced.

Figure 4 illustrates the activity diagram of the proposed interactive redundancy detection process. The input of this process is the original test suite of a SUT. Since human knowledge is involved, the precision of the inspection conducted by the human tester is paramount. If the tester follows the process and the three above guidelines carefully, the output would be a reduced test with the same fault detection effectiveness as the original one.

As the first step in this process, redundancy detection system uses a coverage tool to calculate coverage information, which is used later to calculate two redundancy metrics (discussed next).

Two redundancy metrics were proposed in [12]: Pair Redundancy and Suite Redundancy. The Pair Redundancy is defined between two test artifacts and is the ratio of covered items in SUT by the first test artifact with respect to the second one. In Suite Redundancy, this ratio is considered for one test artifact with respect to all other tests in the test suite.

Equations (1) and (2) define the Pair and Suite Redundancy metrics, respectively. In both of these equations, $CoveredItems_i(t_j)$ is the set of code items (e.g., statement and branch) covered by test artifact t_j, according to a given

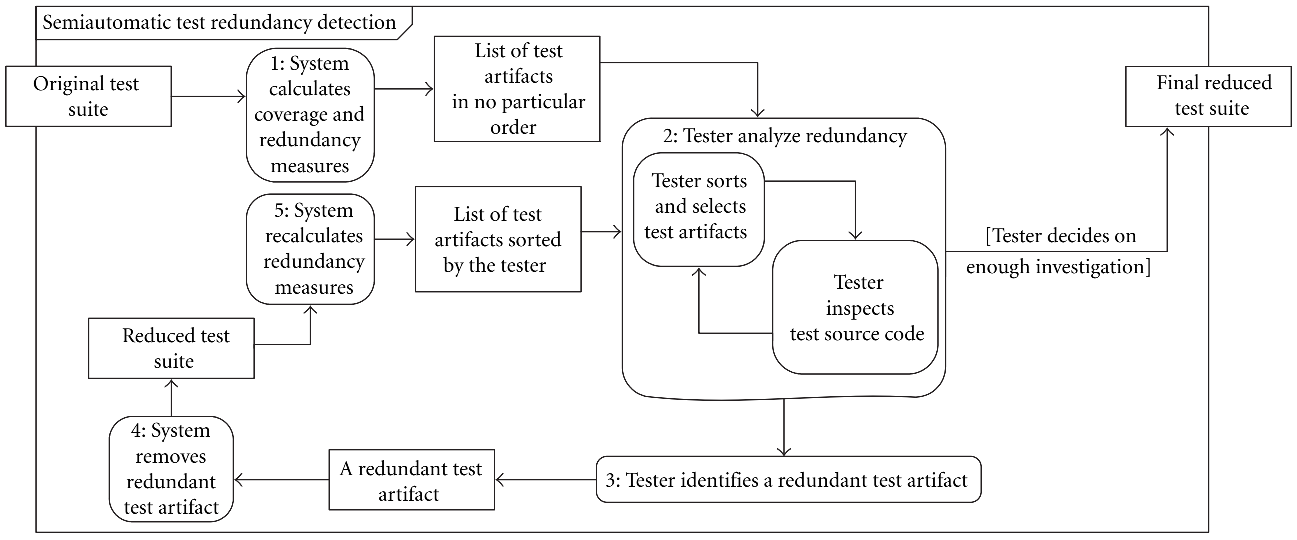

Semiautomatic test redundancy detection
Original test suite
1: System calculates coverage and redundancy measures
List of test artifacts in no particular order
2: Tester analyze redundancy
Tester sorts and selects test artifacts
Tester inspects test source code
[Tester decides on enough investigation]
Final reduced test suite
5: System recalculates redundancy measures
List of test artifacts sorted by the tester
Reduced test suite
4: System removes redundant test artifact
A redundant test artifact
3: Tester identifies a redundant test artifact

Figure 4: Proposed collaborative process for test redundancy detection.

coverage criterion i (e.g., statement coverage). *CoverageCriteria* in these two equations is the set of available coverage criteria used during the redundancy detection process.

Based on the design rationale of the above metrics, their values are always a real number in the range of $[0 \cdots 1]$. This enables us to measure redundancy in a quantitative domain (i.e., partial redundancy is supported too).

However, the results from [12] show that this type of partial redundancy is not precise and may mislead the tester in detecting the redundancy of the test. For instance, suppose that two JUnit test methods have similar setups with different exercises. If for example 90% of the test coverage is in the common setup the pair redundancy metrics would indicate that they are 90% redundant with respect to each other. However different exercises in these tests separate their goals and thus they should not be considered as redundant with respect to each other while 90% redundancy can mislead the tester about their redundancy.

Equation (1) shows Redundancy of test artifact ($\mathbf{t_j}$) with respect to another one ($\mathbf{t_k}$):

$$\begin{aligned}\mathrm{PR}\left(t_j, t_k\right) &= \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \cap CoveredItems_i(t_k) \right| \right) / \\ &\quad \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \right| \right),\end{aligned} \quad (1)$$

equation (2) shows Redundancy of one test artifact (t_j) with respect to all others:

$$\begin{aligned}\mathrm{SR}\left(t_j\right) &= \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \cap CoveredItems_i\left(\mathrm{TS} - t_j\right) \right| \right) / \\ &\quad \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \right| \right).\end{aligned} \quad (2)$$

However, partial redundancy concept can be useful in some cases to warn testers to refactor test code. To find these cases, in [12], we have offered to separate phases in a test case. As this approach is considered as a future work, in this work we do not consider partial redundancy concept. A test artifact can be redundant or nonredundant. The suite redundancy metric is used as a binary measure to separate test artifacts into these two groups: redundant, and nonredundant. If SR value of a test artifact = 1, that test is considered as redundant otherwise it is nonredundant.

In some cases, a test artifact does not cover any type of items (according to the considered coverage criteria). In [12], we have found that these cases may occur for various reasons, for example, (1) a test case may only cover items outside the SUT (e.g., an external library), (2) a test case may verify (assert) a condition without exercising anything from the SUT, or (3) a test method may be completely empty (developed by mistake). In these cases, the nominator and the denominator of both above metrics (PR and SR) will be zero (thus causing the 0-divide-by-0 problem). We assign the value of NaN (Not a Number) to the SR metric for these cases leaving them to be manually inspected to determine the reason.

After calculating coverage and redundancy metrics, the system prepares a list of test artifacts in no particular order. All the information about coverage ratios, number of covered items and redundancy metrics (both SR for each test and PR for each test pair) is available for exploration by the tester.

Step 2 in the process is the tester's turn. He/she should inspect the tests which are identified as a redundant test by the SR value (=1) to find out whether they are really redundant or not. This manual redundancy analysis should be performed for each test artifact separately. Therefore tester needs to choose a test from a set of candidate redundant tests.

The sequence in which test artifacts are inspected may affect the final precision of the process. Test sequencing often becomes important for an application that has internal state. Dependency between test artifacts may cause the erratic test smell in which one or more tests behave erratically (the test result depends on the result of other tests) [4]. However, in this work we do not consider this smell (our case study does not have this problem and thus we did not have any constraints for sequencing the test artifacts).

Our experience with manual redundancy detection in our case study (discussed in next section) helps us to find that the locality principle of test artifacts is an important factor that should be considered in test sequencing. In other words, for instance, test methods inside one test class have more likelihood of redundancy with respect to each other and should be inspected simultaneously.

There can be different strategies for ordering test artifacts and picking one to inspect at a time. One strategy can be defined according to number of covered items by each test artifact. As discussed next ascending and descending orders of number of coverage items each may have their own benefits.

A test expert may prefer to first choose a test artifact with higher redundancy probability. In this case, we hypothesize that the ascending order based on number of covered items is more suitable. The rationale behind this hypothesis is that the likelihood of covering fewer code items (e.g., statement, branch) by more than one test artifact is more than covering more items by the same test artifacts. Relationship between numbers of covered items by a test artifact with probability of redundancy of that test needs to be analyzed in an experiment. However, this is not the main goal of this paper and we leave it as a future work.

Descending order can have its own benefits. A test expert may believe that having test cases with more covered items would lead to the eager test smell (i.e., a test with too many assertions [22]). In this case, he/she would prefer to first analyze a test that covers more items in the SUT.

Finding a customized order of two above extreme cases by considering their benefits and costs is not discussed in this paper. Also other factors more than redundancy and coverage information may be useful in finding a proper test order.

Another strategy for sorting the existing test cases would be according to their execution time. If one of the objectives of reducing test suite is reducing the execution time, by this strategy test cases which need more time to be executed have more priority of redundancy candidates. However, we believe that in unit testing level execution time of test cases is not as important as other smells like being eager.

After picking appropriate test artifact, tester can use PR values of that test with respect to other tests. This information guides tester to inspect source code of that test case and compare it with source code of those tests with higher PR values. Without this information, manual inspection would take much more time from testers since he/she may not have any idea how to find another test to compare the source code together.

As discussed in Section 4, the main reason of need for human knowledge is to cover shortcomings of coverage-based redundancy detection. Therefore testers should be thoroughly familiar with these shortcomings and attempt at covering them.

After redundancy analysis, the test is identified as redundant or not. If it was detected as redundant by tester (Step 3), system removes it from original test set (Step 4). In this step, the whole collaborative process between system and tester should be repeated. Removing one test from test suite changes the value of $CoveredItems_i(TS - t_j)$ in (2). Therefore system should recalculate Suite Redundancy metric for all of the available tests (Step 5). In Section 6 we show how removing a redundant test detected by tester and recalculating the redundancy information can help the tester not to be misled by initial redundancy information and reduce the required effort of the tester.

Stopping condition of this process depends on tester's discretion. To find this stopping point, tester needs to compare the cost of process with savings in test maintenance costs resulting from test redundancy detection. Process cost at any point of the process can be measured by the time and effort that testers have spent in the process.

Test maintenance tasks have two types of costs which should be estimated: (1) costs incurred by updating (synchronizing) test code and SUT code, and (2) costs due to fixing integrity problems in test suite (e.g., one of two test cases testing the same SUT feature fails, while the other passes). Having redundant tests can lead testers to updating more than a test for each modification. Secondly, as a result of having redundant tests, the test suites would suffer from integrity issues, since the tester might have missed to update all the relevant tests.

To estimate the above two cost factors, one might perform change impact analysis on the SUT, and subsequently effort-prediction analysis (using techniques such as [23]) on SUT versus test code changes.

To decide about stopping point of the process, a tester would need to measure the process costs spent so far and to also estimate the maintenance costs containing both the above-discussed cost factors. By comparing them, he/she may decide to either stop or to continue the proposed process.

In the outset of this work, we have not systematically analyzed the above cost factors. As discussed before, we suggest testers to inspect all the tests with the value SR = 1 as many as possible. However, according to high number

Table 1: The size measures of Allelogram code.

SLOC	Number of packages	Number of classes	Number of methods
3,296	7	57	323

Table 2: The size measures of Allelogram test suite.

Test suite SLOC	Number of test packages	Number of test classes	Number of test methods
2,358	6	21	82

of false-positive errors, other tests in this category (with SR = 1) which were not inspected, should be considered as nonredundant. If the SR metric of a test artifact is less than 1, it means that there are some items in the SUT which are covered only by this test artifact. Thus, they should also be considered as nonredundant.

To automate the proposed process for test redundancy detection, we have modified the CodeCover coverage tool [20] to be able to measure our redundancy metrics. We refer to our extended tool as *TeReDetect* (Test Redundancy Detection tool). The tool shows a list of test artifacts containing coverage and redundancy information of each of them, it lets the tester to sort test artifacts according to his/her strategy (as explained before) and to introduce a real detected redundant test to the system for further metrics recalculation. After detecting a redundant test method, system automatically recalculates the redundancy metrics and updates the tester with new redundancy information for the next inspection iteration. A snapshot of the *TeReDetect* tool, during the process being applied to Allelogram, is shown in Figure 5. *TeReDetect* is an open source project (it has been extended to the SVN repository of CodeCover http://codecover.svn.sourceforge.net/svnroot/codecover). *TeReDetect* is not a standalone plug-in, rather it has been embedded inside the CodeCover plug-in. For instance, *ManualRedundancyView.java* is one of the extend-ed classes for our tool which is available from http://codecover.svn.sourceforge.net/svnroot/codecover/trunk/code/eclipse/src/org/codecover/eclipse/views/.

6. Case Study

6.1. Performing the Proposed Process. We used Allelogram [24], an open-source SUT developed in Java, as the object of our case study. Allelogram is a program for processing genomes and is used by biological scientists [24]. Table 1 shows the size measures of this system.

The unit test suite of Allelogram is also available through its project website [24] and is developed in JUnit. Table 2 lists the size metrics of its test suite. As the lowest implemented test level in JUnit is test method, we applied our redundancy detection process on the test method level in this SUT.

As the first step of proposed redundancy detection process, coverage metrics are measured. For this purpose, we used the CodeCover tool [20] in our experiment. This tool is an open-source coverage tool written in Java supporting the following four coverage criteria: statement, branch, condition (MC/DC), and loop. The loop coverage criterion, as supported by CodeCover, requires that each loop is executed 0 times, once, and more than once.

Table 3: Coverage information (%).

	Coverage (%)			
	Statement	Branch	Condition	Loop
Entire Allelogram	23.3	34.7	35.9	22.2
Without GUI components	68.0	72.9	71.4	43.0

Table 4: The percentage of fully redundant test methods.

Coverage criteria	Percentage of fully redundant test methods
Statement	77%
Branch	84%
Condition	83%
Loop	87%
All	69%

Table 3 shows the coverage metrics for our SUT. The first row in this table is the coverage ratios of the whole Allelogram system which are relatively low. We also looked at the code coverage of different packages in this system. Our analysis showed that the Graphical User Interface (GUI) package of this SUT is not tested (covered) at all by its test suite. This is most probably since JUnit is supposed to be used for unit testing and not GUI or functional testing. By excluding the GUI package from coverage measurement, we recalculated the coverage values shown in the second row of Table 3. These values show that the non-GUI parts of the system were tested quite thoroughly.

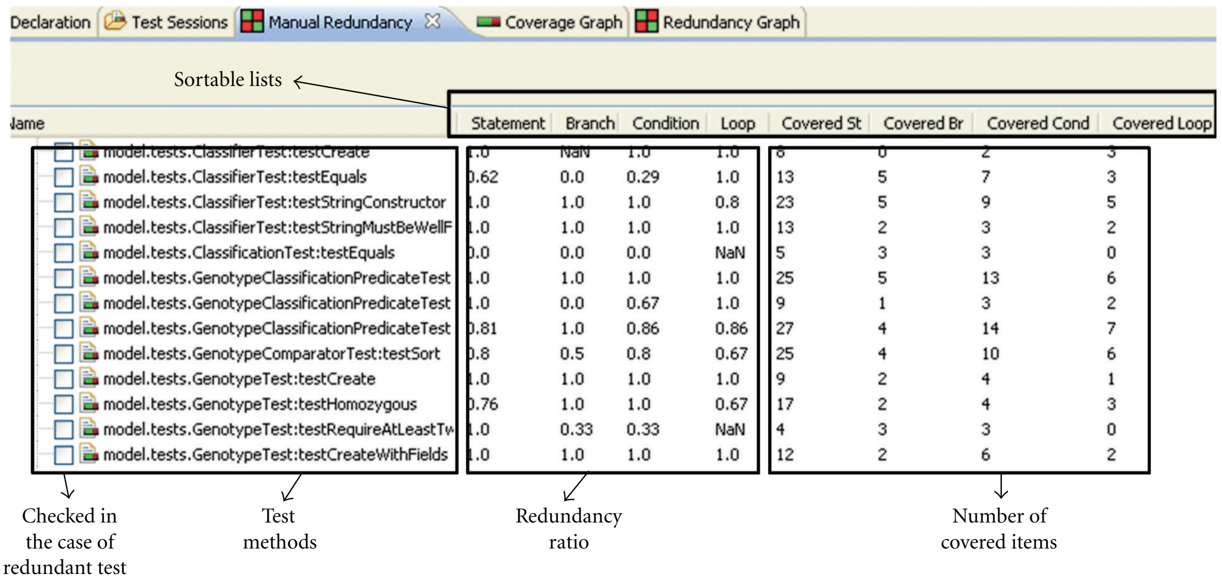
The next step in the process is the calculation of suite-level redundancy for each test method and pairwise redundancy for each pair of test methods in the test suite of our SUT.

To automate the measurement of redundancy of each test method using the two metrics defined in Section 5 ((1) and (2)), we have modified CodeCover to calculate the metrics and export them into a text file, once it executes a test suite.

Table 4 reports the percentage of fully redundant test methods (those with SR = 1) according to each coverage criterion and also by considering all of the criteria together.

As we expected, according to Table 4, ratio of full redundancy detected by considering each coverage criteria separately is higher than the case when all of them are considered. This confirms the fact that the more coverage criteria used in redundancy detection, the less false positive error can be achieved. In other words, *All* coverage criterion detects those tests as nonredundant that improve the coverage ratio values of at least one of the coverage criteria. As *All* criterion is more precise than the others, in the rest of our case study we consider the suite redundancy based on *All* criterion.

According to the suite redundancy result by considering all four coverage criteria (Table 4), 31% (100 − 69) of the tests in test suites of Allelogram are nonredundant. To confirm the nonredundancy of those methods, we randomly sampled



lame	Statement	Branch	Condition	Loop	Covered St	Covered Br	Covered Cond	Covered Loop
model.tests.ClassifierTest:testCreate	1.0	NaN	1.0	1.0	8	0	2	3
model.tests.ClassifierTest:testEquals	).62	0.0	0.29	1.0	13	5	7	3
model.tests.ClassifierTest:testStringConstructor	1.0	1.0	1.0	0.8	23	5	9	5
model.tests.ClassifierTest:testStringMustBeWellF	1.0	1.0	1.0	1.0	13	2	3	2
model.tests.ClassificationTest:testEquals	).0	0.0	0.0	NaN	5	3	3	0
model.tests.GenotypeClassificationPredicateTest	1.0	1.0	1.0	1.0	25	5	13	6
model.tests.GenotypeClassificationPredicateTest	1.0	0.0	0.67	1.0	9	1	3	2
model.tests.GenotypeClassificationPredicateTest	).81	1.0	0.86	0.86	27	4	14	7
model.tests.GenotypeComparatorTest:testSort	).8	0.5	0.8	0.67	25	4	10	6
model.tests.GenotypeTest:testCreate	1.0	1.0	1.0	1.0	9	2	4	1
model.tests.GenotypeTest:testHomozygous	).76	1.0	1.0	0.67	17	2	4	3
model.tests.GenotypeTest:testRequireAtLeastTw	1.0	0.33	0.33	NaN	4	3	3	0
model.tests.GenotypeTest:testCreateWithFields	1.0	1.0	1.0	1.0	12	2	6	2

Figure 5: Snapshot of the *TeReDetect* tool.

a set of test methods in this group and inspected them. We found few cases that seem as redundant tests which are in fact true-negative errors as reported in [12]. However, according to our inspection and code analysis, such test methods cover at least one coverable item not covered by any other test method. For instance, a test method named *testOneBin* in Allelogram covers a loop only once while some other test methods cover that loop more than one time. Therefore, loop redundancy of this method is slightly less than 1 (0.91) and thus detected as nonredundant by our redundancy metrics. For the same test method, the other types of redundancy considering only statement, branch, and condition coverage are 1. In fact, the above test cases contribute to loop coverage and we thus mark it as nonredundant since it covers a loop in a way (only once) not covered by other test methods.

Having a candidate set of redundant test methods (redundant tests based on *All* criterion: 69%), tester needs to decide about their order to inspect their source code. In this study, the first author (a graduate student of software testing) manually inspected the test methods. Recall the heuristics discussed in Section 5 about the sorting strategy of test method in the proposed process: test methods with fewer numbers of covered items have higher likelihood of being redundant. We thus decided to order the tests in the ascending order of the number of covered items (e.g., statement). In this case, we hoped to find redundant test methods sooner which may lead to a reduction in the search space (discussed next).

As the next step, manual inspection of a test was performed by comparing the source code of the test with other tests having high pair redundancy with the current one. The main focus of this step should be detecting redundancy by covering the shortcomings of coverage-based redundancy detection discussed in Section 5.

Redundancy of one test affects the redundancy of others. For instance, if test method A is redundant because it covers the same functionality covered by test method B (while there are no other tests to cover this functionality), test method B cannot be redundant at the same time. Therefore, while both of them are candidates for being redundant tests according to coverage information, but only one of them should be considered redundant finally. We refer to such effects as inter-test-method-redundancy effects

By only using redundancy information from the beginning step of the process, tester would need to keep track of all the tests previously detected as redundant during the process and apply the inter-test-method-redundancy effects by him/her self. However, recalculating the coverage information, after each redundancy detection, can reduce the search space (as explained next). Therefore, detecting redundant tests one by one and subsequently recalculating redundancy metrics increase precision and efficiency of the tester.

In this case study, we manually inspected the whole test suite of Allelogram. Figure 6 illustrates the whole process results by showing the size of five different test sets manipulated during the process. Those five test sets are discussed next.

We divide test methods into two categories: redundancy known and redundancy unknown. The test artifacts in the redundancy-unknown set are pending inspection to determine whether they are redundant or not (Set 1). Redundancy-known set contains redundant (Set 2) and nonredundant test sets whose decisions have been finalized. Furthermore, the set of nonredundant tests inside redundancy-known category contains three different sets: those identified through inspection (Set 3), those identified without inspection (Set 4), and the ones that were identified by system as nonredundant after nonredundancy has been detected through inspection (Set 5).

At the beginning of the process, by calculating redundancy metrics based on coverage information, test methods

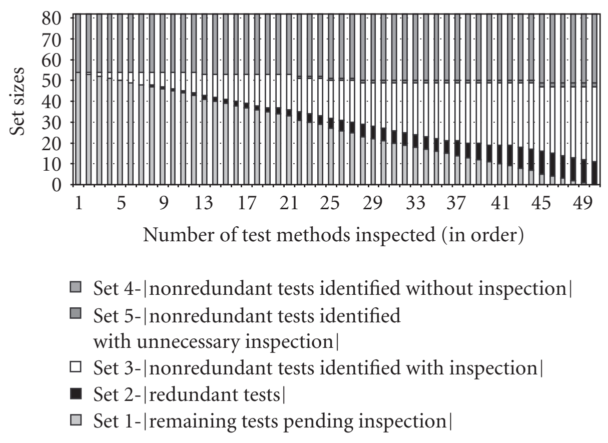


Figure 6: Labeling the test cases through the redundancy detection process.

are divided into two sets of *Nonredundant Tests without Inspection* and *Remaining Tests Pending Inspection* sets. As the figure shows, 28 test methods were recognized as nonredundant, while 54 (82 − 28) test methods needed to be inspected.

After each test method inspection, redundancy of that test is identified. This test method then leaves the *Remaining Tests Pending Inspection* set and Nonredundant test joins *Nonredundant Tests with Inspection* set while each redundant test joins *Redundant Tests* set. In the second case, redundancy metrics are recalculated.

In this case study, as shown in Figure 5, 11 test methods are recognized as redundant (test methods numbered in the x-axis as 7, 12, 19, 21, 24, 27, 36, 38, 40, 41, and 44). In these cases, new iterations of the process were performed by recalculating the redundancy metrics. In 5 cases (test methods numbered 12, 21, 24, 27, and 44), the recalculating led to search space reduction (5 test methods left the *Remaining Tests Pending Inspection* set and joined the *Nonredundant Tests without Inspection* set). In 2 of them (test methods 21 and 44), recalculating caused 2 test methods to leave *Nonredundant Tests with Inspection* set and join *Nonredundant Tests with Unnecessary Inspection* set.

At the beginning of the process, the size of the *Remaining Tests Pending Inspection* set was 54 (our initial search space). However, through the process, recalculating reduced the number of test methods that needed to be inspected to 49. In this case study, we ordered test methods in the ascending order of number of their covered items.

The final result of the process is a reduced test set containing 71 test methods instead of 82 (the original test suite of Allelogram). Stopping point of this process is considered by inspecting all the redundant candidate test methods (with SR = 1) and no cost estimation is applied for this purpose.

6.2. Evaluating the Proposed Process. To evaluate the preciseness of the proposed process, we considered the main purpose of test redundancy detection as discussed by many researchers. Test minimization should be performed in a way that the fault detection effectiveness of the test suite is preserved. Therefore, the process is successful if it does not reduce the fault detection capability.

One way to evaluate the above success factor of our test minimization approach is to inject probable faults in the SUT. Mutation is a technique that is widely used for this purpose ([25, 26]). The researches in [27, 28] show that the use of mutation operators is yielding trustworthy results and generated mutants can be used to predict the detection effectiveness of real faults.

In this work, we used the mutation analysis technique for the evaluation of the fault detection effectiveness of the reduced test suites generated by our technique. However, after completing this research project, we found out that, as another approach, we could also use the mutation analysis technique to detect test redundancy in a different alternative approach as follows. If the mutation scores of a given test suite with and without a particular test case are the same, then that test case is considered redundant. In other words, that test case does not kill (distinguish) any additional mutant. We plan to compare the above test redundancy detection approach with the one we conducted in this paper in a future work.

To inject simple faults into our case study, we used the MuClipse [29] tool which is a reincarnation of the MuJava [30] tool in the form of an Eclipse plug-in. Two main types of mutation operators are supported by MuClipse: method level (traditional) and class level (object oriented) [30].

To inject faults according to the traditional mutation operators, MuClipse replaces, inserts or deletes the primitive operators in the program. 15 different types of traditional mutation operators are available in MuClipse [29]. One example of this operators is the Arithmetic Operator Replacement (AOR) [31].

The strategy in object-oriented mutation operators is to handle all the possible syntactic changes for OO features by deleting, inserting, or changing the target syntactic element. 28 different types of OO mutation operators are available in MuClipse [29]. One example is Hiding variable deletion (IHD) which deletes a variable in a subclass that has the same name and type as a variable in the parent class [32].

All the available above mutation operators were used in this experiment. During this step, we found that MuClipse generates some mutants which failed to compile. These types of mutants are referred to as stillborn mutants which are syntactically incorrect and are killed by the compiler [29]. The total number of mutants for Allelogram that were not stillborn was 229.

To evaluate the fault detection effectiveness of the reduced test set by our proposed process compared to original test set, we calculated their mutation scores. We used MuClipse to execute all the created mutants with the two test sets (original and reduced). Table 5 shows the mutation score of three test sets: original test set, reduced test set only based on coverage information, and reduced test set through collaboration process with a tester.

The result shows that every mutant that is killed by original test set is killed by the reduced set (derived by

Table 5: Mutation score of three test suites for Allelogram.

Test set	Cardinality	Mutation score
Original	82	51%
Reduced (coverage based)	28	20%
Reduced (collaborative process)	71	51%

Table 6: Cost/benefit comparison.

	Cost	Benefit
Full automation	Low	Imprecise reduced set
Full manual	High	Precise reduced set
Semiautomated	Mid	Precise reduced set

the collaborative process) as well. In other words, the effectiveness of these two test sets is equal while the reduced set (solely based on coverage information) has 11 (82 − 71) less tests than the first one. That test suite thus has lower fault detection effectiveness.

Mutation score decreasing from 51% in original test set to 20% in the reduced set only based on coverage information confirms our discussion in Section 3 about impreciseness of test redundancy detection based only on coverage information.

7. Discussion

7.1. Effectiveness and Precision. Let us recall the main purpose of reducing the number of test cases in a test suite (Section 1): decreasing the cost of software maintenance. Thus, if the proposed methodology turns to be very time consuming, then it will not be worthwhile to be applied.

Although the best way to increase the efficiency of the process is to automate all required tasks, at this step we suppose that it is not practical to automate all of them. Thus, as we discuss next, human knowledge is currently needed in this process.

To perform manual inspection on test suite with the purpose of finding redundancy, testers need to spend time and effort on each test source code and compare them together. To decrease the amount of required effort, we have devised the proposed approach in a way to reduce the number of tests needed to be inspected (by using the suite redundancy metric). Our process also suggests useful information such as pair redundancy metric to help testers find other proper tests to compare with the test under inspection.

We believe that by using the above information, the efficiency of test redundancy detection has been improved. This improvement was seen on our case study while we first spent on average more than 15 minutes for each test method of Allelogram test suite before having our process. But inspecting them using the proposed process took on average less than 5 minutes per test method (the reason of time reduction is that in the later we knew other proper test methods to compare them with the current test). Since only one human subject (tester) performed the above two approaches, different parts of the Allelogram test suite were analyzed in each approach to avoid bias (due to learning and gaining familiarity) on time measurement.

However the above results are based on our preliminary experiment and it is thus inadequate to provide a general picture about the efficiency of the process. For a more systematic analysis in that direction, both time and effort should be measured more precisely with more than one subject on more than one object. Such an experiment is considered as a future work.

In addition to the efficiency of the process, precision of redundancy detection was also evaluated in our work. As explained in Section 6.2, this evaluation has been done in our case study by applying mutation technique. The result of analysis on one SUT confirmed the high precision of the process.

However, human's error is inevitable in collaborative processes which can affect the precision of the whole process. To decrease this type of error, the tester needs to be familiar with the written tests. Therefore, we suggest having the original test suite developers involved in the redundancy detection process if possible or that they be at least available for the possible questions during the process. In other words, a precise teamwork communication is required to detect correct test redundancy.

7.2. Cost/Benefit Analysis. According to above discussions, our redundancy detection technique has the following benefits.

(i) Reducing the size of test suite by keeping the fault detection effectiveness of that.

(ii) Preventing possible future integrity issues in the test suite.

(iii) Reducing test maintenance costs.

Different types of required costs in this process are summarized as follows.

(i) TeReDetect installation costs.

(ii) System execution time during the process (steps 1, 4, and 5 in Figure 4).

(iii) Redundancy analysis by human testers (steps 2 and 3 in Figure 4).

The first and second cost items are not considerable while the main part of the cost is about the third one which contains human efforts.

Table 6 shows an informal comparison of above costs and benefits in three approaches of full automation, full manual, and semiautomated process proposed in this paper. In the second and third approaches that human has a role, it is inevitable that the preciseness of human affects the benefits of the results.

7.3. Scalability. In large-scale systems with many LOC and test cases, it is not usually feasible to look at and analyze the test cases for the entire system. However, as mentioned before, in TeReDetect it is possible to select a subset of test suite and also a subset of SUT. This functionality of TeReDetect increases the scalability of this tool to a great extent by making it possible to divide the process of redundancy detection into separate parts and assign each part to a tester. However a precise teamwork communication is required to make the whole process successful.

Flexible stopping point of the proposed process is another reason for its scalability. According to the tester's discretion, the process of redundancy detection may stop after analyzing the subset of test cases or continue for all existing tests. For instance, in huge systems, by considering the cost of redundancy detection, project manager may decide to analyze only the critical part of the system.

7.4. Threats to Validity

7.4.1. External Validity. Two issues limit the generalization of our results. The first one is the subject representativeness of our case study. In this paper the process has been done by the first author (a graduate student). More than one subject should be experimented in this process to be able to compare their results to each other. Also, this subject knew the exact objective of the study which is a threat to the result. The second issue is the object program representativeness. We have performed the process and evaluate the result on one SUT (Allelogram). More objects should be used in experiments to improve the result. Also our SUT is a random project chosen from the open source community. Other industrial programs with different characteristics may have different test redundancy behavior.

7.4.2. Internal Validity. The result about efficiency and precision of the proposed process might be from some other factors which we had no control or had not measured. For instance, the bias and knowledge of the tester while trying to find redundancy can be such a factor.

8. Conclusion and Future Works

Measuring and removing test redundancy can prevent the integrity issues of test suites and decrease the cost of test maintenance. Previous works on test set minimization believed that coverage information is useful resource to detect redundancy.

To evaluate the above idea we performed an experiment in [12]. The result shows that coverage information is not enough knowledge for detecting redundancy according to fault detection effectiveness. However, this information is a very useful starting point for further manual inspection by human testers.

Root-cause analysis of above observation in [12] has helped us to improve the precision of redundancy detection by covering the shortcomings in the process proposed in this paper.

We proposed a collaborative process between human testers and redundancy system based on coverage information. We also performed an experiment with that process on a real java project. This in turn led us to find out that the sharing the knowledge between the human user and the system can be useful for the purpose of test redundancy detection. We conclude that test redundancy detection can be performed more effectively when it is done in an interactive process.

The result of the case study performed in this paper shows that fault detection effectiveness of the reduced set is the same as the original test set while the cost of test maintenance for reduced one is less than the other (since the size of the first set is less than the second one).

The efficiency of this process in terms of time and effort is improved comparing to the case of manual inspection for finding test redundancy without this proposed process.

In this paper, the efficiency factor was discussed qualitatively. Therefore measuring precise time and efforts spent in this process is considered as a future experiment.

Finding the stopping point of the process needs maintenance and effort cost estimation which is not studied thoroughly in this work and is also considered as a future work.

As explained in Section 5, the order of the tests inspected in the proposed process can play an important role in the test reduction result. In this work we suggested a few strategies with their benefits to order the test while this needs to be studied more precisely. Also, test sequential constraints such as the case of dependent test cases are not discussed in this work.

Visualization of coverage and redundancy information can also improve the efficiency of this process extensively. We are now in the process of developing such a visualization technique to further help human testers in test redundancy detect processes.

In addition to above, some tasks which are now done manually in this proposed process could be automated in future works. One example is the automated detection of redundancy in the verification phase of JUnit test methods which will most probably require the development of sophisticated code analysis tools to compare the verification phase of two test methods.

Acknowledgments

The authors were supported by the Discovery Grant no. 341511-07 from the Natural Sciences and Engineering Research Council of Canada (NSERC). V. Garousi was further supported by the Alberta Ingenuity New Faculty Award no. 200600673.

References

[1] S. G. Eick, T. L. Graves, A. F. Karr, U. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[2] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering (ICSE '94)*, pp. 279–287, Sorrento, Italy, May 1994.

[3] B. V. Rompaey, B. D. Bois, and S. Demeyer, "Improving test code reviews with metrics: a pilot study," Tech. Rep., Lab

on Reverse Engineering, University of Antwerp, Antwerp, Belgium, 2006.

[4] G. Meszaros, *xUnit Test Patterns, Refactoring Test Code*, Addison-Wesley, Reading, Mass, USA, 2007.

[5] A. Deursen, L. Moonen, A. Bergh, and G. Kok, “Refactoring test code,” in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP ’01)*, Sardinia, Italy, May 2001.

[6] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An empirical study of the effects of minimization on the fault detection capabilities of test suites,” in *Proceedings of the Conference on Software Maintenance (ICSM ’98)*, pp. 34–43, Bethesda, Md, USA, November 1998.

[7] M. J. Harrold, R. Gupta, and M. L. Soffa, “Methodology for controlling the size of a test suite,” *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[8] J. A. Jones and M. J. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[9] A. J. Offutt, J. Pan, and J. M. Voas, “Procedures for reducing the size of coverage-based test sets,” in *Proceedings of the 11th International Conference on Testing Computer Software (ICTCS ’95)*, pp. 111–123, Washington, DC, USA, June 1995.

[10] W. E. Wong, J. R. Morgan, S. London, and A. P. Mathur, “Effect of test set minimization on fault detection effectiveness,” *Software—Practice & Experience*, vol. 28, no. 4, pp. 347–369, 1998.

[11] W. E. Wong, J. R. Horgan, A. P. Mathur, and Pasquini, “Test set size minimization and fault detection effectiveness: a case study in a space application,” in *Proceedings of the IEEE Computer Society’s International Computer Software and Applications Conference (COMPSAC ’97)*, pp. 522–528, Washington, DC, USA, August 1997.

[12] N. Koochakzadeh, V. Garousi, and F. Maurer, “Test redundancy measurement based on coverage information: evaluations and lessons learned,” in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST ’09)*, pp. 220–229, Denver, Colo, USA, April 2009.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, Calif, USA, 1990.

[14] A. Ngo-The and G. Ruhe, “A systematic approach for solving the wicked problem of software release planning,” *Soft Computing*, vol. 12, no. 1, pp. 95–108, 2008.

[15] R. Milner, “Turing, computing, and communication,” in *Interactive Computation: The New Paradigm*, pp. 1–8, Springer, Berlin, Germany, 2006.

[16] F. Arbab, “Computing and Interaction,” in *Interactive Computation: The New Paradigm*, pp. 9–24, Springer, Berlin, Germany, 2006.

[17] M. Takaai, H. Takeda, and T. Nishida, “A designer support environment for cooperative design,” *Systems and Computers in Japan*, vol. 30, no. 8, pp. 32–39, 1999.

[18] Parasoft Corporation, “Parasoft Jtest,” October 2009, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[19] IBM Rational Corporation, “Rational manual tester,” January 2009, http://www-01.ibm.com/software/awdtools/tester/manual/.

[20] T. Scheller, “CodeCover,” 2007, http://codecover.org/.

[21] nitinpatil, “JFeature,” June 2009, https://jfeature.dev.java.net/.

[22] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, “On the detection of test smells: a metrics-based approach for general fixture and eager test,” *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–816, 2007.

[23] L. C. Briand and J. Wüst, “Modeling development effort in object-oriented systems using design properties,” *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 963–986, 2001.

[24] C. Manaster, “Allelogram,” August 2008, http://code.google.com/p/allelogram/.

[25] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: help for the practicing programmer,” *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[26] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

[27] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[28] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of the 27th International Conference on Software Engineering (ICSE ’05)*, pp. 402–411, 2005.

[29] B. Smith and L. Williams, “MuClipse,” December 2008, http://muclipse.sourceforge.net/.

[30] J. Offutt, Y. S. Ma, and Y. R. Kwon, “MuJava,” December 2008, http://cs.gmu.edu/∼offutt/mujava/.

[31] Y. S. Ma and J. Offutt, “Description of method-level mutation operators for java,” December 2005, http://cs.gmu.edu/∼offutt/mujava/mutopsMethod.pdf.

[32] Y. S. Ma and J. Offutt, “Description of class mutation mutation operators for java,” December 2005, http://cs.gmu.edu/∼offutt/mujava/mutopsClass.pdf.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 323429, 7 pages
doi:10.1155/2010/323429

Research Article

A Strategy for Automatic Quality Signing and Verification Processes for Hardware and Software Testing

Mohammed I. Younis and Kamal Z. Zamli

School of Electrical and Electronics, Universiti Sains Malaysia, 14300 Nibong Tebal, Malaysia

Correspondence should be addressed to Mohammed I. Younis, younismi@gmail.com

Received 14 June 2009; Revised 4 August 2009; Accepted 20 November 2009

Academic Editor: Phillip Laplante

We propose a novel strategy to optimize the test suite required for testing both hardware and software in a production line. Here, the strategy is based on two processes: Quality Signing Process and Quality Verification Process, respectively. Unlike earlier work, the proposed strategy is based on integration of black box and white box techniques in order to derive an optimum test suite during the Quality Signing Process. In this case, the generated optimal test suite significantly improves the Quality Verification Process. Considering both processes, the novelty of the proposed strategy is the fact that the optimization and reduction of test suite is performed by selecting only mutant killing test cases from cumulating t-way test cases. As such, the proposed strategy can potentially enhance the quality of product with minimal cost in terms of overall resource usage and time execution. As a case study, this paper describes the step-by-step application of the strategy for testing a 4-bit Magnitude Comparator Integrated Circuits in a production line. Comparatively, our result demonstrates that the proposed strategy outperforms the traditional block partitioning strategy with the mutant score of 100% to 90%, respectively, with the same number of test cases.

1. Introduction

In order to ensure acceptable quality and reliability of any embedded engineering products, many inputs parameters as well as software/hardware configurations need to be tested against for conformance. If the input combinations are large, exhaustive testing is next to impossible due to combinatorial explosion problem.

As illustration, consider the following small-scale product, a 4-bit Magnitude Comparator IC. Here, the Magnitude Comparator IC consists of 8 bits for inputs and 3 bits for outputs. It is clear that each IC requires 256 test cases for exhaustive testing. Assuming that each test case takes one second to run and be observed, the testing time for each IC is 256 seconds. If there is a need to test one million chips, the testing process will take more than 8 years using a single line of test.

Now, let us assume that we received an order of delivery for one million qualified (i.e., tested) chips within two weeks. As an option, we can do parallel testing. However, parallel testing can be expensive due to the need for 212 testing lines. Now, what if there are simultaneous multiple orders? Here, as the product demand grows in numbers, parallel testing can also become impossible. Systematic random testing could also be another option. In random testing, test cases are chosen randomly from some input distribution (such as a uniform distribution) without exploiting information from the specification or previously chosen test cases. More recent results have favored partition testing over random testing in many practical cases. In all cases, random testing is found to be less effective than the investigated partition testing methods [1].

A systematic solution to this problem is based on *Combinatorial Interaction Testing* (CIT) strategy. The CIT approach can systematically reduce the number of test cases by selecting a subset from exhaustive testing combination based on the strength of parameter interaction coverage (t) [2]. To illustrate the CIT approach, consider the web-based system example (see Table 1) [3].

Considering full strength interaction $t = 4$ (i.e., interaction of all parameters) for testing yields exhaustive combinations of $3^4 = 81$ possibilities. Relaxing the interaction strength to $t = 3$ yields 27 test cases, a saving of nearly 67 percent. Here, all the 3-way interaction elements are all covered by at least

Table 1: Web-based system example.

Parameter 1	Parameter 2	Parameter 3	Parameter 4
Netscape	Windows XP	LAN	Sis
IE	Windows VISTA	PPP	Intel
Firefox	Windows 2008	ISDN	VIA

one test. If the interaction is relaxed further to $t = 2$, then the number of combination possibilities is reduced even further to merely 9 test cases, a saving of over 90 percent.

In the last decade, CIT strategies were focused on 2-way (pairwise) testing. More recently, several strategies (e.g., Jenny [4], TVG [5], IPOG [6], IPOD [7], IPOF [8], DDA [9], and GMIPOG [10]) that can generate test suite for high degree interaction ($2 \leq t \leq 6$).

Being predominantly black box, CIT strategy is often criticized for not being efficiently effective for highly interacting parameter coverage. Here, the selected test cases sometimes give poor coverage due to the wrong selection of parameter strength. In order to address this issue, we propose to integrate the CIT strategy with that of fault injection strategy. With such integration, we hope to effectively measure the effectiveness of the test suite with the selection of any particular parameter strength. Here, the optimal test case can be selected as the candidate of the test suite only if it can help detect the occurrence of the injected fault. In this manner, the desired test suite is the most optimum for evaluating the *System Under Test* (SUT).

The rest of this paper is organized as follows. Section 2 presents related work on the state of the art of the applications of t-way testing and fault injection tools. Section 3 presents the proposed minimization strategy. Section 4 gives a step-by-step example as prove of concept involving the 4-bit Magnitude Comparator. Section 5 demonstrates the comparison with our proposed strategy and the traditional block partitioning strategy. Finally, Section 6 describes our conclusion and suggestion for future work.

2. Related Work

Mandl was the first researcher who used pairwise coverage in the software industry. In his work, Mandl adopts orthogonal Latin square for testing an Ada compiler [11]. Berling and Runeson use interaction testing to identify real and false targets in target identification system [12]. Lazic and Velasevic employed interaction testing on modeling and simulation for automated target-tracking radar system [13]. White has also applied the technique to test graphical user interfaces (GUIs) [14]. Other applications of interaction testing include regression testing through the graphical user interface [15] and fault localization [16, 17]. While earlier work has indicated that pairwise testing (i.e., based on 2-way interaction of variables) can be effective to detect most faults in a typical software system, a counter argument suggests such conclusion infeasible to generalize to all software system faults. For example, a test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [18–20]. In other works it is found that 100% of faults are detectable by a relatively low degree of interaction, typically 4-way combinations [21–23].

More recently, a study by *The National Institute of Standards and Technology* (NIST) for error-detection rates in four application domains included medical devices, a Web browser, an HTTP server, and a NASA-distributed database reported that 95% of the actual faults on the test software involve 4-way interaction [24, 25]. In fact, according to the recommendation from NIST, almost all of the faults detected with 6-way interaction. Thus, as this example illustrates, system faults caused by variable interactions may also span more than two parameters, up to 6-way interaction for moderate systems.

All the aforementioned related work in CIT applications highlighted the potential of adopting the CIT strategies for both software/hardware testing. While the CIT strategies can significantly partition the exhaustive test space into manageable manner, additional reduction can still be possible particularly by systematically examining the effectiveness of each test case in the test suite, that is, by exploiting fault injection techniques.

The use of fault injection techniques for software and hardware testing is not new. Tang and Chen [26], Boroday [27], and Chandra et al. [28] study circuit testing in hardware environment, proposing test coverage that includes each 2^t of the input settings for each subset of t inputs. Seroussi and Bshouty [29] give a comprehensive treatment for circuit testing. Dumer [30] examines the related question of isolating memory faults and uses binary covering arrays. Finally, Ghosh and Kelly give a survey to include a number of studies and tools that have been reported in the area of failure mode identification [31]. These studies help in the long-term improvement of the software development process as the recurrence of the same failures can be prevented. Failure modes can be specific to a system or be applicable to systems in general. They can be used in testing for fault tolerance, as realistic faults are needed to perform effective fault injection testing. Additionally, Ghosh and Kelly also describe a technique that injects faults in Java software by manipulating the bytecode level for third party software components used by the developers.

3. Proposed Strategy

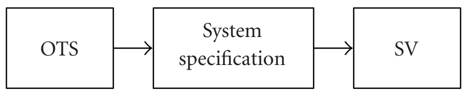
The proposed strategy consists for two processes, namely, *Test Quality Signing* (TQS) process and *Test Verification* process (TV). Briefly, the TQS process deals with optimizing the selection of test suite for fault injection as well as performs the actual injection whilst the TV process analyzes for conformance (see Figure 1).

As implied earlier, the TQS process aims to derive an effective and optimum test suite and works as follows.

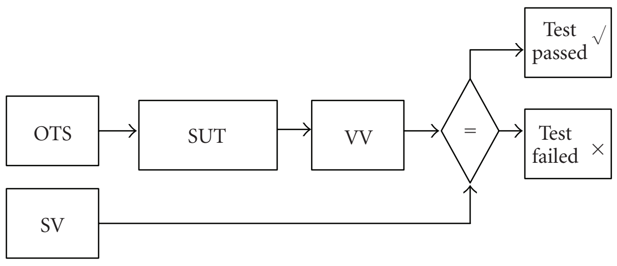
(1) Start with an empty *Optimized Test Suite* (OTS), and empty *Signing Vector* (SV).

(2) Select the desired software class (for software testing). Alternatively, build an equivalent software class for the *Circuit Under Test* (CUT) (for hardware testing).

(3) Store these faults in *fault list* (FL).



(a) Quality signing process



(b) Quality verification process

FIGURE 1: The quality signing and verification processes.

(4) Inject the class with all possible faults.

(5) Let N be maximum number of parameters.

(6) Initialize CIT strategy with strength of coverage (t) equal one (i.e., $t = 1$).

(7) Let CIT strategy partition the exhaustive test space. The portioning involves generating one test case at a time for t coverage. If t coverage criteria are satisfied, then $t = t + 1$.

(8) CIT strategy generates one *Test Case* (TC).

(9) Execute TC.

(10) If TC detects any fault in FL, remove the detected fault(s) from FL, and add TC and its specification output(s) to OTS and SV, respectively.

(11) If FL is not empty or $t <= \text{N}$, go to 7.

(12) The desired optimized test suite and its corresponding output(s) are stored in OTS and SV, respectively.

The TV process involves the verification of fault free for each unit. TV process for a single unit works as follows.

(1) for $i = 1..\text{Size(OTS)}$ each TC in OTS do:

   (a) Subject the SUT to TC[i], store the output in *Verification Vector* VV[i].

   (b) If VV[i] = SV [i], continue. Else, go to 3.

(2) Report that the cut has been passing in the test. Go to 4.

(3) Report that the cut has failed the test.

(4) The verification process ends.

As noted in the second step of the TQS process, the rationale for taking equivalent software class for the CUT is to ensure that the cost and control of the fault injection be more practical and manageable as opposed to performing it directly to a real hardware circuit. Furthermore, the derivation of OTS is faster in software than in hardware. Despite using equivalent class for the CUT, this verification process should work for both software and hardware systems. In fact, it should be noted that the proposed strategy could also be applicable in the context of N-version programming (e.g., the assessment of student programs for the same assignment) and not just hardware production lines. The concept of N-version programming was introduced by Chen and Avizienis with the central conjecture that the "independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program" [32, 33].

4. Case Study

As proof of concept, we have adopted GMIPOG [10] as our CIT strategy implementation, and MuJava version 3 (described in [34, 35]) as our fault injection strategy implementation.

Briefly, GMIPOG is a combinatorial test generator based on specified inputs and parameter interaction. Running on a Grid environment, GMIPOG adopts both the horizontal and vertical extension mechanism (i.e., similar to that of IPOG [6]) in order to derive the required test suite for a given interaction strength. While there are many useful combinatorial test generators in the literature (e.g., Jenny [3], TConfig [4], TVG [5], IPOG [6], IPOD [7], IPOF [8], DDA [9]), the rationale for choosing GMIPOG is the fact that it supports high degree of interaction and can be run in cumulative mode (i.e., support one-test-at-a-time approach with the capability to vary t automatically until the exhaustive testing is reached).

Complementary to GMIPOG, MuJava is a fault injection tool that permits mutated Java code (i.e., based on some defined operators) to be injected into the running Java program. Here, the reason for choosing MuJava stemmed from the fact that it is a public domain Java tool freely accessible for download in the internet [35].

Using both tools (i.e., GMIPOG and MuJava), a case study problem involving a 4-bit Magnitude Comparator IC will be discussed here in order to evaluate the proposed strategy. A 4-bit Magnitude Comparator consists of 8 inputs (two four bits inputs, namely, a0. . .a3, and b0. . .b3. where a0 and b0 are the most significant bits), 4 xnor gates (or equivalent to 4xor with 4 not gates), five not gates, five and gates, three or gates, and three outputs. The actual circuit realization of the Magnitude Comparator is given in Figure 2. Here, it should be noted that this version of the circuit is a variant realization (implementation) of the Magnitude Comparator found in [36]. The equivalent class of the Magnitude Comparator is given in Figure 3 (using the Java-programming language).

Here, it is important to ensure that the software implementation obeys the hardware implementation strictly. By doing so, we can undertake the fault injection and produce the OTS in the software domain without affecting the logical of relation and parameter interactions of the hardware implementation.

Now, we apply the TQS process; as illustrated in Section 3. Here, there are 80 faults injected in the system. To assist our work, we use GMIPOG [10] to produce the TC

Table 2: Derivation of OTS for the 4-bit Magnitude Comparator.

t =	Cumulative Test Size	Live Mutant	Killed Mutant	% Mutant Score	Effective test size
1	2	15	65	81.25	2
2	9	5	75	93.75	6
3	24	2	78	97.50	8
4	36	0	80	100.00	9

Table 3: OTS and SV for the 4-bit Magnitude Comparator.

#TC	OTS TC (a0...a3, b0...b3)	SV Outputs ($A > B$, $A = B$, $A < B$)	Accumulative faults detected/80
1	FFFFFFFF	F T F	53
2	TTTTTTTT	F T F	65
3	FTTTTTTT	F F T	68
4	TTFTFTFT	T F F	71
5	TTFFTFTT	T F F	72
6	TTTFTTFF	T F F	75
7	TTFTTTTF	F F T	77
8	FFTTTTTF	F F T	78
9	TFTTTFTF	T F F	80

Table 4: Cumulative faults detected when $x = 7$.

#TC	TC (a0...a3, b0...b3)	Cumulative faults detected /80
1	FFFFFFFF	53
2	FFFFFTTT	54
3	FFFFTTTT	54
4	FTTTFFFF	59
5	FTTTFTTT	67
6	FTTTTTTT	70
7	TTTTFFFF	71
8	TTTTFTTT	71
9	TTTTTTTT	72

Table 5: Cumulative faults detected when x is randomly selective.

#TC	TC (a0...a3, b0...b3)	Cumulative faults detected /80
1	FFFFFFFF	53
2	FFFFFTTF	55
3	FFFFTTTT	55
4	TFTTFFFF	59
5	TFFTFTTT	61
6	TFTFTTTT	61
7	TTTTFFFF	61
8	TTTTTFFF	64
9	TTTTTTTT	72

in a cumulative mode. Following the steps in TQS process, Table 2 demonstrates the derivation of OTS. Here, it should be noted that the first 36 test cases can remove all the faults. Furthermore, only the first 12 test cases when $t = 4$ are needed to catch that last two live mutants. The efficiency of integration GMIPOG with MuJava can be observed (by taken only the effective TC) in the last column in Table 2.

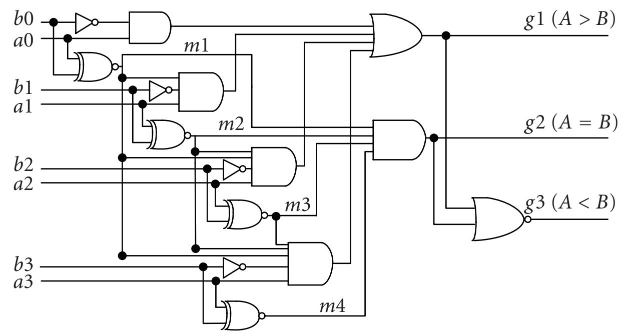


Figure 2: Schematic diagram for the 4-bit magnitude comparator.

Table 3 gives the desired OTS and SV, where T and F represent true and false, respectively. In this case, TQS process reduces the test size to nine test cases only, which significantly improves the TV process.

To illustrate how the verification process is done (see Figure 2), assume that the second output (i.e., $A = B$) is out-of-order (i.e., malfunction). Suppose that $A = B$ output is always on (i.e., short circuit to "VCC"). This fault cannot be detected as either TC1 or TC2 (according to Table 2). Nevertheless, when TC3, the output vector ("VV") of faulty IC, is FTT, and the SV is FFT, the TV process can straightforwardly detects that the IC is malfunctioning (i.e., cut fails).

To consider the effectiveness of the proposed strategy in the production line, we return to our illustrative example given in Section 1. Here, the reduction of exhaustive test from 256 test cases to merely nine test cases is significantly important. In this case, the TV process requires only 9 seconds instead of 256 seconds for considering all tests. Now, using one testing line and adopting our strategy for two

```
public class Comparator {
//Comparator takes two four bits numbers (A&B), where A = a0a1a2a3
//B=b0b1b2b3. Here, a0 and b0 are the most significant bits.
//The function returns an output string that s
//g1, g2, and g3 represent the logical outputs of A > B, A = B, and A < B respectively.
//the code symbols (!, ∧, |, and &)
//represent the logical operator for Not, Xor, Or, and And respectively.
        public static String compare
            (boolean a0, boolean a1, boolean a2, boolean a3,
            boolean b0, boolean b1, boolean b2, boolean b3) {
            boolean   g1,g2,g3;
            boolean   m1,m2,m3,m4;
              String s = null;
              m1 =!(a0 ∧ b0);
              m2 =!(a1 ∧ b1);
              m3 =!(a2 ∧ b2);
              m4 =!(a3 ∧ b3);
              g1 = (a0 &!b0)| (m1&a1 &!b1) |(m1&m2&a2 &!b2)| (m1&m2 &m3&a3 &!b3);
              g2 =  (m1&m2 &m3&m4);
              g3 =!(g1|g2);
              s = g1 +'''' +g2 +'''' +g3; // just to return output strings for MuJava compatibility
              return s;
    }
}
```

Figure 3: Equivalent class Java program for the 4-bit magnitude comparator.

weeks can test (14X24X60X60/9 = 134400) chips. Hence, to deliver one millions tested ICs' during these two weeks, our strategy requires eight parallel testing lines instead of 212 testing lines (if the test depends on exhaustive testing strategy). Now, if we consider the saving efforts factor as the size of exhaustive test suite minus optimized test suite to the size of exhaustive test suite, we would obtain the saving efforts factor of $256 - 9/256 = 96.48\%$.

5. Comparison

In this section, we demonstrate the possible test reduction using block partitioning approach [1, 37] for comparison purposes. Here, the partitions could be two 4-bit numbers, with block values =0, $0 < x < 15$, =15 and 9 test cases would give all combination coverage. In this case, we have chosen $x = 7$ as a representative value. Additionally, we have also run a series of 9 tests where x is chosen at random between 0 and 15. The results of the generated test cases and their corresponding cumulative faults detected are tabulated in Tables 4 and 5, respectively.

Referring to Tables 4 and 5, we observe that block partitioning techniques have achieved the mutant score of 90%. For comparative purposes, it should be noted that our proposed strategy achieved a mutant score of 100% with the same number of test cases.

6. Conclusion

In this paper, we present a novel strategy for automatic quality signing and verification technique for both hardware and software testing. Our case study in hardware production line demonstrated that the proposed strategy could improve the saving efforts factor significantly. In fact, we also demonstrate that our proposed strategy outperforms the traditional block partitioning strategy in terms of achieving better mutant score with the same number of test cases. As such, we can also potentially predict benefits in terms of the time and cost saving if the strategy is applied as part of software testing endeavor.

Despite giving a good result (i.e., as demonstrated in earlier sections), we foresee a number of difficulties as far as adopting mutation testing is concerned. In general, mutation testing does not scale well. Applying mutation testing in large programs can result in very large numbers of mutations making it difficult to find a good test suite to kill all the mutants. We are addressing this issue as part of our future work by dealing with variable strength interaction testing.

Finally, we also plan to investigate the application of our proposed strategy for computer-aided software application and hardware design tool.

Acknowledgments

The authors acknowledge the help of Jeff Offutt, Jeff Lei, Raghu Kacker, Rick Kuhn, Myra B. Cohen, and Sudipto Ghosh for providing them with useful comments and the background materials. This research is partially funded by the USM: Post Graduate Research Grant—T-Way Test Data Generation Strategy Utilizing Multicore System, USM GRID—The Development and Integration of Grid Services & Applications, and the fundamental research grants—"Investigating Heuristic Algorithm to Address Combinatorial Explosion Problem" from the Ministry of Higher Education (MOHE). The first author, Mohammed I. Younis, is the USM fellowship recipient.

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," Tech. Rep. ISETR-04-05, GMU, July 2004.

[2] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "Algebraic strategy to generate pairwise test set for prime number parameters and variables," in *Proceedings of the International Symposium on Information Technology (ITSim '08)*, vol. 4, pp. 1662–1666, IEEE Press, Kuala Lumpur, Malaysia, August 2008.

[3] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "IRPS: an efficient test data generation strategy for pairwise testing," in *Proceedings of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES '08)*, vol. 5177 of *Lecture Notes in Computer Science*, pp. 493–500, 2008.

[4] Jenny tool, June 2009, http://www.burtleburtle.net/bob/math/.

[5] TVG tool, June 2009, http://sourceforge.net/projects/tvg/.

[6] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: a general strategy for T-way software testing," in *Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems*, pp. 549–556, Tucson, Ariz, USA, March 2007.

[7] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG-IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[8] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, pp. 287–297, 2008.

[9] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[10] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "A strategy for grid based T-Way test data generation," in *Proceedings the 1st IEEE International Conference on Distributed Frameworks and Application (DFmA '08)*, pp. 73–78, Penang, Malaysia, October 2008.

[11] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[12] T. Berling and P. Runeson, "Efficient evaluation of multifactor dependent system performance using fractional factorial design," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 769–781, 2003.

[13] L. Lazic and D. Velasevic, "Applying simulation and design of experiments to the embedded software testing process," *Software Testing Verification and Reliability*, vol. 14, no. 4, pp. 257–282, 2004.

[14] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '00)*, pp. 110–121, IEEE Computer Society, San Jose, Calif, USA, 2000.

[15] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proceedings of the 9th Joint European Software Engineering Conference (ESEC) and the 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)*, pp. 118–127, ACM, September 2003.

[16] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[17] M. S. Reorda, Z. Peng, and M. Violanate, Eds., *System-Level Test and Validation of Hardware/Software Systems*, Advanced Microelectronics Series, Springer, London, UK, 2005.

[18] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of AT&T PMX/StarMail using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[19] S. R. Dalal, A. Jain, N. Karunanithi, et al., "Model-based testing in practice," in *Proceedings of the International Conference on Software Engineering*, pp. 285–294, 1999.

[20] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, 2002.

[21] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: an analysis of 15 years of recall data," *International Journal of Reliability, Quality, and Safety Engineering*, vol. 8, no. 4, pp. 351–371, 2001.

[22] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, pp. 91–95, IEEE Computer Society, December 2002.

[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[24] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW '06)*, pp. 153–158, April 2006.

[25] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[26] D. T. Tang and C. L. Chen, "Iterative exhaustive pattern generation for logic testing," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 212–219, 1984.

[27] S. Y. Boroday, "Determining essential arguments of Boolean functions," in *Proceedings of the International Conference on Industrial Mathematics (ICIM '98)*, pp. 59–61, Taganrog, Russia, 1998.

[28] A. K. Chandra, L. T. Kou, G. Markowsky, and S. Zaks, "On sets of Boolean n-vectors with all k-projections surjective," *Acta Informatica*, vol. 20, no. 1, pp. 103–111, 1983.

[29] G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Transactions on Information Theory*, vol. 34, no. 3, pp. 513–522, 1988.

[30] I. I. Dumer, "Asymptotically optimal codes correcting memory defects of fixed multiplicity," *Problemy Peredachi Informatskii*, vol. 25, pp. 3–20, 1989.

[31] S. Ghosh and J. L. Kelly, "Bytecode fault injection for Java software," *Journal of Systems and Software*, vol. 81, no. 11, pp. 2034–2043, 2008.

[32] A. A. Avizienis, *The Methodology of N-Version Programming*, Software Fault Tolerance, John Wiley & Sons, New York, NY, USA, 1995.

[33] L. Chen and A. Avizienis, "N-version programming: a fault-tolerance approach to reliability of software operation," in *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*, pp. 3–9, 1995.

[34] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Software Testing Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[35] MuJava Version 3, June 2009, http://cs.gmu.edu/∼offutt/mujava/.

[36] M. M. Mano, *Digital Design*, Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2002.

[37] L. Copeland, *A Practitioner's Guide to Software Test Design*, STQE Publishing, Norwood, Mass, USA, 2004.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 428521, 18 pages
doi:10.1155/2010/428521

Research Article

Automated Test Case Prioritization with Reactive GRASP

Camila Loiola Brito Maia, Rafael Augusto Ferreira do Carmo, Fabrício Gomes de Freitas, Gustavo Augusto Lima de Campos, and Jerffeson Teixeira de Souza

Optimization in Software Engineering Group (GOES.UECE), Natural and Intelligent Computing Lab (LACONI), State University of Ceará (UECE), Avenue Paranjana 1700, Fortaleza, 60740-903 Ceará, Brazil

Correspondence should be addressed to Camila Loiola Brito Maia, camila.maia@gmail.com

Received 15 June 2009; Revised 17 September 2009; Accepted 14 October 2009

Academic Editor: Phillip Laplante

Modifications in software can affect some functionality that had been working until that point. In order to detect such a problem, the ideal solution would be testing the whole system once again, but there may be insufficient time or resources for this approach. An alternative solution is to order the test cases so that the most beneficial tests are executed first, in such a way only a subset of the test cases can be executed with little lost of effectiveness. Such a technique is known as regression test case prioritization. In this paper, we propose the use of the Reactive GRASP metaheuristic to prioritize test cases. We also compare this metaheuristic with other search-based algorithms previously described in literature. Five programs were used in the experiments. The experimental results demonstrated good coverage performance with some time overhead for the proposed technique. It also demonstrated a high stability of the results generated by the proposed approach.

1. Introduction

More than often, when a system is modified, the modifications may affect some functionality that had been working until that point in time. Due to the unpredictability of the effects that such modifications may cause to the system's functionalities, it is recommended to test the system, as a whole or partially, once again every time a modification takes place. This is commonly known as regression testing. Its purpose is to guarantee that the software modifications have not affected the functions that were working previously.

A test case is a set of tests performed in a sequence and related to a test objective [1], and a test suite is a set of test cases that will execute sequentially. There are basically two ways to perform regression tests. The first one is by reexecuting all test cases in order to test the entire system once again. Unfortunately, and usually, there may not be sufficient resources to allow the reexecution of all test cases every time a modification is introduced. Another way to perform regression test is to order the test cases in respect to their beneficial factor to some attribute, such as coverage, and reexecute the test cases according to that ordering. In doing this, the most beneficial test cases would be executed first, in such a way only a subset of the test cases can be executed with little lost of effectiveness. Such a technique is known as regression test case prioritization. When the time required to reexecute an entire test suite is sufficiently long, test case prioritization may be beneficial because meeting testing goals earlier can yield meaningful benefits [2].

According to Myers [3], since exhaustive testing is out of question, the objective should be to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases. As Fewster stated in [1], software testing needs to be effective at finding any defects which are there, but it should also be efficient by performing the tests as quickly and cheaply as possible.

The regression test case prioritization problem is closely related to the regression test case selection problem. The Regression Test Case Selection problem can be directly modeled as a set covering problem, which is a well-known NP-Hard problem [4]. This fact points to the complexity of the Test Case Prioritization problem.

To order the test cases, it is necessary to consider a base comparison measure. A straightforward measure to evaluate a test case would be based on APFD (Average of the Percentage of Faults Detected). Higher APFD numbers mean faster fault detection rates [5]. However, it is not possible to know the faults exposed by a test case in advance, so this value cannot be estimated before testing has taken place. Therefore, the research on test case prioritization concentrates on coverage measures. The following coverage criteria have been commonly used, APBC (Average Percentage Block Coverage), which measures the rate at which a prioritized test suite covers the blocks of the code, APDC (Average Percentage Decision Coverage), which measures the rate at which a prioritized test suite covers the decision statements in the code, and APSC (Average Percentage Statement Coverage), which measures the rate at which a prioritized test suite covers the statements. In this work, these three coverage measures will be considered.

As an example, consider a test suite T containing n test cases that covers a set B of m blocks. Let TB_i be the first test case in the order T' of T that covers block i. The APBC for ordering T' is given by the following equation (equivalent for the APDC and APSC metrics) [6]:

$$\text{APBC} = 1 - \frac{TB_1 + TB_2 + \cdots + TB_m}{nm} + \frac{1}{2n}. \quad (1)$$

Greedy algorithms have been employed in many researches regarding test case prioritization, in order to find an optimal ordering [2]. Such Greedy algorithms perform by iteratively adding a single test case to a partially constructed test suite if this test case covers, as much as possible, some piece of code not covered yet. Despite the wide use, as pointed out by Rothermel [2] and Li et al. [6], Greedy algorithms may not choose the optimal test case ordering. This fact justifies the application of global approaches, that is, approaches which consider the evaluation of the ordering as a whole, not individually to each test case. In that context, metaheuristics have become the focus in this field. In this work, we have tested Reactive GRASP, not yet used for test case prioritization.

Metaheuristic search techniques are algorithms that may find optimal or near optimal solutions to optimization problems [7]. In the context of software engineering, a new research field has emerged by the application of search techniques, especially metaheuristics, to well-known complex software engineering problems. This new field has been named SBSE (Search-Based Software Engineering). In this field, the software engineering problems are modeled as optimization problems, with the definitions of an objective function—or a set of functions—and a set of constraints. The solutions to the problems are found by the application of search techniques.

The application of genetic algorithms, an evolutionary metaheuristic, has been shown to be effective for regression test case prioritization [8, 9]. We examine in this paper the application of another well-known metaheuristic, GRASP, not applied yet neither to the regression test case selection problem nor to any other search-based software engineering problem. The GRASP metaheuristic was considered due to its good performance reported by several studies in solving complex optimization problems.

The remaining of this paper is organized as follows: Section 2 describes works related to the regression test case prioritization problem and introduces some algorithms which have been applied to this problem. These algorithms will be employed in the evaluation of our approach later on the paper. Section 3 describes the GRASP metaheuristic and the proposed algorithm using Reactive GRASP. Section 4 presents the details of the experiments, and Section 5 reports the conclusions of this research and states future works.

2. Related Work

This section reports the use of search-based prioritization approaches and metaheuristics. Some algorithms implemented in [6] by Li et al. which will have their performance compared to that of the approach proposed later on this paper will also be described.

2.1. Search-Based Prioritization Approaches.

The works below employed search-based prioritization approaches, such as greedy- and metaheuristic-based solutions.

Elbaum et al. [10] analyze several prioritization techniques and provide responses to which technique is more suitable for specific test scenarios and their conditions. The metric APFD is calculated through a greedy heuristic. Rothermel et al. [2] describe a technique that incorporates a Greedy algorithm called Optimal Prioritization, which considers the known faults of the program, and the test cases are ordered using the fault detection rates. Walcott et al. [8] propose a test case prioritization technique with a genetic algorithm which reorders test suites based on testing time constraints and code coverage. This technique significantly outperformed other prioritization techniques described in the paper, improving in, on average, 120% the APFD over the others.

Yoo and Harman [9] describe a Pareto approach to prioritize test case suites based on multiple objectives, such as code coverage, execution cost, and fault-detection history. The objective is to find an array of decision variables (test case ordering) that maximize an array of objective functions. Three algorithms were compared: a reformulation of a Greedy algorithm (Additional Greedy algorithm), Non-Dominating Sorting Genetic Algorithm (NSGA-II) [11], and a variant of NSGA-II, vNSGA-II. For two objective functions, a genetic algorithm outperformed the Additional Greedy algorithm, but for some programs the Additional Greedy algorithm produced the best results. For three objective functions, Additional Greedy algorithm had reasonable performance.

Li et al. [6] compare five algorithms: Greedy algorithm, which adds test cases that achieve the maximum value for the coverage criteria, Additional Greedy algorithm, which adds test cases that achieve the maximum coverage not already consumed by a partial solution, 2-Optimal algorithm, which selects two test cases that consume the maximum coverage together, Hill Climbing, which performs local search in a defined neighborhood, and genetic algorithm, which generates new test cases based on previous ones. The authors separated test suites in 1,000 small suites of size 8-155 and 1,000 large suites of size 228-4,350. Six C programs were used in the experience, ranging from 374 to 11,148 LoC (lines of code). The coverage metrics studied in that work were APBC, APDC, and APSC, as described earlier. For each program, the block, decision, and statement coverage data were found by tailor-made version of a commercial tool, Cantata++. The coverage data were obtained over 500 executions for each search algorithm, using a different suite for each execution. For small programs, the performance was almost identical for all algorithms and coverage criteria, considering both small and large test suites. The Greedy algorithm performed the worst and the genetic algorithm and Additional Greedy algorithm produced the best results.

2.2. Algorithms. This section describes some algorithms which have been used frequently in literature to deal with the test case prioritization problem. The performance of them will be compared to that of the approach proposed later on this paper.

2.2.1. Greedy Algorithm. The Greedy Algorithm performs in the following way: all candidate test cases are ordered by their coverage. Then, the test case with the highest percentage of coverage is then added to an initially empty solution. Next, the test case with the second highest percentage is added, and so on, until all test cases have been added.

For example, let APBC be the coverage criterion, and let a partial solution contain two test cases that cover 100 blocks of code. Suppose there are two other test cases that can be added to the solution. The first one covers 80 blocks, but 50 of these were already covered by the current solution. Then, this solution covers 80% of the blocks, but the actual added coverage of this test case is of 30% of coverage (30 blocks). The second test case covers 40 blocks of code, but none of these blocks was covered by the current solution. This means that this solution covers 40% of the blocks. The Greedy algorithm would select the first test case, because it has greater percentage of block coverage overall.

2.2.2. Additional Greedy Algorithm. The Additional Greedy algorithm adds a locally optimal test case to a partial test suite. Starting from an empty solution, the algorithm follows these steps: for each iteration, the algorithm adds the test case which gives the major coverage gain to the partial solution.

Let us use the same example from Section 2.2.1. Let a partial solution contain two test cases that cover 100 blocks of code. There are two remaining test cases: the first one covers 80 blocks, but 50 of these were already covered; the second one covers 40 blocks of code, none of these already covered. The first solution represents an actual 30% of coverage and the second one represents 40% of coverage. The Additional Greedy algorithm would select the second test case, because that solution has greater coverage factor related to the current partial solution.

2.2.3. Genetic Algorithm. Genetic algorithm is a type of Evolutionary Algorithm which has been employed extensively to solve optimization problems [12]. In this algorithm, an initial population of solutions—in our case a set of test suites—is randomly generated. The procedure then works, until a stopping criterion is reached, as new populations are generated based on the previous one [13]. The evolution from one population to the next one is performed via "genetic operators", including operations of selection, that is, the biased choice of which individuals of the current population will reproduce to generate individuals for the new population. This selection prioritizes individuals with high fitness value, which represents how good this solution is. The other two genetic operators are crossover, that is, the combination of individuals to produce the offspring, and mutation, which randomly changes a particular individual.

In the genetic algorithm proposed by Li et al. [6], the initial population is produced by selecting test cases randomly from the test case pool. The fitness function is based on the test case position in the current test suite. The fitness value was calculated as follows:

$$\text{fitness}(pos) = 2 \cdot \frac{(pos - 1)}{(n - 1)}, \tag{2}$$

where pos is the test case's position in the current test suite and n is the population size.

The crossover algorithm follows the ordering chromosome crossover style adopted by Antoniol [14] and used in [6] by Li et al. for the genetic algorithm in the experiments. It works as follows. Let p_1 and p_2 be the parents, and let o_1 and o_2 be the offspring. A random position k is selected, and the first k elements of p_1 become the first k elements of o_1, and the last $n - k$ elements of o_1 are the $n - k$ elements of p_2 which remain when the k elements selected from p_1 are removed from p_2. In the same way, the first k elements of p_2 become the first k elements of o_2, and the last $n - k$ elements of o_2 are the $n-k$ elements of p_1 which remain when the k elements selected from p_2 are removed from p_1. The mutation is performed by randomly exchanging the position of two test cases.

2.2.4. Simulated Annealing. Simulated annealing is a generalization of a Monte Carlo method. Its name comes from annealing in metallurgy, where a melt, initially disordered at high temperature, is slowly cooled, with the purpose of obtaining a more organized system (a local optimum solution). The system approaches a frozen ground state with $T = 0$. Each step of simulated annealing algorithm replaces the current solution by a random solution in its neighborhood, based on a probability that depends on the energies of the two solutions.

3. Reactive GRASP for Test Case Prioritization

This section is intended to present a novel approach for test case prioritization based on the Reactive GRASP metaheuristic.

3.1. The Reactive GRASP Metaheuristic. Metaheuristics are general search algorithms that find a good solution, sometimes optimal, to **optimization** problems. In this section we present, in a general fashion, the metaheuristic which will be

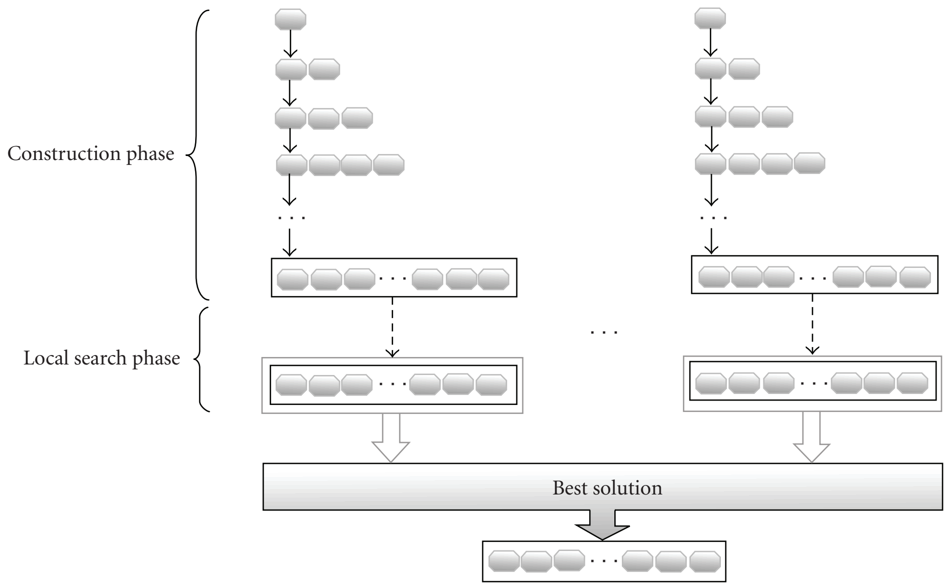


Figure 1: GRASP's phases.

employed to prioritize test cases by the approach proposed later on this paper.

GRASP (Greedy Randomized Adaptative Search Procedures) is a metaheuristic with two phases: construction and local search [15]. This metaheuristic is defined as a multistart algorithm, since the procedure is executed multiple times in order to get the best solution found overall; see Figure 1.

In the construction phase, a feasible solution is built by applying some Greedy algorithm. The greedy strategy used in GRASP is to add to an initially empty solution one element at a time. This algorithm tends to find a local optimum. Therefore, in order to avoid this local best, GRASP uses a randomization greedy strategy as follows. The Restrict Candidate List (RCL) stores the possible elements which can be added at each step in this construction phase. The element to be added is picked randomly from this list. RCL is associated with a parameter named α, which limits the length of the RCL. If $\alpha = 0$, only the best element—with highest coverage—will be present in the RCL, making the construction process a pure Greedy algorithm. Otherwise, if $\alpha = 1$, the construction phase will be completely random, because all possible elements will be in RCL. The parameter α should be set to calibrate how random and greedy the construction process will be. The found solution is then used in the local search phase.

In the local search phase, the aim is to find the best solution in the current solution neighborhood. Indeed, a local search is executed in order to replace the current solution by the local optimum in its neighborhood. After this process, this local optimum is compared with the best local optimum solution found in earlier iterations. If the local optimum just found is better, then this is set to be the best solution already found. Otherwise, there is no replacement.

As can be easily seen, the performance of the GRASP algorithm will strongly depend on the choice of the parameter α. In order to decrease this influence, a GRASP variation named Reactive GRASP [15, 16] has been proposed. This approach performs GRASP while varying the values of α according to their previous performance. In practice, Reactive GRASP will initially determine a set of possible values for α. Each value will have a probability of being selected in each iteration.

Initially, all α probabilities are assigned to $1/n$, where n is the quantity of α. For each one of the i values of α, the probabilities p_i are reevaluated for each iteration, according to the following equation:

$$p_i = \frac{q_i}{\sum_{j=1}^{n} qj}, \tag{3}$$

with $q_i = S^*/A_i$, where S^* is the incumbent solution and A_i is the average value of all solutions found with $\alpha = \alpha_i$. This way, when a particular α generates a good solution, its probability, given by p_i, of being selected in the future is increased. On the other hand, if a bad solution is created, the α value used in the process will have its selection probability decreased.

3.2. The Reactive GRASP Algorithm. The pseudocode below, in Algorithm 1, describes the Reactive GRASP algorithm.

The first step initializes the probabilities associated with the choice of each α (line 1).

Initially, all probabilities are assigned to $1/n$, where n is the length of α Set, the set of α values. Next, the GRASP algorithm runs the construction and local search phases, as described next, until the stopping criterion is reached. For each iteration, the best solution is updated when the new solution is better.

For each iteration, α is selected as follows; see Algorithm 2. Let S^* be the incumbent solution, and let A_i be the coverage average value of all solutions found with $\alpha = \alpha_i$, where $i = 1, \ldots, m$, and m is the number of test cases. As

```
(1) initialize probabilities associated
    with $\alpha$ (all equal to $\frac{1}{n}$)
(2) for $k = 1$ to *max_iterations* do
(3)     $\alpha \leftarrow$ select_$\alpha$ ($\alpha Set$);
(4)     *solution* $\leftarrow$ run_construction_phase($\alpha$);
(5)     *solution* $\leftarrow$ run_local_search_phase(*solution*);
(6)     update_solution(*solution*, *best_solution*);
(7) end;
(8) return *best_solution*;
```

Algorithm 1: Reactive GRASP for Test Case Prioritization.

```
procedure select_$\alpha$($\alpha Set$)
(1) $\alpha \leftarrow \alpha$ with probability $p_i = \frac{q_i}{\sum_{j=1}^{m} qj}$
(2) return $\alpha$
```

Algorithm 2: Selection of α.

described in Section 3.1, the probabilities p_i are reevaluated at each iteration by taking

$$p_i = \frac{q_i}{\sum_{j=1}^{m} qj}. \quad (4)$$

The pseudocode in Algorithm 3 details the construction phase. For each iteration, one test case which increases the coverage of the current solution (set of test cases) is selected by a greedy evaluation function. This element is randomly selected from the RCL (Restricted Candidate List), which has the best elements, that is, the best coverage values. After the element is incorporated to the partial solution, the RCL is updated. The increment of coverage is then reevaluated.

The α Set is updated after the solution is found, in order to change the selection probabilities of the α Set elements. This update is detailed in Algorithm 4.

After the construction phase, a local search phase is executed in order to improve the generated solution. This phase is important to avoid the problems mentioned by Rothermel [2] and Li et al. [6], where Greedy algorithms may fail to choose the optimal test case ordering. The pseudocode for the local search is described in Algorithm 5.

Let *s* be the test suite generated by the construction phase. The local search is performed as follows: the first test case on the test suite is exchanged with the other test cases, one at a time, that is, $n - 1$ new test suites are generated, exchanging the first test case with the *i*th one, where *i* varies from 2 to *n*, and *n* is the length of the original test suite. The original test suite is then compared with all generated test suites. If one of those test suites is better—in terms of coverage—than the original one, it replaces the original solution. This strategy was chosen because, even with very little computational effort, any exchange with the first test case can generate a very significant difference in coverage. In addition, it would be prohibitive to test all possible exchanges, since it would generate n^2 new test suites, instead of $n - 1$, in which most of

```
(1) *solution* $\leftarrow \emptyset$;
(2) initialize the candidate set $C$ with random
        test cases from the pool of test cases;
(3) evaluate the coverage $c'(e)$ for all $e \in C$;
(4) while $C \neq \emptyset$ do
(5)     $c^{\min} = \min\{c'(e) \mid e \in C\}$;
(6)     $c^{\max} = \max\{c'(e) \mid e \in C\}$;
(7)     RCL $= \{e \in C \mid c'(e) \leq c^{\min}$
                $+ \alpha(c^{\max} - c^{\min})\}$;
(8)     $s \leftarrow$ test case from the RCL at random;
(9)     *solution* $\leftarrow$ *solution* $\cup \{s\}$;
(10)    update $C$;
(11)    reevaluate $c'(e)$ for all $e \in C$;
(12) end;
(13) update_$\alpha Set$(*solution*);
(14) return *solution*;
```

Algorithm 3: Reactive GRASP for Test Case Prioritization, Construction Phase.

```
procedure update_$\alpha Set$(*solution*)
(1) update probabilities of all $\alpha$ in $\alpha Set$, using
        $p_i = \frac{q_i}{\sum_{j=1}^{m} qj}$
```

Algorithm 4: Update of α.

```
(1) while $s$ not locally optimal do
(2)     Find $s' \in Neighbour(s)$ with $f(s') < f(s)$;
(3)     $s \leftarrow s'$;
(4) end;
(5) return $s$;
```

Algorithm 5: Reactive GRASP for Test Case Prioritization, Local Search Phase.

them would exchange the last elements, with no significant difference in coverage.

4. Empirical Evaluation

In order to evaluate the performance of the proposed approach, a series of empirical tests was executed. More specifically, the experiments were designed to answer the following question.

(1) How does the Reactive GRASP approach compare—in terms of coverage and time performances—to other search-based algorithms, including Greedy algorithm, Additional Greedy algorithm, genetic algorithm, and Simulated Annealing?

In addition to this result, the experiments can confirm results previously described in literature, including the performance of the Greedy algorithm.

Table 1: Programs used in the Evaluation.

Program	LoC	Blocks	Decisions	Test Pool Size
Print_tokens	726	126	123	4,130
Print_tokens2	570	103	154	4,115
Schedule	412	46	56	2,650
Schedule2	374	53	74	2,710
Space	9,564	869	1,068	13,585

4.1. Experimental Design. Four small programs (print_tokens, print_tokens2, schedule, and schedule2) and a larger program (space) were used in the tests. These programs were assembled by researchers at Siemens Corporate Research [17] and are the same Siemens' programs used in Li et al. [6] for the experiments regarding test case prioritization. Table 1 describes the five programs' characteristics.

Besides Reactive GRASP, other search algorithms have also been implemented, in order to compare their effectiveness. They are Greedy algorithm, Additional Greedy algorithm, genetic algorithm, and simulated annealing. These algorithms were implemented exactly as described in Section 3 of this paper. For the genetic algorithm, as presented by Li et al. [6], the population size was set at 50 individuals and the algorithm was terminated after 100 generations. Stochastic universal sampling was used in selection and mutation, the crossover probability (per individual) was set to 0.8, and the mutation probability was set to 0.1. For the Reactive GRASP approach, the maximum number of iterations was set, by preliminary experimentation, to 300.

For the simulated annealing approach, the initial temperature was set to a random number between 20 and 99. For each iteration, the new temperature is given by the following steps:

(1) $dividend = actualTemperature + initialTemperature$,

(2) $divisor = 1 + \log_{10} 1$,

(3) $new\ temperature = \dfrac{dividend}{divisor}$.

In the experiments, we considered the three coverage criteria described earlier (APBC, APDC, and APSBC). In addition, we considered different percentages of the pool of test cases. For example, if the percentage is 5%, 5% of test cases are randomly chosen from the pool to compare the performance of the algorithms. We tested with 1%, 2%, 3%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% for the four small programs and 1%, 5%, 10%, 20%, 30%, 40%, and 50% for space. Each algorithm was executed 10 times for the four small programs and 1 time for the space program, for each coverage criterion and each percentage.

The pools of test cases used in the experiments were collected from SEBASE [18]. The test cases used are composed of "0"s and "1"s, where "0" represents "code not covered" and "1" represents "code covered". The length of a test case is the quantity of portions of code of the program. For example, when we are analyzing the decision coverage, the length of the test cases is the quantity of decisions on the program. In the APDC, a "0" for the first decision means that the first decision is not covered by the test suite and a "1" for the second decision means that the second decision is covered by the test suite, and so on.

All experiments were performed on Ubuntu Linux workstations with kernel 2.6.22-14, a Core Duo processor, and 1 GB of main memory. The programs used in the experiment were implemented using the Java programing language.

4.2. Results. The results are presented in Tables 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18 and Figures 2 to 17, separating the four small programs from the space program. Tables 2, 3, 4, and 5 detail the average of 10 executions of the coverage percentage achieved for each coverage criterion and each algorithm for printtokens, printtokens2, schedule, and schedule2, respectively. Table 12 has this information regarding the space program. The TSSp column is the percentage of test cases selected from the test case pool. The mean differences on time execution in seconds are also presented in Tables 6 and 16, for small programs and space, respectively.

Tables 7 and 14 show the weighted average for the metrics (APBC, APDC, and APSC) for each algorithm. Figures 2 to 17 demonstrate a comparison among the algorithms for the metrics APBC, APDC, and APSC, for the small programs and space program.

4.3. Analysis. Analyzing the results obtained from the experiments, which are detailed in Tables 2, 3, 4, 5, and 9 and summarized in Tables 6 and 13, several relevant results can be pointed out. First, the Additional Greedy algorithm had the best performance in effectiveness of all tests. It performed significantly better than the Greedy algorithm, the genetic algorithm, and simulated annealing, both for the four small programs and for the space program. The good performance of the Additional Greedy algorithm had already been demonstrated in several works, including Li et al. [6] and Yoo and Harman [9].

4.3.1. Analysis for the Four Small Programs. The Reactive GRASP algorithm had the second best performance. This approach also significantly outperformed the Greedy algorithm, the genetic algorithm, and simulated annealing, considering the coverage results. When compared to the Additional Greedy algorithm, there were no significant differences in terms of coverage. Comparing the metaheuristic-based approaches, the better performance obtained by the Reactive GRASP algorithm over genetic algorithm and simulated annealing was clear.

In 168 experiments, the genetic algorithm generated a better coverage only once (block criterion, the schedule program, and 100% of tests being considered). The two algorithms tied also once. For all other tests, the Reactive GRASP outperformed the genetic algorithm. The genetic algorithm approach performed the fourth best in our evaluation. In Li et al. [6], the genetic algorithm was also worse than the Additional Greedy algorithm. The results

Table 2: Results of Coverage Criteria (Average of 10 Executions), Program Print-tokens.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.6591	98.235	96.6893	96.1242	97.808
2%	98.3209	99.2101	98.3954	98.3113	99.0552
3%	98.6763	99.5519	98.5483	98.5553	99.3612
5%	98.5054	99.6909	98.8988	98.9896	99.5046
10%	98.2116	99.8527	99.2378	99.3898	99.7659
20%	98.266	99.9317	99.2378	99.6414	99.8793
30%	98.3855	99.9568	99.6603	99.6879	99.9204
40%	98.3948	99.9675	99.7829	99.736	99.9457
50%	98.4064	99.9747	99.8321	99.8213	99.9627
60%	98.4097	99.979	99.8666	99.8473	99.9622
70%	98.4133	99.9818	99.8538	99.8698	99.9724
80%	98.4145	99.9841	99.8803	99.8657	99.9768
90%	98.4169	99.9859	99.9013	99.8958	99.9783
100%	98.418	99.9873	99.9001	99.8895	99.9775
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.7692	98.3836	96.9125	95.9213	98.1204
2%	98.0184	99.1429	97.9792	98.2299	98.8529
3%	98.5569	99.4499	98.3785	98.0762	99.2886
5%	98.4898	99.6971	98.7105	98.7513	99.4631
10%	98.1375	99.8462	98.8659	99.1759	99.697
20%	98.2486	99.928	99.3886	99.5111	99.8668
30%	98.3131	99.952	99.587	99.6955	99.9061
40%	98.3388	98.3388	99.7137	99.7505	99.9237
50%	98.3437	99.9712	99.7305	99.78	99.9386
60%	98.358	99.9766	99.817	99.8235	99.959
70%	98.3633	99.9799	99.8109	99.7979	99.9543
80%	98.3651	99.9821	99.8631	99.8447	99.9663
90%	98.4169	99.9859	99.9013	99.8541	99.9783
100%	98.418	99.9873	99.9001	99.869	99.9775
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.2989	98.3561	97.0141	97.251	98.0439
2%	97.7834	99.2557	98.0175	98.576	98.9675
3%	98.0255	99.4632	98.5163	98.5633	99.2356
5%	97.8912	99.6826	98.5167	99.0268	99.4431
10%	97.8137	99.8534	99.1497	99.3131	99.681
20%	98.0009	99.9264	99.5024	99.5551	99.8554
30%	98.0551	99.954	99.6815	99.7151	99.9079
40%	98.0661	99.9656	99.7342	99.7677	99.9296
50%	98.0705	99.9724	99.8123	99.8108	99.9464
60%	98.0756	99.9773	99.8348	99.8456	99.9598
70%	98.0887	99.9805	99.8641	99.8633	99.9704
80%	98.088	99.9831	99.89	99.8649	99.9682
90%	98.0924	99.985	99.9026	99.8819	99.9709
100%	98.0943	99.9865	99.8998	99.8897	99.977

Table 3: Results of Coverage Criteria (Average of 10 Executions), Program Print-tokens2.

Block Coverage%					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.233	98.3518	97.6629	98.042	98.1576
2%	98.3869	99.2665	98.6723	98.8302	99.208
3%	97.9525	99.5122	98.8576	99.1817	99.3274
5%	98.1407	99.711	99.2379	99.3382	99.5932
10%	98.131	99.8564	99.5558	99.6731	99.7994
20%	98.01	99.9293	99.7894	99.8015	99.8689
30%	98.0309	99.9535	99.8269	99.839	99.9239
40%	98.0462	99.9656	99.8602	99.8957	99.9495
50%	98.0569	99.9727	99.9166	99.9106	99.9653
60%	98.0589	99.977	99.9165	99.9269	99.9689
70%	98.0611	99.9805	99.9264	99.9236	99.9756
80%	98.0632	99.9828	99.9383	99.9261	99.9778
90%	98.0663	99.9849	99.9543	99.9385	99.9796
100%	98.0671	99.9864	99.9562	99.9434	99.9811
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.05	98.3055	97.2108	97.6375	98.0859
2%	98.6637	99.2489	98.4368	98.5244	99.0987
3%	98.5798	99.5496	98.8814	99.0411	99.4344
5%	98.5903	99.7371	99.3676	99.2289	99.6772
10%	98.5673	99.8628	99.5183	99.6528	99.8118
20%	98.6351	99.9353	99.7939	99.8084	99.913
30%	98.6747	99.9593	99.8615	99.8405	99.9482
40%	98.6837	99.9692	99.8836	99.8802	99.9556
50%	96.1134	99.9552	99.8269	99.8992	99.9318
60%	98.6948	99.9795	99.9181	99.9109	99.9751
70%	98.6964	99.9826	99.9358	99.9302	99.9768
80%	98.6985	99.9848	99.9478	99.931	99.9788
90%	98.0663	99.9849	99.9543	99.9409	99.9796
100%	98.0671	99.9864	99.9562	99.9424	99.9811
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.4458	98.3742	97.8234	97.453	98.2804
2%	98.7611	99.2552	98.755	98.5444	99.0653
3%	98.3634	99.4745	98.9385	98.9165	99.3279
5%	97.8694	99.6856	99.1507	99.3858	99.5327
10%	98.0271	99.8494	99.5268	99.6258	99.7906
20%	98.1264	99.927	99.7455	99.7283	99.9086
30%	97.9467	99.9518	99.8533	99.8297	99.9328
40%	97.9653	99.9645	99.8833	99.864	99.9564
50%	97.9762	99.9717	99.9126	99.8891	99.9584
60%	97.9792	99.9768	99.9162	99.905	99.9644
70%	97.9851	99.9803	99.9265	99.9156	99.9708
80%	97.9854	99.9827	99.9399	99.9187	99.9759
90%	97.9877	99.9847	99.9399	99.9288	99.9789
100%	97.9894	99.9863	99.9477	99.9262	99.9791

Table 4: Results of Coverage Criteria (Average of 10 Executions), Program Schedule.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.6505	98.2286	98.2286	97.9387	98.2286
2%	96.5053	99.0237	98.9499	98.8596	99.0073
3%	96.451	99.3315	99.2336	99.1955	99.2445
5%	95.6489	99.5652	99.2481	99.4066	99.3233
10%	95.2551	99.767	99.5586	99.6455	99.7013
20%	95.9548	99.8884	99.7604	99.7497	99.8589
30%	95.8225	99.9224	99.8219	99.8442	99.8918
40%	96.0783	99.9429	99.8995	99.8982	99.9163
50%	96.3159	99.9553	99.9051	99.899	99.9396
60%	96.9283	99.9644	99.918	99.9156	99.9546
70%	97.0744	99.9695	99.9235	99.9322	99.9643
80%	97.0955	99.9733	99.9464	99.9411	99.9649
90%	97.1171	99.9763	99.9474	99.946	99.9704
100%	97.0495	99.9786	99.9573	99.9454	99.7013
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.3492	98.2671	98.0952	98.0092	98.2407
2%	95.9838	99.0566	98.7129	98.7364	98.9218
3%	95.933	99.3303	98.9107	98.9575	99.2589
5%	95.1047	99.5327	98.6224	99.0856	99.2427
10%	94.8611	99.7668	99.2237	99.3422	99.6159
20%	94.75	99.8739	99.4858	99.6266	99.7749
30%	95.3616	99.9241	99.7047	99.7181	99.8757
40%	95.3396	99.9413	99.7944	99.7871	99.9144
50%	96.1134	99.9552	99.8269	99.8515	99.9318
60%	96.3241	99.9627	99.852	99.8541	99.9416
70%	96.5465	99.968	99.8673	99.8927	99.9586
80%	96.9312	99.9722	99.88	99.8868	99.9553
90%	97.1171	99.9763	99.9474	99.9077	99.9704
100%	97.0495	99.9786	99.9573	99.9118	99.9701
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.747	98.1768	98.0792	98.1911	98.1596
2%	97.0323	99.039	98.8108	98.8664	99.0273
3%	96.937	99.3284	99.1366	99.1927	99.2257
5%	96.3181	99.5751	99.2731	99.4252	99.4398
10%	96.1091	99.782	99.452	99.6635	99.6428
20%	96.9909	99.8945	99.7965	99.8168	99.8693
30%	97.2931	99.9307	99.8703	99.8683	99.9112
40%	97.0724	99.9471	99.9003	99.8983	99.9358
50%	97.4288	99.9584	99.9214	99.9146	99.9445
60%	97.4015	99.9653	99.932	99.9281	99.9594
70%	97.6458	99.9707	99.9374	99.931	99.9653
80%	97.8832	99.9748	99.9399	99.9273	99.9722
90%	97.8907	99.9777	99.9496	99.9471	99.9653
100%	97.8901	99.9799	99.9627	99.9494	99.978

Table 5: Results of Coverage Criteria (Average of 10 Executions), Program Schedule2.

Block Coverage %

TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.2708	98.1199	98.066	98.167	98.1064
2%	98.2538	99.0566	98.9605	99.0325	99.036
3%	98.6447	99.3764	99.3304	99.3464	99.3534
5%	99.4678	99.6184	99.5851	99.5879	99.6184
10%	98.2116	99.8527	99.2378	99.7869	99.7659
20%	99.9056	99.907	99.8952	99.893	99.907
30%	99.9385	99.9385	99.9348	99.9267	99.9385
40%	99.9538	99.9538	99.9476	99.9418	99.9538
50%	99.963	99.963	99.9586	99.9535	99.963
60%	99.9692	99.9692	99.9676	99.9612	99.9692
70%	99.9736	99.9736	99.9702	99.9584	99.9736
80%	99.9769	99.9769	99.972	99.9641	99.9769
90%	99.9794	99.9794	99.9779	99.9735	99.9794
100%	99.9815	99.9815	99.9796	99.9701	99.9815

Decision Coverage %

TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	95.6563	98.3687	97.9922	98.1129	98.3301
2%	96.1375	98.9533	98.2113	98.5404	98.8501
3%	95.5965	99.3111	98.4344	98.9122	99.0936
5%	97.6887	99.6164	99.058	99.2189	99.4773
10%	97.1277	99.7985	99.4385	99.4873	99.7057
20%	97.2249	99.9027	99.7033	99.7575	99.8713
30%	97.2647	99.9352	99.8177	99.8224	99.9126
40%	97.2726	99.9513	99.8145	99.8673	99.9144
50%	97.2823	99.9712	99.8745	99.8907	99.9411
60%	97.2869	99.9676	99.8827	99.9143	99.9584
70%	97.2981	99.9722	99.915	99.9013	99.9595
80%	97.3005	99.9756	99.9311	99.915	99.9695
90%	99.9794	99.9794	99.9779	99.9304	99.9794
100%	99.9815	99.9815	99.9796	99.9297	99.9815

Statement Coverage %

TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.7116	98.2883	98.1984	98.0316	98.2777
2%	97.4612	99.1097	98.9346	98.6235	99.0208
3%	97.1499	99.336	98.9259	99.0397	99.1481
5%	97.7227	99.6029	99.3066	99.428	99.5114
10%	98.3422	99.8072	99.6104	99.6295	99.734
20%	98.4317	99.9014	99.7765	99.7866	99.8815
30%	98.474	99.9363	99.8543	99.8455	99.9074
40%	98.4861	99.9525	99.8892	99.8833	99.9441
50%	98.4988	99.962	99.9159	99.9055	99.9568
60%	98.5041	99.9683	99.9251	99.9123	99.9626
70%	98.5109	99.9728	99.9345	99.9278	99.9663
80%	98.512	99.9762	99.9429	99.9291	99.9725
90%	98.5166	99.9788	99.9549	99.9463	99.9757
100%	98.521	99.981	99.9583	99.9453	99.9783

TABLE 6: Coverage Significance and Time Mean Difference, Small Programs.

Algorithm (x)	Algorithm (y)	Mean Coverage Difference (%) ($x - y$)	Coverage Difference Significance (t-test)	Time Mean Difference (s) ($x - y$)
Greedy Algorithm	Additional Greedy Algorithm	−1.9041	0.0000	−1.4908
	Genetic Algorithm	−1.8223	0.0000	−8.4361
	Simulated Annealing	−1.8250	0.0000	−0.0634
	Reactive GRASP	−1.8938	0.0000	−100.0312
Additional Greedy Algorithm	Greedy Algorithm	1.9041	0.0000	1.4908
	Genetic Algorithm	0.0818	0.0000	−6.9452
	Simulated Annealing	0.0790	0.0000	1.4274
	Reactive GRASP	0.0103	0.1876	−98.5403
Genetic Algorithm	Greedy Algorithm	1.8223	0.0000	8.4361
	Additional Greedy Algorithm	−0.0818	0.0000	6.9452
	Simulated Annealing	−0.0026	0.4918	8.3727
	Reactive GRASP	−0.0715	0.0000	−91.5951
Simulated Annealing	Greedy Algorithm	1.8250	0.0000	0.0634
	Additional Greedy Algorithm	−0.0790	0.0000	−1.4274
	Genetic Algorithm	0.0026	0.4918	−8.3727
	Reactive GRASP	−0.0688	0.0000	−99.9679
Reactive GRASP	Greedy Algorithm	1.8938	0.0000	100.0312
	Additional Greedy Algorithm	−0.0103	0.1876	98.5403
	Genetic Algorithm	0.0715	0.0000	91.5951
	Simulated Annealing	0.0688	0.0000	99.9679

TABLE 7: Weighted Average for the Metrics, Small Programs.

Coverage Criterion	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Block Coverage	98.2858	**99.9578**	99.8825	99.8863	99.9335
Decision Coverage	97.8119	99.9276	99.8406	99.8417	**99.9368**
Statement Coverage	98.0328	**99.9573**	99.8743	99.8706	99.9417

TABLE 8: Difference in Performance between the Best and Worst Criteria, Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Difference in performance between the best and worst criteria	0.4739	0.0302	0.0419	0.0446	0.0082

TABLE 9: Average for Each Algorithm (All Metrics), Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Final Average	98.0435	99.9476	99.8658	99.8662	99.9373

TABLE 10: Standard Deviation of the Effectiveness for the Four Algorithms, Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Standard Deviation	0.002371	0.000172	0.000222	0.000226	0.000041

TABLE 11: Summary of Results, Small Programs.

Algorithm	Coverage Performance	Execution Time	Observations
Greedy Algorithm	The worst performance	Fast	
Additional Greedy Algorithm	Best performance of all	Fast	
Genetic Algorithm	Fourth best performance	Medium	It generated a better coverage only once.
Simulated Annealing	Third best performance	Fast	No significant difference to genetic algorithm.
Reactive GRASP	Second best performance	Slow	No significant difference to Additional Greedy Algorithm.

Table 12: Results of Coverage Criteria (1 Execution), Program Space.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	87.4115	96.4804	92.6728	91.4603	95.6961
5%	85.8751	98.5599	94.8614	94.9912	98.0514
10%	85.5473	99.1579	95.9604	96.7242	98.6774
20%	86.5724	99.6063	98.0118	97.991	99.4235
30%	86.9639	99.7423	98.5998	98.6937	99.6431
40%	87.3629	99.811	98.9844	98.9004	99.7339
50%	87.8269	99.842	99.1271	99.216	99.7755
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	88.753	96.9865	91.6811	92.0529	96.4502
5%	85.5131	98.553	93.6639	94.9256	97.8443
10%	86.9345	99.1999	95.9172	96.6152	98.358
20%	87.9909	99.6074	98.0217	97.7348	99.2446
30%	88.4008	99.7464	98.4662	98.5373	99.3256
40%	88.6799	99.8074	98.9283	98.8599	99.7149
50%	88.6635	99.8476	99.0786	98.84	99.7469
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	92.8619	97.7642	94.3287	93.5957	97.0516
5%	90.9306	99.1171	95.7946	96.4218	98.4031
10%	91.3637	99.5086	97.5863	97.7154	99.3172
20%	91.7803	99.7598	98.6129	98.6336	99.6214
30%	92.1344	99.8473	99.0048	99.2151	99.6555
40%	92.1866	99.8859	99.3106	99.2963	99.8365
50%	92.2787	99.9117	99.4053	99.4852	99.8517

in the present paper demonstrate that this algorithm is also worse than the proposed Reactive GRASP approach. The simulated annealing algorithm had the third best performance, outperforming only the Greedy algorithm.

Figures 2, 3, and 4 demonstrate a comparison among the five algorithms used in the experiments. It is easy to see that the best performance was that of the Additional Greedy algorithm, followed by that of the Reactive GRASP algorithm. Reactive GRASP surpassed the genetic algorithm and simulated annealing in all coverage criteria, and it had the best performance at APDC criterion. The Additional Greedy algorithm was better at APBC and APSC criteria and Greedy algorithm was the worst of all.

For better visualization, consider Figures 5 and 6 that show these comparisons among the used algorithms. To make the result clearer, Figures 7 and 8 have this information regarding the 3 more efficient algorithms in this experiment. Figure 9 shows the final coverage average for each algorithm.

To investigate the statistical significance, we used *t*-test, which can be seen in Table 6. For each pair of algorithms, the mean coverage difference is given, and the significance level. If the significance is smaller than 0.05, the difference between the algorithms is statistically significant [6]. As can be seen, there is no significant difference between Reactive GRASP and Additional Greedy, in terms of coverage. In addition, one can see that there is no significant difference between simulated annealing and genetic algorithm, also in accordance with Table 6.

We can also notice in Table 6 the time mean difference for execution, for each pair of algorithms. It is important to mention that the time required to execute Reactive GRASP was about 61.53 larger than the time required to execution for Additional Greedy algorithm.

Another conclusion that can be drawn from the graphs is that the performance of the Reactive GRASP algorithm has remained similar for all metrics used, while Additional Greedy algorithm was a slightly different behavior for each metric.

Table 7 shows the weighted average of the algorithms, for each coverage criterion. The best results are highlighted in the table (bold). Table 8 shows the difference in performance between the best and the worst metric regarding the coverage percentage. In this experiment, Reactive GRASP had the minor difference in performance between the best and the worst coverage criterion, which demonstrates an interesting characteristic of this algorithm: its stability.

Table 9 contains the effectiveness average for all coverage criteria for each algorithm (APBC, APDC, and APSC).

Table 13: Coverage Significance and Time Mean Difference, Program Space.

Algorithm (x)	Algorithm (y)	Mean Coverage Difference (%) $(x - y)$	Coverage Difference Significance (t-test)	Time Mean Difference (s) $(x - y)$
Greedy Algorithm	Additional Greedy Algorithm	−10.5391	0.0000	−16.643
	Genetic Algorithm	−9.4036	0.0000	−495.608
	Simulated Annealing	−9.4459	0.0000	−5.339
	Reactive GRASP	−10.3639	0.0000	−36, 939.589
Additional Greedy Algorithm	Greedy Algorithm	10.5391	0.0000	16.643
	Genetic Algorithm	1.1354	0.0000	−478.965
	Simulated Annealing	1.0931	0.0000	11.303
	Reactive GRASP	0.1752	0.0613	−36, 922.945
Genetic Algorithm	Greedy Algorithm	9.4036	0.0000	495.608
	Additional Greedy Algorithm	−1.1354	0.0000	478.965
	Simulated Annealing	−0.0423	0.4418	490.268
	Reactive GRASP	−0.9602	0.0000	−36, 443.980
Simulated Annealing	Greedy Algorithm	9.4459	0.0000	5.339
	Additional Greedy Algorithm	−1.0931	0.0000	−11.303
	Genetic Algorithm	0.0423	0.4418	−490.268
	Reactive GRASP	−0.9180	0.0000	−3, 6934.249
Reactive GRASP	Greedy Algorithm	10.3639	0.0000	36, 939.589
	Additional Greedy Algorithm	−0.1752	0.0613	36,922.945
	Simulated Annealing	0.9180	0.0000	3,6934.249
	Genetic Algorithm	0.9602	0.0000	36,443.980

Table 14: Weighted Average for the Metrics, Program Space.

Coverage Criterion	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Block Coverage	87.1697	**99.6781**	98.4650	98.53273	99.5424
Decision Coverage	88.3197	**99.6856**	98.3631	98.33361	99.4221
Statement Coverage	92.0653	**99.8081**	98.9375	99.02625	99.6819

Together with Figure 9, Table 9 reinforces that the best performance was obtained by Additional Greedy algorithm, followed by that of the Reactive GRASP algorithm. Notice that Reactive GRASP algorithm has little difference in the performance compared with that of Additional Greedy algorithm.

The standard deviation shown in Table 10 refers to the 3 metrics (APBC, APDC, and APSC). It was calculated using the weighted average percentage of each algorithm. According to data in Table 10, the influence of the effectiveness performance regarding the coverage criterion is the lowest in the proposed Reactive GRASP algorithm, since its standard deviation value is the minimum among the algorithms. These data mean that the proposed technique is the one that less varies its performance related to the coverage criteria, which, again, demonstrates its higher stability.

4.3.2. Analysis for the Space Program. The results for space program were similar to results for the four small programs. The Reactive GRASP algorithm had the second best performance. Additional Greedy algorithm, genetic algorithm, simulated annealing, and Reactive GRASP algorithms significantly outperformed the Greedy algorithm. Comparing both metaheuristic-based approaches, the better performance obtained by the Reactive GRASP algorithm over the genetic algorithm and simulated annealing is clear.

The Reactive GRASP algorithm was followed by genetic algorithm approach, which performed the fourth best in our evaluation. The third best evaluation was obtained by simulated annealing.

Figures 10, 11, and 12 demonstrate a comparison between the five algorithms used in the experiments, for the space program. Based on these figures, it is possible to conclude that the best performance was that of the Additional Greedy algorithm, followed by the Reactive GRASP algorithm. Reactive GRASP surpassed the genetic algorithm, simulated annealing, and Greedy algorithm. One difference between the results for space program and the small programs is that Additional Greedy algorithm was better for all criteria, while, for small programs, Reactive GRASP had the best results for the APDC criteria. Another difference is the required execution time. As the size of the

Table 15: Difference in Performance between the Best and the Worst Criteria, Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Difference in performance between the best and worst criteria	4.8956	0.1300	0.5744	0.6926	0.2598

Table 16: Average for Each Algorithm (All Metrics), Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Final Average	89.1849	99.7240	98.5885	98.6308	99.5488

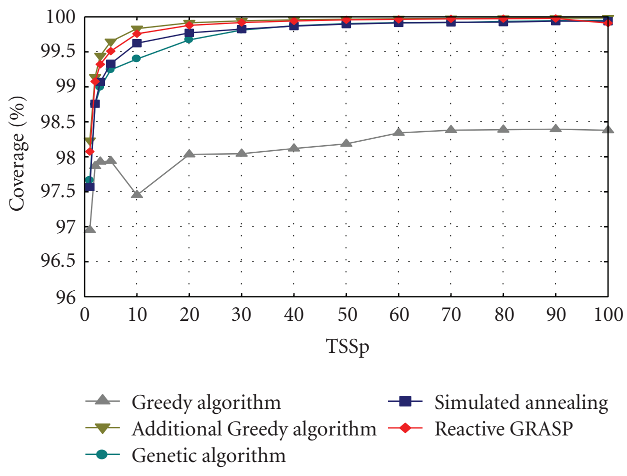


Figure 2: APBC (Average Percentage Block Coverage), Comparison among Algorithms for Small Programs.

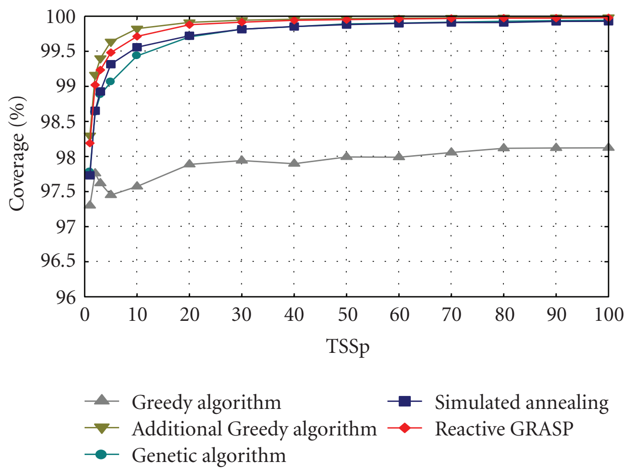


Figure 4: APSC (Average Percentage Statement Coverage), Comparison among Algorithms for Small Programs.

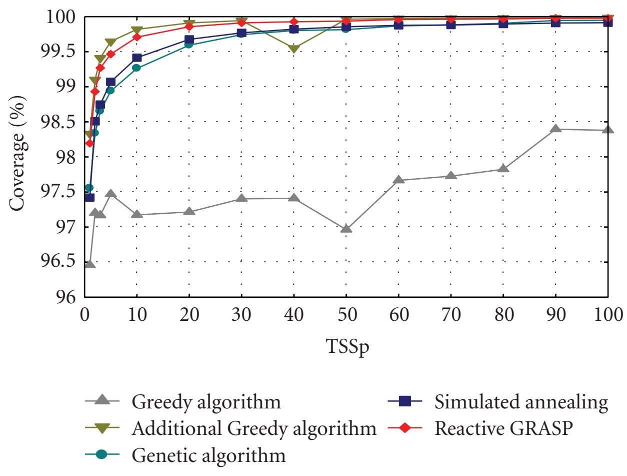


Figure 3: APDC (Average Percentage Decision Coverage), Comparison among Algorithms for Small Programs.

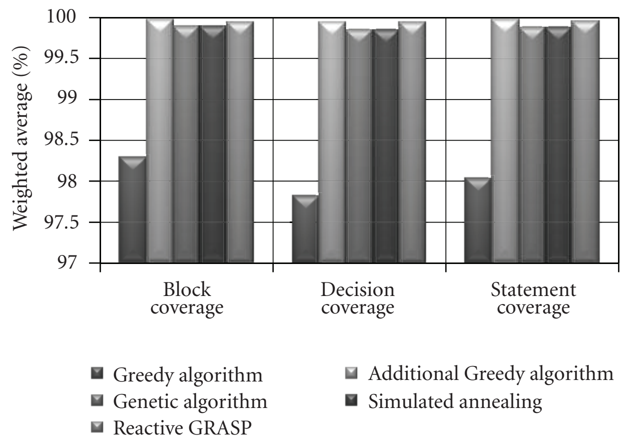


Figure 5: Weighted Average for the Metrics (Comparison among the Metrics), Small Programs.

program increases, the Reactive GRASP algorithm has its time relatively less slow compared with the others.

For better visualization, consider Figures 13 and 14 that show these comparisons among the used algorithms. To make the result clearer, Figures 15 and 16 have this information regarding the 3 more efficient algorithms in this experiment. Figure 17 shows the coverage average for each algorithm.

The t-test was used to investigate the statistical significance for space program, which can be seen in Table 13. As in the analysis for the small programs, the level of significance of the result was set to 0.05. In the same way to the small

Table 17: Standard Deviation of the Effectiveness for the Four Algorithms, Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Standard Deviation	0.025599	0.000730	0.003065	0.003566	0.001300

Table 18: Summary of Results, Program Space.

Algorithm	Coverage Performance	Execution Time	Observations
Greedy Algorithm	The worst performance.	Fast	
Additional Greedy Algorithm	Best performance of all.	Fast	
Genetic Algorithm	Fourth best performance.	Medium	
Simulated Annealing	Third best performance.	Fast	No significant difference to genetic algorithm.
Reactive GRASP	Second best performance	Slow	No significant difference to Additional Greedy Algorithm.

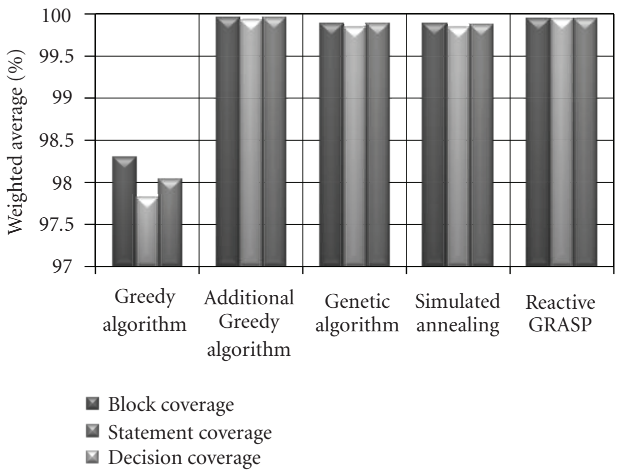


Figure 6: Weighted Average for the Metrics (Comparison among the Algorithms), Small Programs.

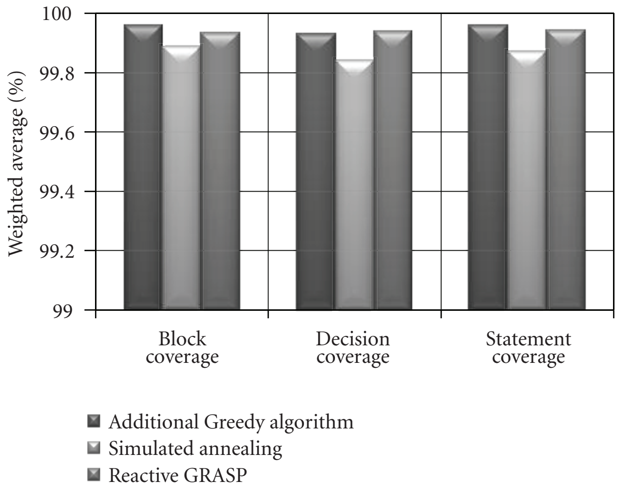


Figure 7: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Metrics), Small Programs.

programs, there is no significant difference between Reactive GRASP and Additional Greedy, in terms of coverage, for space program, neither for simulated annealing nor for genetic algorithm.

4.3.3. Final Analysis. These results qualify the Reactive GRASP algorithm as a good global coverage solution for the prioritization test case problem.

It is also important to mention that the results were consistently similar across coverage criteria. This fact had already been reported by Li et al. [6]. It suggests that there is no need to consider more than one criterion in order to generate good prioritizations of test cases. In addition, we could not find any significant difference in the coverage performance of all algorithms when varying the percentage of test cases being considered.

Note that we have tried from 1% to 100% of test cases for each program and criterion for the four small programs, and the performances of all algorithms remained unaltered. This demonstrated that the ability of the five algorithms discussed here is not deeply related to the number of test cases required to order.

In terms of time, as expected, the use of global approaches, such as both metaheuristic-based algorithms evaluated here, adds an overhead to the process. Considering time efficiency, one can see from Tables 6 and 13 that the Greedy algorithm performed more efficiently than all other algorithms. This algorithm was, on average, 1.491 seconds faster than Additional Greedy algorithm, 8.436 faster than the genetic algorithm, 0.057 faster than the simulated annealing, and almost 50 seconds faster than the Reactive GRASP approach, for the small programs. In terms of relative values, Reactive GRASP was 61.53 times slower than Additional Greedy, 11.68 slower than genetic algorithm, 513.87 slower than simulated annealing, and 730.92 slower than Greedy algorithm. This result demonstrates, once again, the great performance obtained by the Additional Greedy algorithm compared to that of the Greedy algorithm, since it was significantly better, performance-wise, and achieved these results with a very similar execution time. On the other spectrum, we had the Reactive GRASP algorithm, which performed on average 48,456 seconds slower than the Additional Greedy algorithm and 41,511 seconds slower than the genetic algorithm. In favor of both metaheuristic-based approaches is the fact that one may calibrate the time

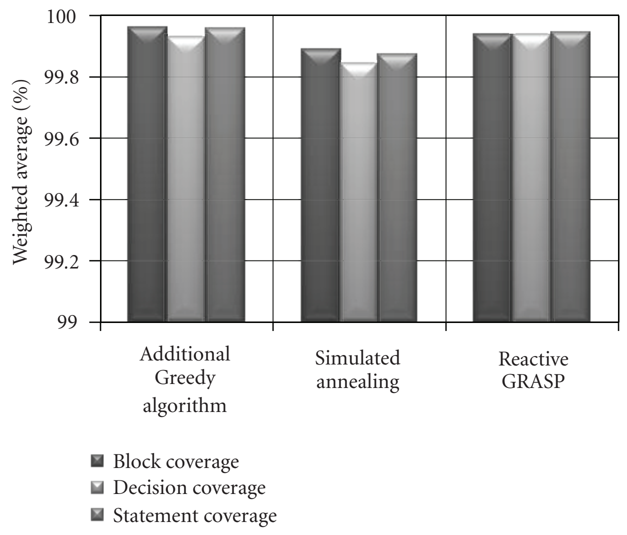


FIGURE 8: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Algorithms), Small Programs.

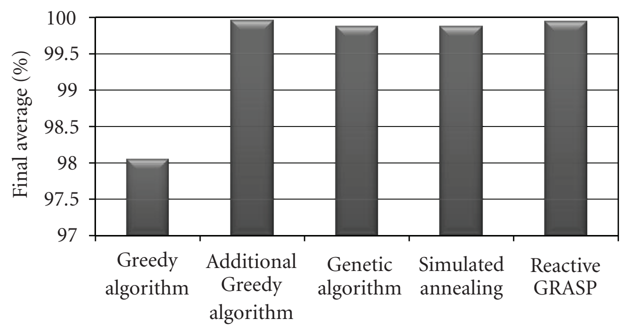


FIGURE 9: Average for Each Algorithm (All Metrics), Small Programs.

required for prioritization depending on time constraints and characteristics of programs and test cases. This flexibility is not present in the Greedy algorithms.

Tables 11 and 18 summarize the results described above.

5. Conclusions and Future Works

Regression testing is an important component of any software development process. Test Case Prioritization is intended to avoid the execution of all test cases every time a change is made to the system. Modeled as an optimization problem, this prioritization problem can be solved with well-known search-based approaches, including metaheuristics.

This paper proposed the use of the Reactive GRASP metaheuristic for the regression test case prioritization problem and compared its performance with other solutions previously reported in literature. Since the Reactive GRASP algorithm performed significantly better—in terms of coverage performance—than the genetic algorithm, Simulated Annealing, and similarly to the Greedy algorithm and it avoids the problems mentioned by Rothermel [2] and Li et al. [6], where Greedy algorithms may fail to choose the optimal test case ordering, the use of the Reactive GRASP algorithm is

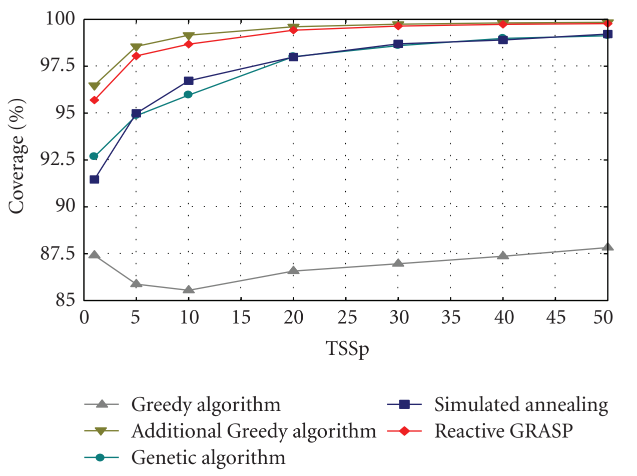


FIGURE 10: APBC (Average Percentage Block Coverage), Comparison among Algorithms for Program Space.

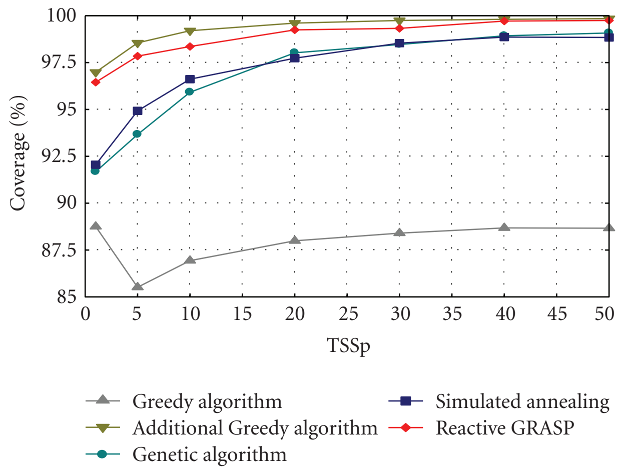


FIGURE 11: APDC (Average Percentage Decision Coverage), Comparison among Algorithms for Program Space.

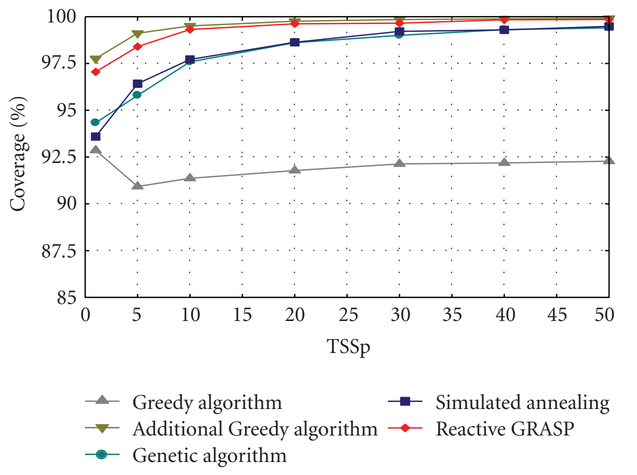


FIGURE 12: APSC (Average Percentage Statement Coverage), Comparison among Algorithms for Program Space.

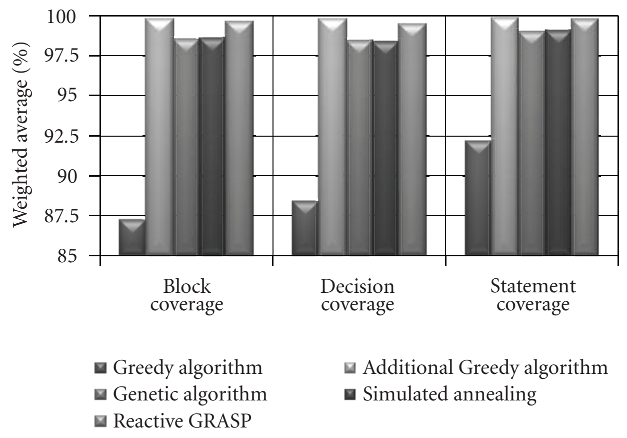


Figure 13: Weighted Average for the Metrics (Comparison among the Metrics), Program Space.

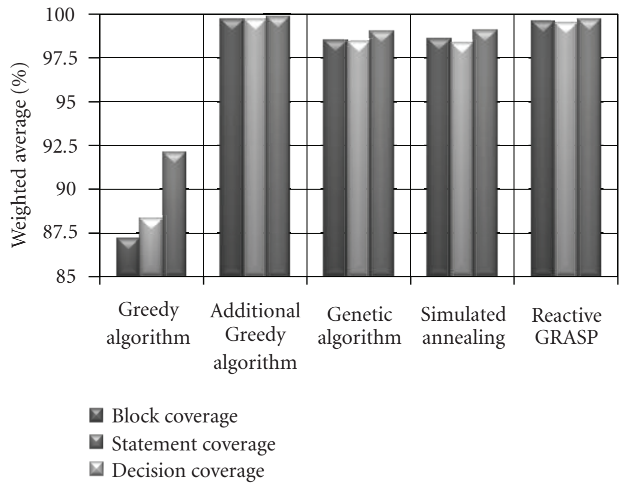


Figure 14: Weighted Average for the Metrics (Comparison among the Algorithms), Program Space.

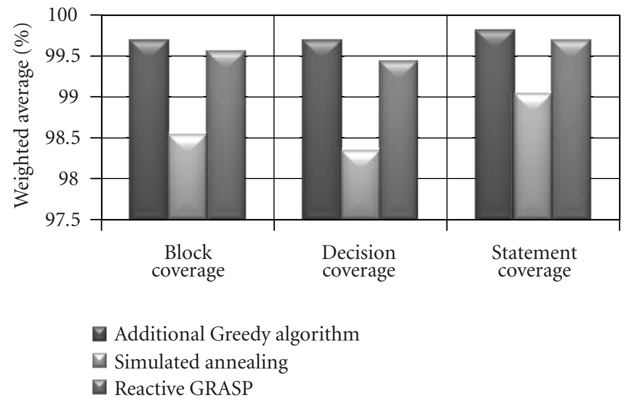


Figure 15: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Metrics), Program Space.

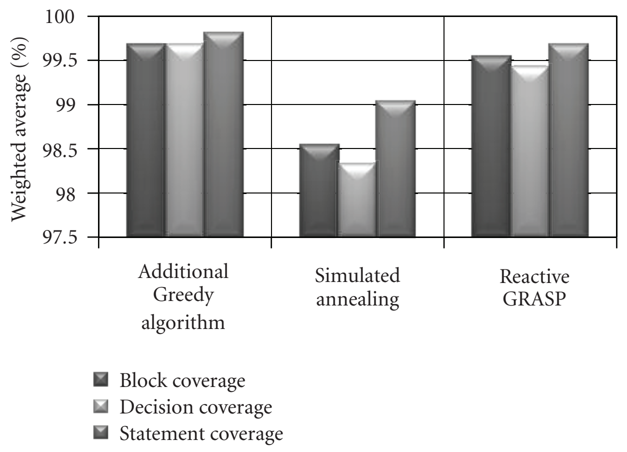


Figure 16: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Algorithms, Program Space.

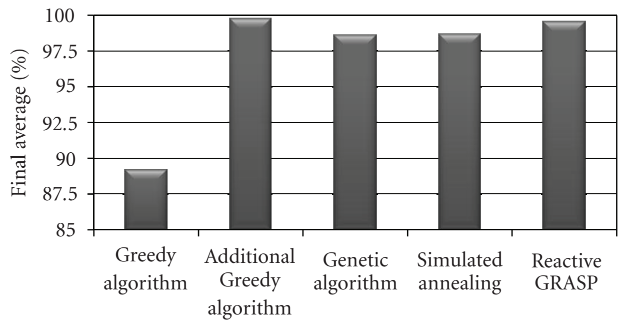


Figure 17: Final Average for Each Algorithm, Program Space.

indicated to the problem of test case prioritization, especially when time constraints are not too critical, since the Reactive GRASP added a considerable overhead.

Our experimental results confirmed also the previous results reported in literature regarding the good performance of the Additional Greedy algorithm. However, some results point out to some interesting characteristics of the Reactive GRASP solution. First, the coverage performance was not significantly worse when compared to that of the Additional Greedy algorithm. In addition, the proposed solution had a more stable behavior when compared to all other solutions. Next, GRASP can be set to work with as many or as little time as available.

As future work, we will evaluate the Reactive GRASP with different number of iterations. This will elucidate whether its good performance was due to its intelligent search heuristics or its computational effort. Finally, other metaheuristics will be considered, including Tabu Search and VNS, among others.

References

[1] M. Fewster and D. Graham, *Software Test Automation*, Addison-Wesley, Reading, Mass, USA, 1st edition, 1994.

[2] G. Rothermel, R. H. Untcn, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[3] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, NY, USA, 2nd edition, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Mass, USA; McGraw-Hill, New York, NY, USA, 2nd edition, 2001.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pp. 179–188, Oxford, UK, September 1999.

[6] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[7] F. Glover and G. Kochenberger, *Handbook of Metaheuristics*, Springer, Berlin, Germany, 1st edition, 2003.

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

[9] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 140–150, London, UK, July 2007.

[10] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[11] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *Proceedings of the 6th Parallel Problem Solving from Nature Conference (PPSN '00)*, pp. 849–858, Paris, France, September 2000.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan, Ann Arbor, Mich, USA, 1975.

[13] M. Harman, "The current state and future of search based software engineering," in *Proceedings of the International Conference on Software Engineering—Future of Software Engineering (FoSE '07)*, pp. 342–357, Minneapolis, Minn, USA, May 2007.

[14] G. Antoniol, M. D. Penta, and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 240–252, Budapest, Hungary, September 2005.

[15] M. Resende and C. Ribeiro, "Greedy randomized adaptative search procedures," in *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds., pp. 219–249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[16] M. Paris and C. C. Ribeiro, "Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment," *INFORMS Journal on Computing*, vol. 12, no. 3, pp. 164–176, 2000.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE '99)*, pp. 191–200, Los Angeles, Calif, USA, 1999.

[18] SEBASE, Software Engineering By Automated Search, September 2009, http://www.sebase.org/applications.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 671284, 8 pages
doi:10.1155/2010/671284

Research Article

A Proposal for Automatic Testing of GUIs Based on Annotated Use Cases

Pedro Luis Mateo Navarro,[1, 2] Diego Sevilla Ruiz,[1, 2] and Gregorio Martínez Pérez[1, 2]

[1] *Departamento de Ingeniería de la Información y las Comunicaciones, University of Murcia, 30071 Murcia, Spain*
[2] *Departamento de Ingeniería y Tecnología de Computadores, University of Murcia, 30071 Murcia, Spain*

Correspondence should be addressed to Pedro Luis Mateo Navarro, pedromateo@um.es

Received 16 June 2009; Accepted 14 August 2009

Academic Editor: Phillip Laplante

This paper presents a new approach to automatically generate GUI test cases and validation points from a set of annotated use cases. This technique helps to reduce the effort required in GUI modeling and test coverage analysis during the software testing process. The test case generation process described in this paper is initially guided by use cases describing the GUI behavior, recorded as a set of interactions with the GUI elements (e.g., widgets being clicked, data input, etc.). These use cases (modeled as a set of initial test cases) are annotated by the tester to indicate interesting variations in widget values (ranges, valid or invalid values) and validation rules with expected results. Once the use cases are annotated, this approach uses the new defined values and validation rules to automatically generate new test cases and validation points, easily expanding the test coverage. Also, the process allows narrowing the GUI model testing to precisely identify the set of GUI elements, interactions, and values the tester is interested in.

1. Introduction

It is well known that testing the correctness of a *Graphical User Interfaces* (GUI) is difficult for several reasons [1]. One of those reasons is that the space of possible interactions with a GUI is enormous, which leads to a large number of GUI states that have to be properly tested (a related problem is to determine the coverage of a set of test cases); the large number of possible GUI states results in a large number of input permutations that have to be considered. Another one is that validating the GUI state is not straightforward, since it is difficult to define which objects (and what properties of these objects) have to be verified.

This paper describes a new approach between Model-less and Model-Based Testing approaches. This new approach describes a GUI test case autogeneration process based on a set of use cases (which are used to describe the GUI behavior) and the annotation (definition of values, validation rules, etc.) of the relevant GUI elements. The process generates automatically all the possible test cases depending on the values defined during the annotation process and incorporates new validation points, where the validation rules have been defined. Then, in a later execution and validation process, the test cases are automatically executed and all the validation rules are verified in order to check if they are met or not.

The rest of the paper is structured as follows. Related work is presented in Section 2. In Section 3 we describe the new testing approach. Annotation, autogeneration, and execution/validation processes are described in Sections 4, 5 and 6, respectively. Finally, Section 8 provides conclusions and lines of future work.

This paper is an extended version of the submitted contribution to the "Informatik 2009: Workshop MoTes09" [2].

2. Related Work

Model-Based GUI Testing approaches can be classified depending on the amount of GUI details that are included in the model. By GUI details we mean the elements which are chosen by the *Coverage Criteria* to faithfully represent the tested GUI (e.g., window properties, widget information and properties, GUI metadata, etc.).

Many approaches usually choose all window and widget properties in order to build a highly descriptive model of the GUI. For example, in [1] (Xie and Memon) and in [3, 4] (Memon et al.) it is described a process based on GUI Ripping, a method which traverses all the windows of the GUI and analyses all the events and elements that may appear to automatically build a model. That model is composed of a set of graphs which represent all the GUI elements (a tree called GUI Forest) all the GUI events and their interaction (Event-Flow Graphs (EFG), and Event Interaction Graphs (EIG)). At the end of the model building process, it has to be verified, fixed, and completed manually by the developers.

Once the model is built, the process explores automatically all the possible test cases. Of those, the developers select the set of test cases identified as meaningful, and the *Oracle Generator* creates the expected output(a *Test Oracle* [5] is a mechanism which generates outputs that a product should have for determining, after a comparison process, whether the product has passed or failed a test (e.g., a previous stored state that has to be met in future test executions). Test Oracles also may be based on a set of rules (related to the product) that have to be validated during test execution). Finally, test cases are automatically executed and their output compared with the Oracle expected results.

As said in [6], the primary problem with these approaches is that as the number of GUI elements increases, the number of event sequences grows exponentially. Another problem is that the model has to be verified, fixed, and completed manually by the testers, with this being a tedious and error-prone process itself. These problems lead to other problems, such a scalability and modifications tolerance. In these techniques, adding a new GUI element (e.g., a new widget or event) has two worrying side effects. First, it may cause the set of generated test cases to grow exponentially (all paths are explored); second, it forces a GUI Model update (and a manual verification and completion) and the regeneration of all affected test cases.

Other approaches use a more restrictive coverage criteria in order to focus the test case autogeneration efforts on only a section of the GUI which usually includes all the relevant elements to be tested. In [7] Vieira et al. describe a method in which enriched UML Diagrams (UML Use Cases and Activity Diagrams) are used to describe which functionalities should be tested and how to test them. The diagrams are enriched in two ways. First, the UML Activity Diagrams are refined to improve the accuracy; second, these diagrams are annotated by using custom UML Stereotypes representing additional test requirements. Once the model is built, an automated process generates test cases from these enriched UML diagrams. In [8] Paiva et al. also describe a UML Diagrams-based model. In this case, however, the model is translated to a formal specification.

The scalability of this approach is better than the previously mentioned because it focuses its efforts only on a section of the model. The diagram refinement also helps to reduce the number of generated test cases. On the other hand, some important limitations make this approach not so suitable for certain scenarios. The building, refining, and annotation processes require a considerable effort since they have to be performed manually, which does not suit some methodologies such as, for instance, Extreme Programming; these techniques also have a low tolerance to modifications; finally, testers need to have a knowledge of the design of the tested application (or have the UML model), which makes impossible to test binary applications or applications with an unknown design.

3. Overview of the Annotated Use Case Guided Approach

In this paper we introduced a new GUI Testing approach between Mode-less and Model-Based testing. The new approach is based on a Test Case Autogeneration process that does not build a complete model of the GUI. Instead, it models two main elements that are the basis of the *test case autogeneration process*.

(i) A Set of Use Cases. These use cases are used to describe the behavior of the GUI to be tested. The use cases are used as the base of the future test cases that are going to be generated automatically.

(ii) A Set of Annotated Elements. This set includes the GUI elements whose values may vary and those with interesting properties to validate. The values define new variaton points for the base use cases; the validation rules define new validation points for the widget properties.

With these elements, the approach addresses the needs of GUI verification, since, as stated in [7], the testing of a scenario can usually be accomplished in three steps: launching the GUI, performing several use cases in sequence, and exiting. The approach combines the benefits from both "Smoke Testing" [4, 9] and "Sanity Testing" [10], as it is able to assure that the system under test will not catastrophically fail and test the main functionality (in the first steps of the development process) and fine-tune checking and properties validation (in the final steps of the development process) by an automated script-based process.

The *test case generation process* described in this paper takes as its starting point the set of use cases (a use case is a sequence of events performed on the GUI; in other words, a use case is a test case) that describe the GUI behavior. From this set, it creates a new set of autogenerated test cases, taking into account the variation points (according to possible different values of widgets) and the validation rules included in the annotations. The resulting set includes all the new autogenerated test cases.

The *test case autogeneration process* can be seen, in a test case level, as the construction of a tree (which initially represents a test case composed of a sequence of test items) to which a new branch is added for each new value defined in the annotations. The validation rules are incorporated later as validation points.

Therefore, in our approach, modeling the GUI and the application behavior does not involve building a model including all the GUI elements and generating a potentially large amount of test cases exploring all the possible event

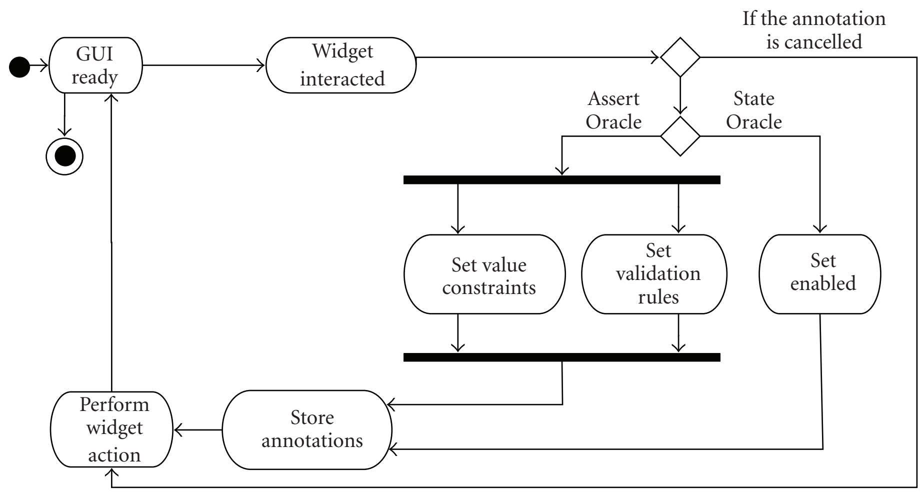


FIGURE 1: Schematic representation of the Widget Annotation Process.

sequences. In the contrary, it works by defining a set of test cases and annotating the most important GUI elements to include both interesting values (range of valid values, out-of-range values) and a set of validation rules (expected results and validation functions) in order to guide the *test case generation process*. It is also not necessary to manually verify, fix, or complete any model in this approach, which removes this tedious and error-prone process from the GUI Testing process and eases the work of the *testers*. These characteristics help to improve the scalability and the modifications tolerance of the approach.

Once the new set of test cases is generated, and the validation rules are incorporated, the process ends with the *test case execution process* (that includes the *validation process*). The result of the execution is a report including any relevant information to the tester (e.g., number of test performed, errors during the execution, values that caused these errors, etc). In the future, the generated test case set can be re-executed in order to perform a regression testing process that checks if the functionality that was previously working correctly is still working.

4. Annotation Process

The *annotation process* is the process by which the tester indicates what GUI elements are important in terms of the following: First, which values can a GUI element hold (i.e., a new set of values or a range), and thus should be tested; second, what constraints should be met by a GUI element at a given time (i.e., validation rules), and thus should be validated. The result of this process is a set of annotated GUI elements which will be helpful during the *test case autogeneration process* in order to identify the elements that represent a variation point, and the constraints that have to be met for a particular element or set of elements. From now on, this set will be called *Annotation Test Case*.

This process could be implemented, for example, using a capture and replay (C&R) tool(a *Capture and Replay Tool* captures events from the tested application and use them to generate test cases that replay the actions performed by the user. Authors of this paper have worked on the design and implementation of such tool as part of a previous research work, accessible on-line at http://sourceforge.net/projects/openhmitester/ and at http://www.um.es/catedraSAES/) . These tools provide the developers with access to the widgets information (and also with the ability to store it), so they could use this information along with the new values and the validation rules (provided by the tester in the *annotation process*) to build the *Annotation Test Case*.

As we can see in Figure 1, the *annotation process*, which starts with the tested application launched and its GUI ready for use, can be performed as follows:

(1) For each widget the tester interacts with (e.g., to perform a click action on a widget or enter some data by using the keyboard), he or she can choose between two options: annotate the widget (go to the next step) or continue as usual (go to step 3).

(2) A widget can be annotated in two ways, depending on the chosen Test Oracle method. It might be an “Assert Oracle” (checks a set of validation rules related to the widget state), or a “State Oracle” (checks if the state of the widget during the *execution process* matches the state stored during the *annotation process*).

(3) The annotations (if the tester has decided to annotate the widget) are recorded by the C&R tool as part of the *Annotation Test Case*. The GUI performs the actions triggered by the user interaction as usual.

(4) The GUI is now ready to continue. The tester can continue interacting with the widgets to annotate them or just finish the process.

The annotated widgets should be chosen carefully as too many annotated widgets in a test case may result in an explosion of test cases. Choosing an accurate value set also helps to get a reasonable test suite size, since during the *test case autogeneration process*, all the possible combinations of annotated widgets and defined values are explored in order to generate a complete test suite which explores all the paths that can be tested. So, these are two important aspects to consider, since the scalability of the generated test suite depends directly on the amount of annotated widgets and the values set defined for them.

Regarding to the definition of the validation rules that are going to be considered in a future *validation process*, the tester has to select the type of the test oracle depending on his or her needs.

For the *annotation process* of this approach we consider two different test oracles.

(i) *Assert Oracles.* These oracles are useful in two ways. First, if the tester defines a new set of values or a range, new test cases will be generated to test these values in the *test case autogeneration process*; second, if the tester also defines a set of validation rules, these rules will be validated during the *execution and validation process*.

(ii) *State Oracles.* These oracles are useful when the tester has to check if a certain widget property or value remains constant during the *execution and validation process* (e.g., a widget that can not be disabled).

In order to define the new values set and the validation rules, it is necessary to incorporate to the process a specification language which allows the tester to indicate which are going to be the new values to be tested and what constraints have to be met. This specification language might be a constraint language as, for instance, the Object Constraint Language (OCL) [11], or a script language as, for instance, Ruby [12]. This kind of languages can be used to allow the tester to identify the annotated object and specify new values and validation rules for it. It is also necessary to establish a mapping between widgets and constructs of the specification language; both languages have mechanisms to implement this feature.

Validation rules also can be set to specify if the tester wants the rules to be validated before (precondition) or after (postcondition) an action is performed on the annotated widget. For example, if the tester is annotating a button (during the *annotation process*), it might be interesting to check some values before the button is pressed, as that button operates with those values; it also might be interesting to check, after that button is pressed, if the obtained result met some constraints. The possibility to decide if the validation rules are going to be checked before of after an action is performed (these are the well-known preconditions and postconditions) allows the tester to perform a more powerful *validation process*. This process could be completed with the definition of an invariant, for example, together with the state oracles, since the invariant is composed of a set of constraints that have to be met through the process (an invariant in this domain would be a condition that is always met in the context of the current dialog.).

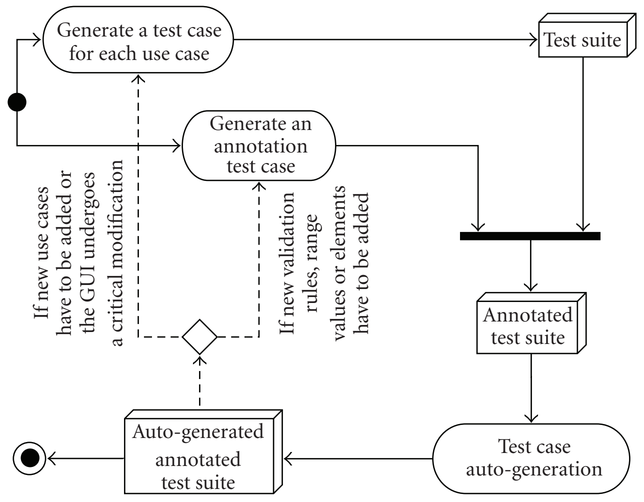

Generate a test case for each use case
Test suite
Generate an annotation test case
If new use cases have to be added or the GUI undergoes a critical modification
If new validation rules, range values or elements have to be added
Annotated test suite
Auto-generated annotated test suite
Test case auto-generation

FIGURE 2: Schematic representation of the Test-Case Auto Generation Process.

5. Test Case AutoGeneration Process

The *test case autogeneration process* is the process that automatically generates a new set of test cases from two elements:

(i) a test suite composed of an initial set of test cases (those corresponding to the use cases that represent the behavior of the GUI);

(ii) an special test case called *Annotation Test Case* which contains all the annotations corresponding to the widgets of a GUI.

As can be seen in Figure 2, the process follows these steps:

(1) As said above, the process is based on an initial test suite and an *Annotation Test Case*. Both together make up the initial *Annotated Test Suite*.

(2) The *test case autogeneration process* explores all the base use cases. For each use case, it generates all the possible variations depending on the values previously defined in the annotations. It also adds validators for ensuring that the defined rules are met. (This process is properly explained at the end of this section).

(3) The result is a new *Annotated Test Suite* which includes all the auto-generated test cases (one for each possible combination of values) and the *Annotation Test Case* used to generate them.

The set of auto-generated test cases can be updated, for example, if the tester has to add or remove new use cases due to a critical modification in the GUI, or if new values or validation rules have to be added or removed. The tester will then update the initial test case set, the *Annotation Test Case*, or both, and will rerun the generation process.

The algorithm corresponding to the *test case autogeneration process* is shown in Algorithm 1.

The process will take as its starting point the *Annotation Test Case* and the initial set of test cases, from which it will generate new test cases taking into account the variation points (the new values) and the validation rules included in the annotations.

For each test case in the initial set, the process inspects every test item (a test case is composed of a set of steps called test items) in order to detect if the widget referred by this test item is included in the annotated widget list. If so, the process generates all the possible variations of the test case (one for each different value, if exist), adding also a validation point if some validation rules have been defined. Once the process has generated all the variations of a test case, it adds them to the result set. Finally, the process returns a set of test cases which includes all the variations of the initial test cases.

Figure 3 is a graphical representation of how the algorithm works. The figure shows an initial test case which includes two annotated test items (an *Annotated Test Item* is a test item that includes a reference to an annotated widget). The annotation for the first widget specifies only two different values (15 and 25); the annotation for the second one specifies two new values (1 and 2) and introduces two validation rules (one related to the *colour* property of the widget and another related to the *text* property). The result of the *test case autogeneration process* will be four new test cases, one for each possible path (15-1, 15-2, 25-1, and 25-2), and a validation point in the second annotated test item which will check if the validation rules mentioned before are met or not.

6. Execution and Validation Process

The *execution and validation process* is the process by which the test cases (auto-generated in the last step) are executed over the target GUI and the validation rules are asserted to check whether the constraints are met. The *test case execution process* executes all the test cases in order. It is very important that for each test case is going to be executed, the GUI must be reset to its initial state in order to ensure that all the test cases are launched and executed under the same conditions.

This feature allows the tester to implement different test configurations, ranging from a set of a few test cases (e.g., to test a component, a single panel, a use case, etc.), to an extensive battery of tests (e.g., for a nightly or regression testing process [4]).

As for the *validation process*, in this paper we describe a Test Oracle based validation process, which uses test oracles [1, 5] to perform widget-level validations (since the validation rules refer to the widget properties) (A *Test Oracle* is a mechanism that generates the expected output that a product should have for determining, after a comparison process, whether the product has passed or failed a test) . The features of the *validation process* vary depending on the oracle method selected during the *annotation process* as we can read below.

(i) *Assert Oracles.* These oracles check if a set of validation rules related to a widget are met or not. Therefore, the tester needs to somehow define a set of validation rules. As said in Section 4 corresponding to the *annotation process*, defining these rules is not straightforward. Expressive and flexible (e.g., constraint or script) languages are needed to allow the tester to define assert rules for the properties of the annotated widget, and, possibly, to other widgets. Another important pitfall is that if the GUI encounters an error, it may reach an unexpected or inconsistent state. Further executing the test case is useless; therefore it is necessary to some mechanism to detect these "bad states" and stop the test case execution (e.g., a special statement which indicates that the *execution and validation process* have to finish if an error is detected).

(ii) *State Oracles.* These oracles check if the state of the widget during the *execution process* matches the state stored during the *annotation process.* To implement this functionality, the system needs to know how to extract the state from the widgets, represent it somehow, and be able to check it for validity. In our approach, it could be implemented using widget adapters which, for example, could represent the state of a widget as a string; so, the validation would be as simple as a string comparison.

The *validation process* may be additionally completed with *Crash Oracles*, which perform an application-level validation (as opposed to widget-level) as they can detect crashes during test case execution. These oracles are used to signal and identify serious problems in the software; they are very useful in the first steps of the development process.

Finally, it is important to remember that there are two important limitations when using test oracles in GUI testing [5]. First, GUI events have to be deterministic in order to be able to predict their outcome (e.g., it would not make sense if the process is validating a property which depends on a random value); second, since the software back-end is not modeled (e.g., data in a data base), the GUI may return a nonexpected state which would be detected as an error (e.g., if the process is validating the output in a database query application, and the content of this database changes during the process).

7. Example

In order to show this process working on a real example, we have chosen a *fixed-term deposit calculator* application. This example application has a GUI (see Figure 4) composed of a set of widgets: a menu bar, three number boxes (two integer and one double), two buttons (one to validate the values and another to operate with them), and a label to output the obtained result. Obviously, there are other widgets in that GUI (i.e., a background panel, text labels, a main window, etc.), but these elements are not of interest for the example.

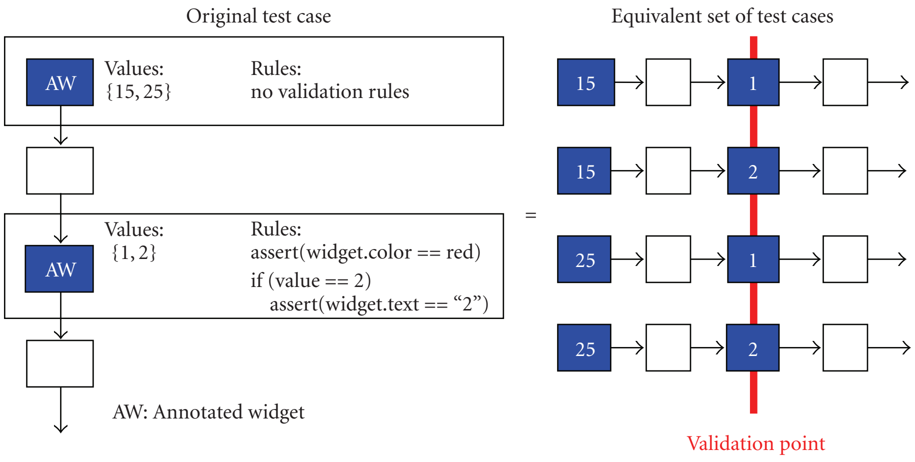

Original test case
AW
Values: {15, 25}
Rules: no validation rules
AW
Values: {1, 2}
Rules: assert(widget.color == red) if (value == 2) assert(widget.text == "2")
AW: Annotated widget
=
Equivalent set of test cases
15
1
15
2
25
1
25
2
Validation point

Figure 3: Test case branching.

```
initial_test_set ⟵ ... // initial test case set
auto_gen_test_set ⟵ {} // auto-generated test case set (empty)
annotated_elements ⟵ ... // user-provided annotations

for all TestCase tc ∈ initial_test_set do
    new_test_cases ⟵ add_test_case (new_test_cases, tc)
    for all TestItem ti ∈ tc do
        if ti.widget ∈ annotated_elements then
            annotations ⟵ annotations_for_widget(ti.widget)
            new_test_cases ⟵ create_new_test_cases (new_test_cases, annotations.values)
            new_test_cases ⟵ add_validation_rules (new_test_cases, annotations.rules)
        end if
    end for
    auto_gen_test_set ⟵ auto_gen_test_set ∪ new_test_cases
end for
return auto_gen_test_set
```

Algorithm 1: Test case autogeneration algorithm.

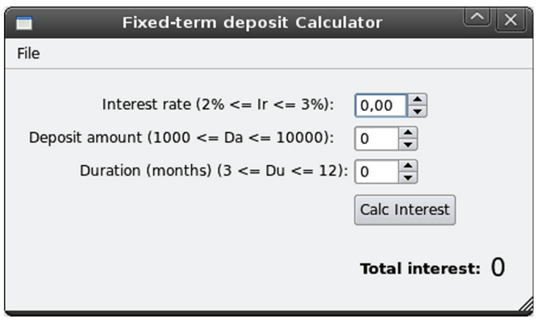

Fixed-term deposit Calculator
File
Interest rate (2% <= Ir <= 3%): 0,00
Deposit amount (1000 <= Da <= 10000): 0
Duration (months) (3 <= Du <= 12): 0
Calc Interest
Total interest: 0

Figure 4: Example dialog.

A common use case for this application is the following:

(1) start the application (the GUI is ready),

(2) insert the values in the three number boxes,

(3) if so, click the "Calc Interest" button and see the result,

(4) exit by clicking the "Exit" option in the "File" menu.

The valid values for the number boxes are the following.

(i) *Interest Rate*. Assume that the interest rate imposed by the bank is between 2 and 3 percent (both included).

(ii) *Deposit Amount*. Assume that the initial deposit amount has to be greater or equal to 1000, and no more than 10 000.

(iii) *Duration*. Assume that the duration in months has to be greater or equal to 3, and less than or equal to 12 months.

The behavior of the buttons is the following. If a number box is out of range, the "Calc Interest" button changes its background colour to red (otherwise, it has to stay white); once it is pressed, it calculates the result using the values, and writes it in the corresponding label. If the values are out of range, the label must read "Data error." In other case, the actual interest amount must be shown.

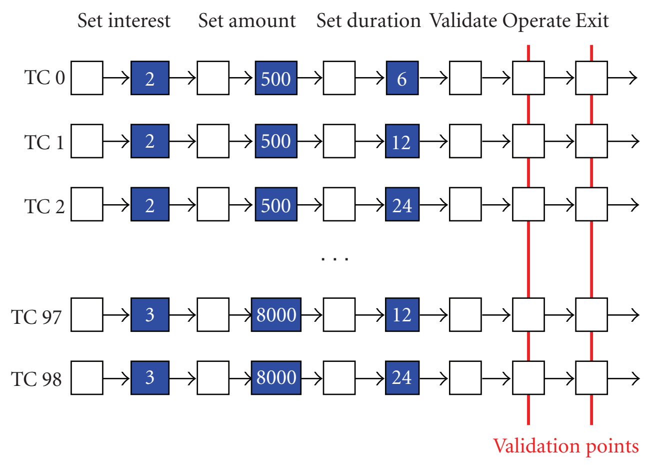


FIGURE 5: Auto-generated test case representation for the example dialog.

Therefore, the annotations for widgets are as follows.

(i) *"Interest rate" spinbox*: a set of values from 2 to 3 with a 0.1 increase.

(ii) *"Deposit amount" spinbox*: a set of values composed of the three values 500, 1000, and 8000. (Note that the value of 500 will introduce a validation error in the test cases.)

(iii) *"Duration" spinbox*: a set of three values, 6, 12, and 24. Again, the last value will not validate.

(iv) *"Calc Interest" button*: depending on the values of the three mentioned text boxes, check the following.

(1) If the values are within the appropriate ranges, the background color of this button must be white, and as a postcondition, the value of the label must hold the calculated interest value (a formula may be supplied to actually verify the value).

(2) Else, if the values are out of range, the background color of the button must be red, and as a postcondition, the value of the label must be "Data error."

Once the initial use case is recorded and the widgets are properly annotated (as said, both processes might be performed with a capture/replay tool), they are used to compose the initial *Annotated Test Suite*, which will be the basis for the *test case autogeneration process*.

We can see the *test case autogeneration process* result in Figure 5. The new *Annotated Test Suite* generated by the process is composed of 99 test cases (11 values for the "Interest rate," 3 different "Deposit amounts," and 3 different "Durations") and a validation point located at the "Calc Interest" button clicking (to check if the values are valid and the background colour accordingly).

The process automatically generates one test case for each possible path by taking into account all the values defined in the annotation process; it also adds validation points where the validation rules have been defined. The new set of auto-generated test cases allows the *tester* to test all the possible variations of the application use cases.

Finally, the *execution and validation process* will execute all the test cases included in the generated *Annotated Test Suite* and will return a report including all the information related to the *execution and validation process*, showing the number of test cases executed, the time spent, and the values not equal to those expected.

8. Conclusions and Future Work

Automated GUI test case generation is an extremely resource intensive process as it is usually guided by a complex and fairly difficult to build GUI model. In this context, this paper presents a new approach for automatically generating GUI test cases based on both GUI use cases (required functionality), and annotations of possible and interesting variations of graphical elements (which generate families of test cases), as well as validation rules for their possible values. This reduces the effort required in test coverage and GUI modeling processes. Thus, this method would help reducing the time needed to develop a software product since the testing and validation processes spend less efforts.

As a statement of direction, we are currently working on an architecture and the details of an open-source implementation which allow us to implement these ideas and future challenges as, for example, to extend the GUI testing process towards the application logic, or to execute a battery of tests in parallel in a distributed environment.

Acknowledgments

This paper has been partially funded by the "Cátedra SAES of the University of Murcia" initiative, a joint effort between Sociedad Anónima de Electrónica Submarina (SAES), http://www.electronica-submarina.com/ and the University of Murcia to work on open-source software, and real-time and critical information systems.

References

[1] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source GUI applications," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 203–212, Los Alamitos, Calif, USA, 2006.

[2] P. Mateo, D. Sevilla, and G. Martínez, "Automated GUI testing validation guided by annotated use cases," in *Proceedings of the 4th Workshop on Model-Based Testing (MoTes '09) in Conjunction with the Annual National Conference of German Association for Informatics (GI '09)*, Lübeck, Germany, September 2009.

[3] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE '03)*, pp. 260–269, Victoria, Canada, November 2003.

[4] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan, "Dart: a framework for regression testing "nightly/daily builds" of GUI applications," in *Proceedings of the IEEE Internacional Conference on Software Maintenance (ICSM '03)*, pp. 410–419, 2003.

[5] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 4, 2007.

[6] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, Minneapolis, Minn, USA, May 2007.

[7] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI testing using a model-driven approach," in *Proceedings of the International Workshop on Automation of Software Test*, pp. 9–14, Shanghai, China, 2006.

[8] A. Paiva, J. Faria, and R. Vidal, "Towards the integration of visual and formal models for GUI testing," *Electronic Notes in Theoretical Computer Science*, vol. 190, pp. 99–111, 2007.

[9] A. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly envolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.

[10] R. S. Zybin, V. V. Kuliamin, A. V. Ponomarenko, V. V. Rubanov, and E. S. Chernov, "Automation of broad sanity test generation," *Programming and Computer Software*, vol. 34, no. 6, pp. 351–363, 2008.

[11] Object Management Group, "Object constraint language (OCL)," version 2.0, OMG document formal/2006-05-01, 2006, http://www.omg.org/spec/OCL/2.0/.

[12] Y. Matsumoto, "Ruby Scripting Language," 2009, http://www.ruby-lang.org/en/.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 540547, 7 pages
doi:10.1155/2010/540547

Research Article

AnnaBot: A Static Verifier for Java Annotation Usage

Ian Darwin

8748 10 Sideroad Adjala, RR 1, Palgrave, ON, Canada L0N 1P0

Correspondence should be addressed to Ian Darwin, ian@darwinsys.com

Received 16 June 2009; Accepted 9 November 2009

Academic Editor: Phillip Laplante

This paper describes AnnaBot, one of the first tools to verify correct use of Annotation-based metadata in the Java programming language. These Annotations are a standard Java 5 mechanism used to attach metadata to types, methods, or fields without using an external configuration file. A binary representation of the Annotation becomes part of the compiled ".class" file, for inspection by another component or library at runtime. Java Annotations were introduced into the Java language in 2004 and have become widely used in recent years due to their introduction in the Java Enterprise Edition 5, the Hibernate object-relational mapping API, the Spring Framework, and elsewhere. Despite that, mainstream development tools have not yet produced a widely-used verification tool to confirm correct configuration and placement of annotations external to the particular runtime component. While most of the examples in this paper use the Java Persistence API, AnnaBot is capable of verifying anyannotation-based API for which "claims"—description of annotation usage—are available. These claims can be written in Java or using a proposed Domain-Specific Language, which has been designed and a parser (but not the code generator) have been written.

1. Introduction

1.1. Java Annotations. Java Annotations were introduced into the language in 2004 [1] and have become widely used in recent years, especially since their introduction in the Java Enterprise Edition. Many open source projects including the Spring [2] and Seam [1] Frameworks, and the Hibernate and Toplink ORMs use annotations. So do many new Sun Java standards, including the Java Standard Edition, the Java Persistence API (an Object Relational Mapping API), the EJB container, and the Java API for XML Web Services (JAX-WS). Until now there has not been a general-purpose tool for independently verifying correct use of these annotations.

The syntax of Java Annotations is slightly unusual—while they are given class-style names (names begin with a capital letter by convention) and are stored in binary .class files, they may not be instantiated using the new operator. Instead, they are placed by themselves in the source code, preceding the element that is to be annotated (see Figure 1). Annotations may be compile-time or run-time; the latter's binary representation becomes part of the compiled "class" file, for inspection by another component at run time. Annotations are used by preceding their name with the "at" sign (@). For example, here is a class with both a compiletime annotation and a runtime annotation

The @WebService annotation from the Java JAX-WS API is a runtime annotation, used to tell a web container that this class should be exposed on the network as a SOAP-based web service. The @Override from Java SE is a compile-time annotation used to tell the compiler or IDE to ensure that the method is in fact overriding a method in the parent class.

1.2. Assertions. An assertion is a claim about the state of a computer program. Assertions have been used for many years in software verification. Goldschlager [4] devotes a section in his chapter on Algorithms to this topic. Voas et al. [5] describes a syntax for adding assertions to existing programs. A simple runtime assertion syntax was introduced using a new language keyword in version 1.4 of the Java programming language [6]. Assertion Languages have been used in verifying hardware designs [7].

1.3. Overview. This research describes a new tool for making assertions about how annotations are to be used, the Annotation Assertion-Based Object Testing tool or AnnaBot.

```
@WebService
public class Fred extends Caveman {
        @Override
        public void callHome() {
                // call Wilma here
        }
}
```

Figure 1: Java SE and EE Annotations applied to a SOAP Web Service class.

```
@Entity public class Person {
  @Id int id;
  @Column(name="given_name")
  public String getFirstName() {...}
```

Figure 2: Portion of a flawed JPA Entity Class.

Section 2 describes the origins of the research in an otherwise-undetected failure and describes how the resulting tool operates in general terms, giving some examples of the classes of errors that it can detect. Section 3 discusses the internal operation of the verification tool and the two methods of writing “claims”: Java code and the planned DSL.

Section 4 describes the experiences in using this verification tool on a couple of sample code bases, one derived from the author’s consulting practice and one from the examples provided with a popular web development framework. Section 5 details some additional research and development that should be undertaken in the future, along lines primarily suggested by Section 4.

Section 6 discusses Related Research, primarily in the area of annotation verification. Section 7 provides details on obtaining the verification tool for anyone who would like to replicate the results, apply it in different domains, or provide improvements to the software.

2. AnnaBot: The Annotation Assertion Verification Tool

Like most good tools, AnnaBot has its origin in a real-life problem. Sun’s Java Enterprise Edition features the Java Persistence Architecture or JPA [8]. The JPA API is designed to facilitate mapping Java objects into relational database tables. JPA is heavily biased towards use of Java5 annotations to attach metadata describing how classes are mapped to and from the relational database. The `@Entity` annotation on a class marks the class as persistent data. Each persistent entity class must have an `@Id` annotation marking the field(s) that constitutes the primary key in the database. Other properties may be annotated (on the field or on the get method) with other annotations such as `@Column` if the database column name must differ from the Java property name—for example, a Java field called *firstName* might map to the SQL column `GIVEN_NAME`. A brief look at the documentation might lead one to write Figure 2.

The code in Figure 2 may fail mysteriously, because the specification states that one can only annotate fields *or* getters. The code as shown will compile correctly, but when deployed and run, the misleading error message you get—if any—depends on whether you are using Hibernate, EclipseLink, or some other Provider behind JPA. Having wasted half a day trying to track down this error, the author concluded that others might make the same mistake (or similar errors, such as annotating the setter method instead of the getter, which are generally silently ignored).

Yet it is not just my one error that makes a tool such as Annabot useful. Consider the scenario of a JPA- or Hibernate-based enterprise software project undergoing development and maintenance over several years. While the odds of an experienced developer annotating a field instead of a method in JPA (or similar error in the other APIs) are small, it could happen due to carelessness. Add in a couple of junior programmers and some tight deadlines, and the odds increase. The consequences of such a mistake range from almost none—since JPA does tend to provide sensible defaults—to considerable time wasted debugging, what appears to be a correct annotation, the ability to verify externally that the annotations have been used correctly, especially considering the almost zero cost of doing so, makes it very much worthwhile.

And, above all, AnnaBot is not limited to JPA. It can—with provision of suitable metadata files—be used to verify correct annotation use in *any* API that uses Java 5 annotations for runtime discovery of metadata.

The tool works by first reading one or more “claim” files (the term `claim` was chosen because `assert` is already a keyword in Java), which claim or assert certain valid conditions about the API(s) being used. It is anticipated that the tool will gradually acquire a library of “standard” claim files for the more common APIs (both client- and server-side) which are using Annotations. For example, Figure 3 is an example of a partial Assertion file for verifying correct use of annotations in the Java Persistence API. The JPA specification [8, Section 2.1.1] states that annotations may be placed before fields (in which case “field access” via Reflection is used to load/store the properties from/to the object) or before the get methods (in which case these methods and the corresponding set methods are used). It notes that “the behavior is unspecified if mapping annotations are applied to both persistent fields and properties or if the XML descriptor specifies use of different access types within a class hierarchy.”

As can be seen in Figure 3, the syntax of a claim file is reminiscent of Java; this is intentional. The `import` statement, for example, has the same syntax and semantics as the like-named statement in Java; it makes annotations available by their unqualified name. The first `if` statement says that any class decorated with the `@Entity` annotation must have a method or field annotated with the `@Id` annotation—“data to be persisted must have a primary key,” in database terms. The methods *field.annotated()* and *method.annotated()* can be called either with a single class name or, as is done here, with a package wildcard with similar syntax and semantics as on Java’s `import` statement (but

```
import javax.persistence.Entity;
import javax.persistence.Id;

claim JPA {
if (class.annotated(javax.persistence.Entity)) {
    require method.annotated(javax.persistence.Id)
        || field.annotated(javax.persistence.Id);
  atMostOne     method.annotated(javax.persistence.ANY)
||      field.annotated(javax.persistence.ANY)
        error "The JPA Spec only allows JPA annotations on methods OR fields";
};
if (class.annotated(javax.persistence.Embeddable)) {
        noneof   method.annotated(javax.persistence.Id) ||
        field.annotated(javax.persistence.Id);
};
}
```

Figure 3: JPA Claim in AnnaBot DSL.

```
claim EJB3Type {
  atMostOne
   class.annotated(javax.ejb.Stateless),

   class.annotated(javax.ejb.Stateful)
       error "Class has conflicting top-level EJB
annotations"
     ;
}
```

Figure 4: EJB Claim file in AnnaBot DSL.

```
public interface PrePostVerify {
       void preVerify();
       void postVerify();
}
```

Figure 5: PrePostVerify interface.

with ANY instead due to an apparent issue with the parser generator). The example also claims that you must not annotate both methods and fields with JPA annotations.

The second `if` statement says that `Embeddable` data items (items which share the primary key and the row storage of a "main" `Entity`) must *not* have an `@Id`—Embeddable objects are not allowed to have a different primary key from the row they are contained in.

As a second example, here is a rudimentary claim file for one aspect of the Enterprise JavaBean (EJB Version 3.0) specification [8].

The example in Figure 4 shows use of `atMostOne` at class level to confirm that no class is annotated with more than one of the mutually-exclusive Annotation types listed (`Stateless` or `Stateful`—these are "Session Beans"). Beside these and Entity, there are other types of EJB that are less common—the example is not intended to be complete or comprehensive.

2.1. Cross-Class Tests. There are some tests that cannot be performed merely by examining a single class. To draw another example from the Java Persistence API, the choice between field annotation and accessor annotation must be consistent not only within a class (as the JPA Claim above tests) but also across all the classes loaded into a given "persistence unit"—usually meaning all the JPA entity classes in an application. The Claim Files shown above cannot handle this. Annabot has recently been modified to support Java-based Claim classes having an optional `implements PrePostVerify` clause. The `PrePostVerify` interface shown in Figure 5 could be used, for example, to allow the claim to set booleans during the claim testing phase and examine them in the `postVerify` method, called as its name suggests after all that the claim verification has been completed.

3. Implementation

The basic operation of Annabot's use of the reflection API is shown in class AnnaBot0.java in Figure 6. This demonstration has no configuration input; it simply hard-codes a single claim about the Java Persistence Architecture API, that only methods *or* only fields be JPA-annotated. This version was a small, simple proof-of-concept and did one thing well.

The Java class under investigation is accessed using Java's built-in Reflection API [9]. There are other reflection-like packages for Java such as Javassist [10] and the Apache Software Foundation's Byte Code Engineering Language [11]. Use of the standard API avoids dependencies on external APIs and avoids both the original author and potential contributors having to learn an additional API. Figure 6 is a portion of the code from the original `AnnaBot0` which determines whether the class under test contains any fields with JPA annotations.

To make the program generally useful, it was necessary to introduce some flexibility into the processing. It was decided to design and implement a Domain-Specific Language [12, 13] to allow declarative rather than procedural specification of additional checking. One will still be able to extend the functionality of AnnaBot using Java, but some will find it more convenient to use the DSL.

The first version of the language uses the Java compiler to convert claim files into a runnable form. Thus, it is slightly

```
Field[] fields = c.getDeclaredFields();
boolean fieldHasJpaAnno = false;
for (Field field : fields) {
        Annotation[] ann =
        field.getDeclaredAnnotations();
        for (Annotation a : ann) {
                Package pkg =
                        a.annotationType().
                        getPackage();
                if (pkg != null &&
                        pkg.getName().
                        startsWith(
                        "javax.persistence"))
                {
                        fieldHasJpaAnno =
                                true;
                        break;
                }
        }
}

// Similar code for checking if any
// JPA annotations are found on methods

// Then the test
if (fieldHasJpaAnno &&
    methodHasJpaAnno) {
    error("JPA Annotations should be on methods or
fields, not both");
}
```

Figure 6: Portion of AnnaBot0.java.

```
package jpa;

import annabot.Claim;
import tree.*;

public class JPAEntityMethodFieldClaim extends Claim
{
        public String getDescription() {
                return "JPA Entities may have field
OR method annotations, not both";
        }
        public Operator[] getClassFilter() {
                return new Operator[] {
                new ClassAnnotated(
                "javax.persistence.Entity"),
                };
        }

        public Operator[] getOperators() {
                return new Operator[] {
                    new AtMostOne(
                    new FieldAnnotated(
                        "javax.persistence.*"),
                    new MethodAnnotated(
                        "javax.persistence.*",))
                };
        }
}
```

Figure 7: JPA Claim written in Java.

more verbose than the Annabot Language. Figure 7 is the JPA claim from AnnaBot0 rewritten as a Java-based Claim.

While it is more verbose than the DSL version, it is still almost entirely declarative. Further, as can be seen, there is a fairly simple translation between the Java declarative form and the DSL. The AnnaBotC compiler tool will translate claim files from the DSL into Java class files.

```
program:      import_stmt*
        CLAIM IDENTIFIER '{'
              stmt+
        '}'
        ;

import_stmt:  IMPORT NAMEINPACKAGE ';'
        ;

// Statement, with or without an
// if ... { stmt } around.
stmt:   IF '(' checks ')' '{' phrase+ '}' ';'
        | phrase
        ;
phrase: verb checks error? ';'
        ;

verb:   REQUIRE | ATMOSTONE | NONEOF;

checks: check
               | NOT check
               | ( check OR check )
               | ( check AND check )
               | ( check ',' check)
               ;

check:  classAnnotated
        | methodAnnotated
        | fieldAnnotated;

classAnnotated: CLASS_ANNOTATED '('
         NAMEINPACKAGE ')';
methodAnnotated:        METHOD_ANNOTATED '('
         NAMEINPACKAGE ')';
fieldAnnotated: FIELD_ANNOTATED '('
         NAMEINPACKAGE
        ( ',' MEMBERNAME )? ')'
        ;

error:  ERROR QSTRING;
```

Figure 8: EBNF for DSL.

The structure of the DSL suggested that an LL or an LR parser would be adequate. While any type of parser may in theory be written by hand, software developers have used parser generator tools for three decades, starting with the widely-known UNIX tool YACC [14]. There are many “parser generator” tools available for the Java developer; a collection of them is maintained on the web [15]. After some evaluation of the various parser generators, a decision was made to use Antlr [16]. This was based in part on Parr’s enthusiasm for his compiler tool, and his comments in [16] about LR versus LL parsing (YACC is LR, Antlr is LL). “In contrast, LL recognizers are goal-oriented. They start with a rule in mind and then try to match the alternatives. For this reason, LL is easier for humans to understand because it mirrors our own innate language recognition mechanism...”

A basic EBNF description of the AnnaBot input language is in Figure 8; the lexical tokens (names in upper case) have been omitted as they are obvious. Figures 3 and 4 provide examples of the DSL.

This provides sufficient structure for the current design of the DSL to be matched. The parser has been fully implemented in Antlr, but the code generation is not yet implemented. For the present, claims files are written in Java (as in Figure 7).

Table 1: Usage results.

Codebase	Kloc	Classes Entity/Total	Errors	Time (Seconds)
seambay	2.27	7, 22	0	0.6
TCP	26.8	94/156	0	~3 (see Section 4.1)

4. Experiences and Evaluation

The current version has been used to verify a medium-scale web site that is currently being redeveloped using the Seam framework; Seam uses JPA and, optionally, EJB components. The site, which consists of approximately one hundred JPA “entity” classes and a growing number of controller classes, is being developed at the Toronto Centre for Phenogenomics, or TCP [17]. Although AnnaBot has not yet found any actual claim violations in the TCP software—many of the Entity classes were generated automatically by the Hibernate tooling provided for this purpose—AnnaBot provides an ongoing verification that no incorrect annotation uses are being added during development. With a small starter set of four JPA and EJB claims in service, the process takes a few seconds to verify about a hundred and fifty class files. It thus has a very low cost for the potential gain of finding a column that might not get mapped correctly or a constraint that might be violated at runtime in the database.

As a second example, the program was tested against the `seambay` package from the Seam Framework [3] examples folder. As the name implies, `seambay` is an auction site, a small-scale imitation of the well-known eBay.com site. Seam version 2.2.0.GA provides a version of `seambay` that has only 20–30 Java classes depending on version, perhaps to show how powerful Seam itself is as a framework. AnnaBot scanned these in 0.6 seconds with four JPA claims, and again unsurprisingly found no errors (Table 1).

The result of these tests indicate that AnnaBot is a very low-overhead and functioning method of verifying correct use of annotations.

4.1. A Note on Performance. The one risk as the program's use grows is the $O(m\,n)$ running time for the testing phase—for m tests against n target classes, the program must obviously perform $m \times n$ individual tests. To measure this, I created four additional copies of the JPA Entity claim, to double the value of m, and re-ran AnnaBot on the TCP corpus of ~150 classes. Running time for the initialization phase (discovering and loading classes) actually went down slightly; running time for the testing phase went from 2.3 seconds with 4 claim files to 3.2 seconds with 8. All times were measured using Java's `System.currentTimeMillis()` on a basically otherwise idle laptop (An AMD Athlon-64, 3500+, 1800 MHz single core, 2 GB of RAM, running OpenBSD 4.4 with the X Window System and KDE; AnnaBot running in Eclipse 3.2 using Java JDK 1.6.0), and averaged over 3 or more runs. These results would indicate that AnnaBot's performance scales reasonably when running tests. Therefore, adding a useful library of claim files should not create a significant obstacle to running the tests periodically.

5. Future Development

As stated above, experience has shown this to be a low-cost verification method for software. The following enhancements are under consideration for the tool, subject to time and availability.

5.1. Finish Implementation of the Compiler. The parser is written and tested, and the DSL-based claims in this paper have been parsed using the version current as of November, 2009. The parser needs actions added to generate the class; it is planned to use the Javassist API [10] to generate the .class file corresponding to the claim file being read.

5.2. Convert from Java Reflection to Javassist. Given the use of Javassist in the compiler, it may be wise to convert the inspection code in AnnaBot to use Javassist in the verification as well. Java's built-in reflection works well enough, but it requires that all annotation classes be available on the CLASSPATH. This can become an irritation in using AnnaBot; using Javassist to do the code inspection should eliminate this.

5.3. More Fine-Grained Verification. At present only the presence or absence of a given annotation can be verified. For more complete verification, it ought to be possible to interrogate the attributes within an Annotation, for example,

```
@Entity @Table(name="this is not a
valid table name")
```

This would, like most of the examples in this paper, be considered acceptable at compile time (the *name* element merely requires a Java String as its value), but would fail at run time, since “this is not a valid table name” is indeed not a valid table name in the SQL language of most relational databases.

Some consideration has been given to a syntax for this, probably using regular expressions similar to what is done for method names, but nothing concrete has been established.

5.4. Run as Plug-in. The Eclipse IDE [18] and the FindBugs static testing tool [19] are both widely used tools, and both provide extensibility via plug-ins. However, FindBugs uses its own mechanism for parsing the binary class files (for an example, see Listing 4 of Goetz [20]). It may or may not be feasible to reconcile these differing methods of reading class files to make AnnaBot usable as a FindBugs plug-in. Failing this, it would definitely be worth while to make AnnaBot usable as an Eclipse plug-in, given the wide adoption of Eclipse in the Java developer community.

6. Related Research

Relatively little attention has been paid to developing tools that assist in verifying correct use of annotations. Eichberg et al. [21] produced an Eclipse plug-in which uses a different, and I think rather clever, approach: they preprocess the Java class into an XML representation of the byte-code, then use XPath to query for correct use of annotations. This allows them to verify some nonannotation-related attributes of the software's specification conformance. For example, they can check that EJB classes have a no-argument constructor (which Annabot can easily do at present). They can also verify that such a class does not create new threads. Annabot cannot do this at present since that analysis requires inspection of the bytecode to check for "monitor locking" machine instructions. However, this is outside my research's scope of verifying the correct use of annotations. It could be implemented by using one of the non-Sun "reflection" APIs mentioned in Section 3.

The downside of Eichberg's approach is that all classes in the target system must be preprocessed, whereas AnnaBot simply examines target classes by reflection, making it faster and simpler to use.

Noguera and Pawlak [22] explore an alternate approach. They produce a rather powerful annotation verifier called AVal. However, as they point out, "AVal follows the idea that annotations should describe the way in which they should be validated, and that self validation is expressed by meta-annotations (@Validators)." Since my research goal was to explore validation of existing annotation-based APIs provided by Sun, SpringSource, Hibernate project, and others, I did not pursue investigation of procedures that would have required attempting to convince each API provider to modify their annotations.

JavaCOP by Andreae [23] provides a very comprehensive type checking system for Java programs; it provides several forms of type checking, but goes beyond the use of annotations to provide a complete constraint system.

The work of JSR-305 [24] has been suggested as relevant. JSR-305 is more concerned with specifying new annotations for making assertions about standard Java than with ensuring correct use of annotations in code written to more specialized APIs. As the project describes itself, "This JSR will work to develop standard annotations (such as @NonNull) that can be applied to Java programs to assist tools that detect software defects."

Similarly, the work of JSR-308 [25], an offshoot of JSR-305, has been suggested as relevant, but it is concerned with altering the syntax of Java itself to extend the number of places where annotations are allowed. For example, it would be convenient if annotations could be applied to Java 5+ Type Parameters. This is not allowed by the compilers at present but will be when JSR-308 becomes part of the language, possibly as early as Java 7.

Neither JSR-305 nor JSR-308 provides any support for finding misplaced annotations.

7. Where to Obtain the Software

The home page on the web for the project is http://www.darwinsys.com/annabot/.

The source code can be obtained by Anonymous CVS from the author's server, using these command-line tools or their equivalent in an IDE or other tool:

```
export CVSROOT=:pserver:\
anoncvs@cvs.darwinsys.com:/cvspublic
cvs checkout annabot
```

Contributions of patches or new Claim files will be most gratefully received.

Acknowledgments

Research performed in partial fulfillment of the M.Sc degree at Staffordshire University. The web site development which led to the idea for AnnaBot was done while under contract to The Toronto Centre for Phenogenomics [17]; however, the software was developed and tested on the author's own time. Ben Rady suggested the use of Javassist in the Compiler. Several anonymous reviewers contributed significantly to the readability and accuracy of this paper.

References

[1] Java 5 Annotations, November 2009, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[2] Spring Framework Home Page, October 2009, http://www.springsource.org/.

[3] G. King, "Seam Web/JavaEE Framework," October 2009, http://www.seamframework.org/.

[4] L. Goldschlager, *Computer Science: A Modern Introduction*, Prentice-Hall, Upper Saddle River, NJ, USA, 1992.

[5] J. Voas, et al., "A Testability-based Assertion Placement Tool for Object-Oriented Software," October 1997, http://hissa.nist.gov/latex/htmlver.html.

[6] "Java Programming with Assertions," November 2009, http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html.

[7] J. A. Darringer, "The application of program verification techniques to hardware verification," in *Proceedings of the Annual ACM IEEE Design Automation Conference*, pp. 373–379, ACM, 1988.

[8] "EJB3 and JPA Specifications," November 2009, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[9] I. Darwin, "The reflection API," in *Java Cookbook*, chapter 25, O'Reilly, Sebastopol, Calif, USA, 2004.

[10] Javassist bytecode manipulation library, November 2009, http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[11] Apache BCEL—Byte Code Engineering Library, November 2009, http://jakarta.apache.org/bcel/.

[12] J. Bentley, "Programming pearls: little languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.

[13] I. Darwin, "PageUnit: A "Little Language" for Testing Web Applications," Staffordshire University report, 2006, http://www.pageunit.org/.

[14] S. Johnson, "YACC: yet another compiler-compiler," Tech. Rep. CSTR-32, Bell Laboratories, Madison, Wis, USA, 1978.

[15] "Open Source Parser Generators in Java," April 2009, http://java-source.net/open-source/parser-generators.

[16] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, Pragmatic Bookshelf, Raleigh, NC, USA, 2007.

[17] Toronto Centre for Phenogenomics, April 2009, http://www.phenogenomics.ca/.

[18] Eclipse Foundation, Eclipse IDE project, November 2009, http://www.eclipse.org/.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.
[20] B. Goetz, "Java theory and practice: testing with leverage—part 1," April 2009, http://www.ibm.com/developerworks/library/j-jtp06206.html.
[21] M. Eichberg, T. Schäfer, and M. Mezini, "Using annotations to check structural properties of classes," in *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE '05)*, pp. 237–252, Edinburgh, UK, April 2005.
[22] C. Noguera and R. Pawlak, "AVal: an extensible attribute-oriented programming validator for Java," in *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06)*, pp. 175–183, Philadelphia, Pa, USA, September 2006.
[23] C. Andreae, "JavaCOP—User-defined Constraints on Java Programs," November 2009, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.3014&rep=rep1&type=pdf.
[24] JSR-305, November 2009, http://jcp.org/en/jsr/detail?id=305.
[25] JSR-308, November 2009, http://jcp.org/en/jsr/detail?id=308.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 620836, 18 pages
doi:10.1155/2010/620836

Research Article

Software Test Automation in Practice: Empirical Observations

Jussi Kasurinen, Ossi Taipale, and Kari Smolander

Department of Information Technology, Laboratory of Software Engineering, Lappeenranta University of Technology, P.O. Box 20, 53851 Lappeenranta, Finland

Correspondence should be addressed to Jussi Kasurinen, jussi.kasurinen@lut.fi

Received 10 June 2009; Revised 28 August 2009; Accepted 5 November 2009

Academic Editor: Phillip Laplante

The objective of this industry study is to shed light on the current situation and improvement needs in software test automation. To this end, 55 industry specialists from 31 organizational units were interviewed. In parallel with the survey, a qualitative study was conducted in 12 selected software development organizations. The results indicated that the software testing processes usually follow systematic methods to a large degree, and have only little immediate or critical requirements for resources. Based on the results, the testing processes have approximately three fourths of the resources they need, and have access to a limited, but usually sufficient, group of testing tools. As for the test automation, the situation is not as straightforward: based on our study, the applicability of test automation is still limited and its adaptation to testing contains practical difficulties in usability. In this study, we analyze and discuss these limitations and difficulties.

1. Introduction

Testing is perhaps the most expensive task of a software project. In one estimate, the testing phase took over 50% of the project resources [1]. Besides causing immediate costs, testing is also importantly related to costs related to poor quality, as malfunctioning programs and errors cause large additional expenses to software producers [1, 2]. In one estimate [2], software producers in United States lose annually 21.2 billion dollars because of inadequate testing and errors found by their customers. By adding the expenses caused by errors to software users, the estimate rises to 59.5 billion dollars, of which 22.2 billion could be saved by making investments on testing infrastructure [2]. Therefore improving the quality of software and effectiveness of the testing process can be seen as an effective way to reduce software costs in the long run, both for software developers and users.

One solution for improving the effectiveness of software testing has been applying automation to parts of the testing work. In this approach, testers can focus on critical software features or more complex cases, leaving repetitive tasks to the test automation system. This way it may be possible to use human resources more efficiently, which consequently may contribute to more comprehensive testing or savings in the testing process and overall development budget [3]. As personnel costs and time limitations are significant restraints of the testing processes [4, 5], it also seems like a sound investment to develop test automation to get larger coverage with same or even smaller number of testing personnel. Based on market estimates, software companies worldwide invested 931 million dollars in automated software testing tools in 1999, with an estimate of at least 2.6 billion dollars in 2004 [6]. Based on these figures, it seems that the application of test automation is perceived as an important factor of the test process development by the software industry.

The testing work can be divided into manual testing and automated testing. Automation is usually applied to running repetitive tasks such as unit testing or regression testing, where test cases are executed every time changes are made [7]. Typical tasks of test automation systems include development and execution of test scripts and verification of test results. In contrast to manual testing, automated testing is not suitable for tasks in which there is little repetition [8], such as explorative testing or late development verification tests. For these activities manual testing is more suitable, as building automation is an extensive task and feasible only if the case is repeated several times [7, 8]. However, the division between automated and manual testing is not as straightforward in practice as it seems; a large concern is also the testability of the software [9], because every piece of code can be made poorly enough to be impossible to test it reliably, therefore making it ineligible for automation.

Software engineering research has two key objectives: the reduction of costs and the improvement of the quality of products [10]. As software testing represents a significant part of quality costs, the successful introduction of test automation infrastructure has a possibility to combine these two objectives, and to overall improve the software testing processes. In a similar prospect, the improvements of the software testing processes are also at the focus point of the new software testing standard ISO 29119 [11]. The objective of the standard is to offer a company-level model for the test processes, offering control, enhancement and follow-up methods for testing organizations to develop and streamline the overall process.

In our prior research project [4, 5, 12–14], experts from industry and research institutes prioritized issues of software testing using the Delphi method [15]. The experts concluded that process improvement, test automation with testing tools, and the standardization of testing are the most prominent issues in concurrent cost reduction and quality improvement. Furthermore, the focused study on test automation [4] revealed several test automation enablers and disablers which are further elaborated in this study. Our objective is to observe software test automation in practice, and further discuss the applicability, usability and maintainability issues found in our prior research. The general software testing concepts are also observed from the viewpoint of the ISO 29119 model, analysing the test process factors that create the testing strategy in organizations. The approach to achieve these objectives is twofold. First, we wish to explore the software testing practices the organizations are applying and clarify the current status of test automation in the software industry. Secondly, our objective is to identify improvement needs and suggest improvements for the development of software testing and test automation in practice. By understanding these needs, we wish to give both researchers and industry practitioners an insight into tackling the most hindering issues while providing solutions and proposals for software testing and automation improvements.

The study is purely empirical and based on observations from practitioner interviews. The interviewees of this study were selected from companies producing software products and applications at an advanced technical level. The study included three rounds of interviews and a questionnaire, which was filled during the second interview round. We personally visited 31 companies and carried out 55 structured or semistructured interviews which were tape-recorded for further analysis. The sample selection aimed to represent different polar points of the software industry; the selection criteria were based on concepts such as operating environments, product and application characteristics (e.g., criticality of the products and applications, real time operation), operating domain and customer base.

The paper is structured as follows. First, in Section 2 we introduce comparable surveys and related research. Secondly, the research process and the qualitative and quantitative research methods are described in Section 3. Then the survey results are presented in Section 4 and the interview results are presented in Section 5. Finally, the results and observations and their validity are discussed in Section 6 and closing conclusions are discussed in Section 7.

2. Related Research

Besides our prior industry-wide research in testing [4, 5, 12–14], software testing practices and test process improvement have also been studied by others, like Ng et al. [16] in Australia. Their study applied the survey method to establish knowledge on such topics as testing methodologies, tools, metrics, standards, training and education. The study indicated that the most common barrier to developing testing was the lack of expertise in adopting new testing methods and the costs associated with testing tools. In their study, only 11 organizations reported that they met testing budget estimates, while 27 organizations spent 1.5 times the estimated cost in testing, and 10 organizations even reported a ratio of 2 or above. In a similar vein, Torkar and Mankefors [17] surveyed different types of communities and organizations. They found that 60% of the developers claimed that verification and validation were the first to be neglected in cases of resource shortages during a project, meaning that even if the testing is important part of the project, it usually is also the first part of the project where cutbacks and downscaling are applied.

As for the industry studies, a similar study approach has previously been used in other areas of software engineering. For example, Ferreira and Cohen [18] completed a technically similar study in South Africa, although their study focused on the application of agile development and stakeholder satisfaction. Similarly, Li et al. [19] conducted research on the COTS-based software development process in Norway, and Chen et al. [20] studied the application of open source components in software development in China. Overall, case studies covering entire industry sectors are not particularly uncommon [21, 22]. In the context of test automation, there are several studies and reports in test automation practices (such as [23–26]). However, there seems to be a lack of studies that investigate and compare the practice of software testing automation in different kinds of software development organizations.

In the process of testing software for errors, testing work can be roughly divided into manual and automated testing, which both have individual strengths and weaknesses. For example, Ramler and Wolfmaier [3] summarize the difference between manual and automated testing by suggesting that automation should be used to prevent further errors in working modules, while manual testing is better suited for finding new and unexpected errors. However, how and where the test automation should be used is not so straightforward issue, as the application of test automation seems to be a strongly diversified area of interest. The application of test automation has been studied for example in test case generation [27, 28], GUI testing [29, 30] and workflow simulators [31, 32] to name a few areas. Also according to Bertolino [33], test automation is a significant area of interest in current testing research, with a focus on improving the degree of automation by developing advanced techniques for generating the test inputs, or by finding support procedures such as error report generation to ease the supplemental workload. According to the same study, one of the dreams involving software testing is 100% automated testing. However, for example Bach's [23] study observes that this cannot be achieved, as all automation ultimately requires human intervention, if for nothing else, then at least to diagnose results and maintain automation cases.

The pressure to create resource savings are in many case the main argument for test automation. A simple and straightforward solution for building automation is to apply test automation just on the test cases and tasks that were previously done manually [8]. However, this approach is usually unfeasible. As Persson and Yilmaztürk [26] note, the establishment of automated testing is a costly, high risk project with several real possibilities for failure, commonly called as "pitfalls". One of the most common reasons why creating test automation fails, is that the software is not designed and implemented for testability and reusability, which leads to architecture and tools with low reusability and high maintenance costs. In reality, test automation sets several requisites on a project and has clear enablers and disablers for its suitability [4, 24]. In some reported cases [27, 34, 35], it was observed that the application of test automation with an ill-suited process model may be even harmful to the overall process in terms of productivity or cost-effectiveness.

Models for estimating testing automation costs, for example by Ramler and Wolfmaier [3], support decision-making in the tradeoff between automated and manual testing. Berner et al. [8] also estimate that most of the test cases in one project are run at least five times, and one fourth over 20 times. Therefore the test cases, which are done constantly like smoke tests, component tests and integration tests, seem like ideal place to build test automation. In any case, there seems to be a consensus that test automation is a plausible tool for enhancing quality, and consequently, reducing the software development costs in the long run if used correctly.

Our earlier research on the software test automation [4] has established that test automation is not as straightforward to implement as it may seem. There are several characteristics which enable or disable the applicability of test automation. In this study, our decision was to study a larger group of industry organizations and widen the perspective for further analysis. The objective is to observe, how the companies have implemented test automation and how they have responded to the issues and obstacles that affect its suitability in practice. Another objective is to analyze whether we can identify new kind of hindrances to the application of test automation and based on these findings, offer guidelines on what aspects should be taken into account when implementing test automation in practice.

3. Research Process

3.1. Research Population and Selection of the Sample. The population of the study consisted of organization units (OUs). The standard ISO/IEC 15504-1 [36] specifies an organizational unit (OU) as a part of an organization that is the subject of an assessment. An organizational unit deploys one or more processes that have a coherent process context and operates within a coherent set of business goals. An organizational unit is typically part of a larger organization, although in a small organization, the organizational unit may be the whole organization.

The reason to use an OU as the unit for observation was that we wanted to normalize the effect of the company size to get comparable data. The initial population and population criteria were decided based on the prior research on the subject. The sample for the first interview round consisted of 12 OUs, which were technically high level units, professionally producing software as their main process. This sample also formed the focus group of our study. Other selection criteria for the sample were based on the polar type selection [37] to cover different types of organizations, for example different businesses, different sizes of the company, and different kinds of operation. Our objective of using this approach was to gain a deep understanding of the cases and to identify, as broadly as possible, the factors and features that have an effect on software testing automation in practice.

For the second round and the survey, the sample was expanded by adding OUs to the study. Selecting the sample was demanding because comparability is not determined by the company or the organization but by comparable processes in the OUs. With the help of national and local authorities (the network of the Finnish Funding Agency for Technology and Innovation) we collected a population of 85 companies. Only one OU from each company was accepted to avoid the bias of over-weighting large companies. Each OU surveyed was selected from a company according to the population criteria. For this round, the sample size was expanded to 31 OUs, which also included the OUs from the first round. The selection for expansion was based on probability sampling; the additional OUs were randomly entered into our database, and every other one was selected for the survey. In the third round, the same sample as in the first round was interviewed. Table 1 introduces the business domains, company sizes and operating areas of our focus OUs. The company size classification is taken from [38].

3.2. Interview Rounds. The data collection consisted of three interview rounds. During the first interview round, the designers responsible for the overall software structure and/or module interfaces were interviewed. If the OU did not have separate designers, then the interviewed person was selected from the programmers based on their role in

Table 1: Description of the interviewed focus OUs (see also the appendix).

OU	Business	Company size[1]/Operation
Case A	MES[1] producer and electronics manufacturer	Small/National
Case B	Internet service developer and consultant	Small/National
Case C	Logistics software developer	Large/National
Case D	ICT consultant	Small/National
Case E	Safety and logistics system developer	Medium/National
Case F	Naval software system developer	Medium/International
Case G	Financial software developer	Large/National
Case H	MES[1] producer and logistics service systems provider	Medium/International
Case I	SME[2] business and agriculture ICT service provider	Small/National
Case J	Modeling software developer	Large/International
Case K	ICT developer and consultant	Large/International
Case L	Financial software developer	Large/International

[1]Manufacturing Execution System; [2]Small and Medium-sized Enterprise, definition [38].

the process to match as closely as possible to the desired responsibilities. The interviewees were also selected so that they came from the same project, or from positions where the interviewees were working on the same product. The interviewees were not specifically told not to discuss the interview questions together, but this behavior was not encouraged either. In a case where an interviewee asked for the questions or interview themes beforehand, the person was allowed access to them in order to prepare for the meeting. The interviews in all three rounds lasted about an hour and had approximately 20 questions related to the test processes or test organizations. In two interviews, there was also more than one person present.

The decision to interview designers in the first round was based on the decision to gain a better understanding on the test automation practice and to see whether our hypothesis based on our prior studies [4, 5, 12–14] and supplementing literature review were still valid. During the first interview round, we interviewed 12 focus OUs, which were selected to represent different polar types in the software industry. The interviews contained semi-structured questions and were tape-recorded for qualitative analysis. The initial analysis of the first round also provided ingredients for the further elaboration of important concepts for the latter rounds. The interview rounds and the roles of the interviewees in the case OUs are described in Table 2.

The purpose of the second combined interview and survey round was to collect multiple choice survey data and answers to open questions which were based on the first round interviews. These interviews were also tape-recorded for the qualitative analysis of the focus OUs, although the main data collection method for this round was a structured survey. In this round, project or testing managers from 31 OUs, including the focus OUs, were interviewed. The objective was to collect quantitative data on the software testing process, and further collect material on different testing topics, such as software testing and development. The collected survey data could also be later used to investigate observations made from the interviews and vice versa, as suggested in [38]. Managers were selected for this round, as they tend to have more experience on software projects, and have a better understanding of organizational and corporation level concepts and the overall software process beyond project-level activities.

The interviewees of the third round were testers or, if the OU did not have separate testers, programmers who were responsible for the higher-level testing tasks. The interviews in these rounds were also semi-structured and concerned the work of the interviewees, problems in testing (e.g., increasing complexity of the systems), the use of software components, the influence of the business orientation, testing resources, tools, test automation, outsourcing, and customer influence for the test processes.

The themes in the interview rounds remained similar, but the questions evolved from general concepts to more detailed ones. Before proceeding to the next interview round, all interviews with the focus OUs were transcribed and analyzed because new understanding and ideas emerged during the data analysis. This new understanding was reflected in the next interview rounds. The themes and questions for each of the interview rounds can be found on the project website http://www2.it.lut.fi/project/MASTO/.

3.3. Grounded Analysis Method. The grounded analysis was used to provide further insight into the software organizations, their software process and testing policies. By interviewing people of different positions from the production organization, the analysis could gain additional information on testing- and test automation-related concepts like different testing phases, test strategies, testing tools and case selection methods. Later this information could be compared between organizations, allowing hypotheses on test automation applicability and the test processes themselves.

The grounded theory method contains three data analysis steps: open coding, axial coding and selective coding. The objective for open coding is to extract the categories from the data, whereas axial coding identifies the connections between the categories. In the third phase, selective coding, the core category is identified and described [39]. In practice, these

Table 2: Interviewee roles and interview rounds.

Round type	Number of interviews	Interviewee role	Description
(1) Semistructured	12 focus OUs	Designer or Programmer	The interviewee is responsible for software design or has influence on how software is implemented
(2) Structured/ Semistructured	31 OUs quantitative, including 12 focus OUs qualitative	Project or testing manager	The interviewee is responsible for software development projects or test processes of software products
(3) Semistructured	12 focus OUs	Tester	The interviewee is a dedicated software tester or is responsible for testing the software product

steps overlap and merge because the theory development process proceeds iteratively. Additionally, Strauss and Corbin [40] state that sometimes the core category is one of the existing categories, and at other times no single category is broad enough to cover the central phenomenon.

The objective of open coding is to classify the data into categories and identify leads in the data, as shown in Table 3. The interview data is classified to categories based on the main issue, with observation or phenomenon related to it being the codified part. In general, the process of grouping concepts that seem to pertain to the same phenomena is called categorizing, and it is done to reduce the number of units to work with [40]. In our study, this was done using ATLAS.ti software [41]. The open coding process started with "seed categories" [42] that were formed from the research question and objectives, based on the literature study on software testing and our prior observations [4, 5, 12–14] on software and testing processes. Some seed categories, like "knowledge management", "service-orientation" or "approach for software development" were derived from our earlier studies, while categories like "strategy for testing", "outsourcing", "customer impact" or "software testing tools" were taken from known issues or literature review observations.

The study followed the approach introduced by Seaman [43], which notes that the initial set of codes (seed categories) comes from the goals of the study, the research questions, and predefined variables of interest. In the open coding, we added new categories and merged existing categories to others if they seemed unfeasible or if we found a better generalization. Especially at the beginning of the analysis, the number of categories and codes quickly accumulated and the total number of codes after open coding amounted to 164 codes in 12 different categories. Besides the test process, software development in general and test automation, these categories also contained codified observations on such aspects as the business orientation, outsourcing, and product quality.

After collecting the individual observations to categories and codes, the categorized codes were linked together based on the relationships observed in the interviews. For example, the codes "Software process: Acquiring 3rd party modules", "Testing strategy: Testing 3rd party modules", and "Problem: Knowledge management with 3rd party modules" were clearly related and therefore we connected them together in axial coding. The objective of axial coding is to further develop categories, their properties and dimensions, and find causal, or any kinds of, connections between the categories and codes.

For some categories, the axial coding also makes it possible to define dimension for the phenomenon, for example "Personification-Codification" for "Knowledge management strategy", where every property could be defined as a point along the continuum defined by the two polar opposites. For the categories that are given dimension, the dimension represented the locations of the property or the attribute of a category [40]. Obviously for some categories, which were used to summarize different observations like enhancement proposals or process problems, defining dimensions was unfeasible. We considered using dimensions for some categories like "criticality of test automation in testing process" or "tool sophistication level for automation tools" in this study, but discarded them later as they yielded only little value to the study. This decision was based on the observation that values of both dimensions were outcomes of the applied test automation strategy, having no effect on the actual suitability or applicability of test automation to the organization's test process.

Our approach for analysis of the categories included Within-Case Analysis and Cross-Case-Analysis, as specified by Eisenhardt [37]. We used the tactic of selecting dimensions and properties with within-group similarities coupled with inter-group differences [37]. In this strategy, our team isolated one phenomenon that clearly divided the organizations to different groups, and searched for explaining differences and similarities from within these groups. Some of the applied features were, for example, the application of agile development methods, the application of test automation, the size [38] difference of originating companies and service orientation. As for one central result, the appropriateness of OU as a comparison unit was confirmed based on our size difference-related observations on the data; the within-group- and inter-group comparisons did yield results in which the company size or company policies did not have strong influence, whereas the local, within-unit policies did. In addition, the internal activities observed in OUs were similar regardless of the originating company size, meaning that in our study the OU comparison was indeed feasible approach.

We established and confirmed each chain of evidence in this interpretation method by discovering sufficient citations or finding conceptually similar OU activities from the case transcriptions. Finally, in the last phase of the analysis,

Table 3: Open coding of the interview data.

Interview transcript	Codes, Category: Code
"Well, I would hope for *stricter control or management for implementing our testing strategy*, as I am not sure if our *testing covers everything and is it sophisticated enough*. On the other hand, we do have *strictly limited resources, so it can be enhanced only to some degree*, we cannot test everything. And perhaps, recently we have had, in the newest versions, some regression testing, going through all features, seeing if nothing is broken, but *in several occasions this has been left unfinished because time has run out*. So there, on that issue we should focus."	Enhancement proposal: Developing testing strategy Strategy for testing: Ensuring case coverage Problem: Lack of resources Problem: Lack of time

in selective coding, our objective was to identify the core category [40]—a central phenomenon—and systematically relate it to other categories and generate the hypothesis and the theory. In this study, we consider test automation in practice as the core category, to which all other categories were related as explaining features of applicability or feasibility.

The general rule in grounded theory is to sample until theoretical saturation is reached. This means (1) no new or relevant data seem to emerge regarding a category, (2) the category development is dense, insofar as all of the paradigm elements are accounted for, along with variation and process, and (3) the relationships between categories are well established and validated [40]. In our study, the saturation was reached during the third round, where no new categories were created, merged, or removed from the coding. Similarly, the attribute values were also stable, that is, the already discovered phenomena began to repeat themselves in the collected data. As an additional way to ensure the validity of our study and avoid validity threats [44], four researchers took part in the data analysis. The bias caused by researchers was reduced by combining the different views of the researchers (observer triangulation) and a comparison with the phenomena observed in the quantitative data (methodological triangulation) [44, 45].

3.4. The Survey Instrument Development and Data Collection. The survey method described by Fink and Kosecoff [46] was used as the research method in the second round. An objective for a survey is to collect information from people about their feelings and beliefs. Surveys are most appropriate when information should come directly from the people [46]. Kitchenham et al. [47] divide comparable survey studies into exploratory studies from which only weak conclusions can be drawn, and confirmatory studies from which strong conclusions can be drawn. We consider this study as an exploratory, observational, and cross-sectional study that explores the phenomenon of software testing automation in practice and provides more understanding to both researchers and practitioners.

To obtain reliable measurements in the survey, a validated instrument was needed, but such an instrument was not available in the literature. However, Dybå [48] has developed an instrument for measuring the key factors of success in software process improvement. Our study was constructed based on the key factors of this instrument, and supplemented with components introduced in the standards ISO/IEC 29119 [11] and 25010 [49]. We had the possibility to use the current versions of the new standards because one of the authors is a member of the JTC1/SC7/WG26, which is developing the new software testing standard. Based on these experiences a measurement instrument derived from the ISO/IEC 29119 and 25010 standards was used.

The survey consisted of a questionnaire (available at http://www2.it.lut.fi/project/MASTO/) and a face-to-face interview. Selected open-ended questions were located at the end of the questionnaire to cover some aspects related to our qualitative study. The classification of the qualitative answers was planned in advance.

The questionnaire was planned to be answered during the interview to avoid missing answers because they make the data analysis complicated. All the interviews were tape-recorded, and for the focus organizations, further qualitatively analyzed with regard to the additional comments made during the interviews. Baruch [50] also states that the average response rate for self-assisted questionnaires is 55.6%, and when the survey involves top management or organizational representatives the response rate is 36.1%. In this case, a self-assisted, mailed questionnaire would have led to a small sample. For these reasons, it was rejected, and personal interviews were selected instead. The questionnaire was piloted with three OUs and four private persons.

If an OU had more than one respondent in the interview, they all filled the same questionnaire. Arranging the interviews, traveling and interviewing took two months of calendar time. Overall, we were able to accomplish 0.7 survey interviews per working day on an average. One researcher conducted 80% of the interviews, but because of the overlapping schedules also two other researchers participated in the interviews. Out of the contacted 42 OUs, 11 were rejected because they did not fit the population criteria in spite of the source information, or it was impossible to fit the interview into the interviewee's schedule. In a few individual cases, the reason for rejection was that the organization refused to give an interview. All in all, the response rate was, therefore, 74%.

4. Testing and Test Automation in Surveyed Organizations

4.1. General Information of the Organizational Units. The interviewed OUs were parts of large companies (55%) and small and medium-sized enterprises (45%). The OUs belonged to companies developing information systems (11 OUs), IT services (5 OUs), telecommunication (4 OUs),

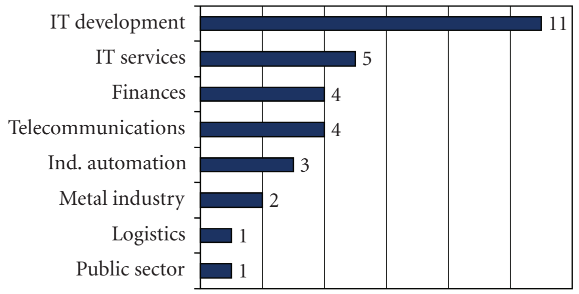


Figure 1: Application domains of the companies.

Table 4: Interviewed OUs.

	Max.	Min.	Median
Number of employees in the company.	350 000	4	315
Number of SW developers and testers in the OU.	600	0[1]	30
Percentage of automation in testing.	90	0	10
Percentage of agile (reactive, iterative) versus plan driven methods in projects.	100	0	30
Percentage of existing testers versus resources need.	100	10	75
How many percent of the development effort is spent on testing?	70	0[2]	25

[1]0 means that all of the OUs developers and testers are acquired from 3rd parties.
[2]0 means that no project time is allocated especially for testing.

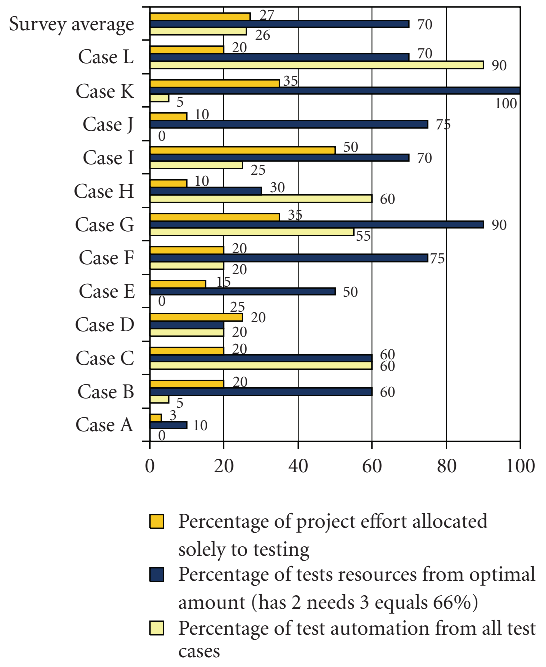


Figure 2: Amount of test resources and test automation in the focus organizations of the study and the survey average.

finance (4 OUs), automation systems (3 OUs), the metal industry (2 OUs), the public sector (1 OU), and logistics (1 OU). The application domains of the companies are presented in Figure 1. Software products represented 63% of the turnover, and services (e.g., consulting, subcontracting, and integration) 37%.

The maximum number of personnel in the companies to which the OUs belonged was 350 000, the minimum was four, and the median was 315. The median of the software developers and testers in the OUs was 30 persons. OUs applied automated testing less than expected, the median of the automation in testing being 10%. Also, the interviewed OUs utilized agile methods less than expected: the median of the percentage of agile (reactive, iterative) versus plan driven methods in projects was 30%. The situation with human resources was better than what was expected, as the interviewees estimated that the amount of human resources in testing was 75%. When asking what percent of the development effort was spent on testing, the median of the answers was 25%. The cross-sectional situation of development and testing in the interviewed OUs is illustrated in Table 4.

The amount of testing resources was measured by three figures; first the interviewee was asked to evaluate the percentage from total project effort allocated solely to testing. The survey average was 27%, the maximum being 70% and the minimum 0%, meaning that the organization relied solely on testing efforts carried out in parallel with development. The second figure was the amount of test resources compared to the organizational optimum. In this figure, if the company had two testers and required three, it would have translated as 66% of resources. Here the average was 70%; six organizations (19%) reported 100% resource availability. The third figure was the number of automated test cases compared to all of the test cases in all of the test phases the software goes through before its release. The average was 26%, varying between different types of organizations and project types. The results are presented in Figure 2, in which the qualitative study case OUs are also presented for comparison. The detailed descriptions for each case organization are available in the appendix.

4.2. General Testing Items. The survey interviewed 31 organization managers from different types of software industry. The contributions of the interviewees were measured using a five-point Likert scale where 1 denoted "I fully disagree" and 5 denoted "I fully agree". The interviewees emphasized that quality is built in development (4.3) rather than in testing (2.9). Then the interviewees were asked to estimate their organizational testing practices according to the new testing standard ISO/IEC 29119 [11], which identifies four main levels for testing processes: the test policy, test strategy, test management and testing. The test policy is the company level guideline which defines the management, framework

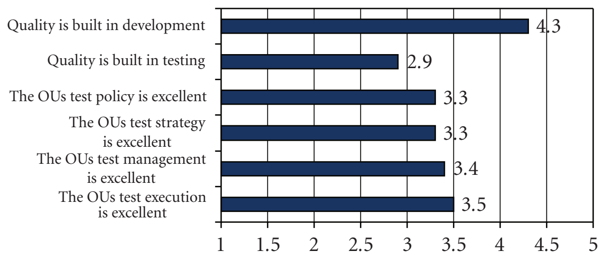


Figure 3: Levels of testing according to the ISO/IEC 29119 standard.

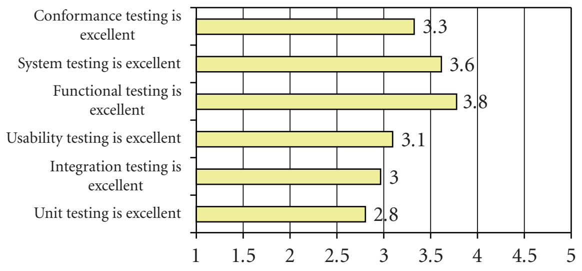


Figure 4: Testing phases in the software process.

and general guidelines, the test strategy is an adaptive model for the preferred test process, test management is the control level for testing in a software project, and finally, testing is the process of conducting test cases. The results did not make a real difference between the lower levels of testing (test management level and test levels) and higher levels of testing (organizational test policy and organizational test strategy). All in all, the interviewees were rather satisfied with the current organization of testing. The resulted average levels from quantitative survey are presented in Figure 3.

Besides the organization, the test processes and test phases were also surveyed. The five-point Likert scale with the same one to five—one being fully disagree and five fully agree—grading method was used to determine the correctness of different testing phases. Overall, the latter test phases—system, functional testing—were considered excellent or very good, whereas the low level test phases such as unit testing and integration received several low-end scores. The organizations were satisfied or indifferent towards all test phases, meaning that there were no strong focus areas for test organization development. However, based on these results it seems plausible that one effective way to enhance testing would be to support low-level testing in unit and integration test phases. The results are depicted in Figure 4.

Finally, the organizations surveyed were asked to rate their testing outcomes and objectives (Figure 5). The first three items discussed the test processes of a typical software project. There seems to be a strong variance in testing schedules and time allocation in the organizations. The outcomes 3.2 for schedule and 3.0 for time allocation do not give any information by themselves, and overall, the direction of answers varied greatly between "Fully disagree" and "Fully agree". However, the situation with test processes

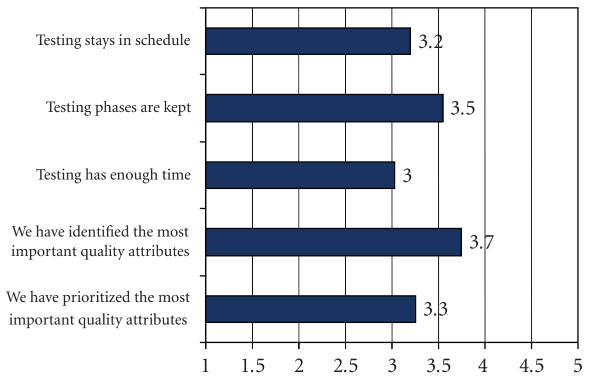


Figure 5: Testing process outcomes.

was somewhat better; the result 3.5 may also not be a strong indicator by itself, but the answers had only little variance, 20 OUs answering "somewhat agree" or "neutral". This indicates that even if the time is limited and the project schedule restricts testing, the testing generally goes through the normal, defined, procedures.

The fourth and fifth items were related to quality aspects, and gave insights into the clarity of testing objectives. The results of 3.7 for the identification of quality attributes indicate that organizations tend to have objectives for the test processes and apply quality criteria in development. However, the prioritization of their quality attributes is not as strong (3.3) as identification.

4.3. Testing Environment. The quality aspects were also reflected in the employment of systematic methods for the testing work. The majority (61%) of the OUs followed a systematic method or process in the software testing, 13% followed one partially, and 26% of the OUs did not apply any systematic method or process in testing. Process practices were derived from, for example, TPI (Test Process Improvement) [51] or the Rational Unified Process (RUP) [52]. Few Agile development process methods such as Scrum [53] or XP (eXtreme Programming) [54] were also mentioned.

A systematic method is used to steer the software project, but from the viewpoint of testing, the process also needs an infrastructure on which to operate. Therefore, the OUs were asked to report which kind of testing tools they apply to their typical software processes. The test management tools, tools which are used to control and manage test cases and allocate testing resources to cases, turned out to be the most popular category of tools; 15 OUs out of 31 reported the use of this type of tool. The second in popularity were manual unit testing tools (12 OUs), which were used to execute test cases and collect test results. Following them were tools to implement test automation, which were in use in 9 OUs, performance testing tools used in 8 OUs, bug reporting tools in 7 OUs and test design tools in 7 OUs. Test design tools were used to create and design new test cases. The group of other tools consisted of, for example, electronic measurement devices, test report generators, code analyzers, and project

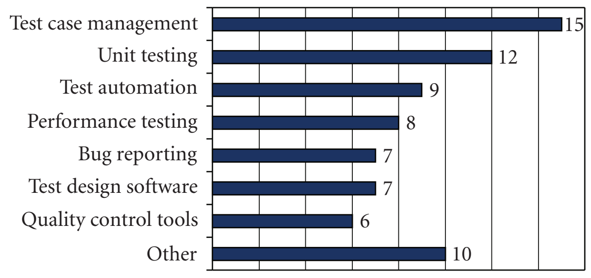


Figure 6: Popularity of the testing tools according to the survey.

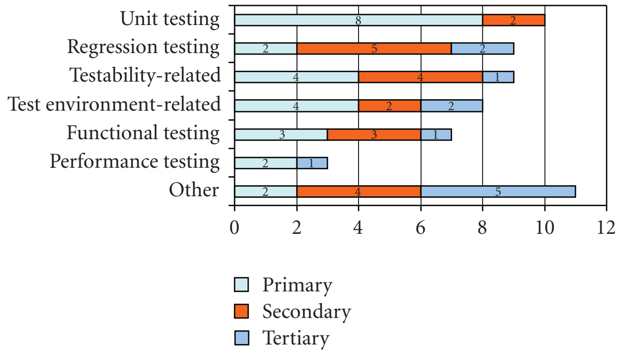


Figure 7: The three most efficient application areas of test automation tools according to the interviewees.

management tools. The popularity of the testing tools in different survey organizations is illustrated in Figure 6.

The respondents were also asked to name and explain the three most efficient application areas of test automation tools. Both the format of the open-ended questions and the classification of the answers were based on the like best (LB) technique adopted from Fink and Kosecoff [46]. According to the LB technique, respondents were asked to list points they considered the most efficient. The primary selection was the area in which the test automation would be the most beneficial to the test organization, the secondary one is the second best area of application, and the third one is the third best area. The interviewees were also allowed to name only one or two areas if they were unable to decide on three application areas. The results revealed the relative importance of software testing tools and methods.

The results are presented in Figure 7. The answers were distributed rather evenly between different categories of tools or methods. The most popular category was unit testing tools or methods (10 interviewees). Next in line were regression testing (9), tools to support testability (9), test environment tools and methods (8), and functional testing (7). The group "others" (11) consisted of conformance testing tools, TTCN-3 (*Testing and Test Control Notation version 3*) tools, general test management tools such as document generators and methods of unit and integration testing. The most popular category, unit testing tools or methods, also received the most primary application area nominations. The most common secondary area of application was regression testing. Several categories ranked third, but concepts such as regression testing, and test environment-related aspects such as document generators were mentioned more than once. Also testability-related concepts—module interface, conformance testing—or functional testing—verification, validation tests—were considered feasible implementation areas for test automation.

4.4. Summary of the Survey Findings. The survey suggests that interviewees were rather satisfied with their test policy, test strategy, test management, and testing, and did not have any immediate requirements for revising certain test phases, although low-level testing was slightly favoured in the development needs. All in all, 61% of the software companies followed some form of a systematic process or method in testing, with an additional 13% using some established procedures or measurements to follow the process efficiency. The systematic process was also reflected in the general approach to testing; even if the time was limited, the test process followed a certain path, applying the test phases regardless of the project limitations.

The main source of the software quality was considered to be in the development process. In the survey, the test organizations used test automation on an average on 26% of their test cases, which was considerably less than could be expected based on the literature. However, test automation tools were the third most common category of test-related tools, commonly intended to implement unit and regression testing. As for the test automation itself, the interviewees ranked unit testing tools as the most efficient tools of test automation, regression testing being the most common secondary area of application.

5. Test Automation Interviews and Qualitative Study

Besides the survey, the test automation concepts and applications were analyzed based on the interviews with the focus organizations. The grounded theory approach was applied to establish an understanding of the test automation concepts and areas of application for test automation in industrial software engineering. The qualitative approach was applied in three rounds, in which a developer, test manager and tester from 12 different case OUs were interviewed. Descriptions of the case OUs can be found in the appendix.

In theory-creating inductive research [55], the central idea is that researchers constantly compare theory and data iterating with a theory which closely fits the data. Based on the grounded theory codification, the categories identified were selected in the analysis based on their ability to differentiate the case organizations and their potential to explain the differences regarding the application of test automation in different contexts. We selected the categories so as to explore the types of automation applications and the compatibility of test automation services with the OUs testing organization. We conceptualized the most common test automation concepts based on the coding and further elaborated them to categories to either cater the essential features such as their role in the overall software process or their relation to test automation. We also concentrated on the OU differences in essential concepts such as automation tools, implementation issues or development strategies. This conceptualization resulted to the categories listed in Table 5.

The category "Automation application" describes the areas of software development, where test automation was applied successfully. This category describes the testing activities or phases which apply test automation processes. In the case where the test organization did not apply automation, or had so far only tested it for future applications, this category was left empty. The application areas were generally geared towards regression and stress testing, with few applications of functionality and smoke tests in use.

The category "Role in software process" is related to the objective for which test automation was applied in software development. The role in the software process describes the objective for the existence of the test automation infrastructure; it could, for example, be in quality control, where automation is used to secure module interfaces, or in quality assurance, where the operation of product functionalities is verified. The usual role for the test automation tools was in quality control and assurance, the level of application varying from third party-produced modules to primary quality assurance operations. On two occasions, the role of test automation was considered harmful to the overall testing outcomes, and on one occasion, the test automation was considered trivial, with no real return on investments compared to traditional manual testing.

The category "Test automation strategy" is the approach to how automated testing is applied in the typical software processes, that is, the way the automation was used as a part of the testing work, and how the test cases and overall test automation strategy were applied in the organization. The level of commitment to applying automation was the main dimension of this category, the lowest level being individual users with sporadic application in the software projects, and the highest being the application of automation to the normal, everyday testing infrastructure, where test automation was used seamlessly with other testing methods and had specifically assigned test cases and organizational support.

The category of "Automation development" is the general category for OU test automation development. This category summarizes the ongoing or recent efforts and resource allocations to the automation infrastructure. The type of new development, introduction strategies and current development towards test automation are summarized in this category. The most frequently chosen code was "general increase of application", where the organization had committed itself to test automation, but had no clear idea of how to develop the automation infrastructure. However, one OU had a development plan for creating a GUI testing environment, while two organizations had just recently scaled down the amount of automation as a result of a pilot project. Two organizations had only recently introduced test automation to their testing infrastructure.

The category of "Automation tools" describes the types of test automation tools that are in everyday use in the OU. These tools are divided based on their technological finesse, varying from self-created drivers and stubs to individual proof-of-concept tools with one specified task to test suites where several integrated components are used together for an effective test automation environment. If the organization had created the tools by themselves, or customized the acquired tools to the point of having new features and functionalities, the category was supplemented with a notification regarding in-house-development.

Finally, the category of "Automation issues" includes the main hindrances which are faced in test automation within the organization. Usually, the given issue was related to either the costs of test automation or the complexity of introducing automation to the software projects which have been initially developed without regards to support for automation. Some organizations also considered the efficiency of test automation to be the main issue, mostly contributing to the fact that two of them had just recently scaled down their automation infrastructure. A complete list of test automation categories and case organizations is given in Table 6.

We elaborated further these properties we observed from the case organizations to create hypotheses for the test automation applicability and availability. These resulting hypotheses were shaped according to advice given by Eisenhardt [37] for qualitative case studies. For example, we perceived the quality aspect as really important for the role of automation in software process. Similarly, the resource needs, especially costs, were much emphasized in the automation issues category. The purpose of the hypotheses below is to summarize and explain the features of test automation that resulted from the comparison of differences and similarities between the organizations.

Hypothesis 1 (Test automation should be considered more as a quality control tool rather than a frontline testing method). The most common area of application observed was functionality verification, that is, regression testing and GUI event testing. As automation is time-consuming and expensive to create, these were the obvious ways to create test cases which had the minimal number of changes per development cycle. By applying this strategy, organizations could set test automation to confirm functional properties with suitable test cases, and acquire such benefits as support for change management and avoid unforeseen compatibility issues with module interfaces.

> *"Yes, regression testing, especially automated. It is not manually "hammered in" every time, but used so that the test sets are run, and if there is anything abnormal, it is then investigated."*—Manager, Case G

> *"... had we not used it [automation tests], it would have been suicidal."*—Designer, Case D

> *"It's [automated stress tests] good for showing bad code, how efficient it is and how well designed ... stress it enough and we can see if it slows down or even breaks completely."*—Tester, Case E

Table 5: Test automation categories.

Category	Definition
Automation application	Areas of application for test automation in the software process
Role in software process	The observed roles of test automation in the company software process and the effect of this role
Test automation strategy	The observed method for selecting the test cases where automation is applied and the level of commitment to the application of test automation in the organizations
Automation development	The areas of active development in which the OU is introducing test automation
Automation tools	The general types of test automation tools applied
Automation issues	The items that hinder test automation development in the OU

However, there seemed to be some contradicting considerations regarding the applicability of test automation. Cases F, J, and K had recently either scaled down their test automation architecture or considered it too expensive or inefficient when compared to manual testing. In some cases, automation was also considered too bothersome to configure for a short-term project, as the system would have required constant upkeep, which was an unnecessary addition to the project workload.

> *"We really have not been able to identify any major advancements from it [test automation]."*—Tester, Case J

> *"It [test automation] just kept interfering."*—Designer, Case K

Both these viewpoints indicated that test automation should not be considered a "frontline" test environment for finding errors, but rather a quality control tool to maintain functionalities. For unique cases or small projects, test automation is too expensive to develop and maintain, and it generally does not support single test cases or explorative testing. However, it seems to be practical in larger projects, where verifying module compatibility or offering legacy support is a major issue.

Hypothesis 2 (Maintenance and development costs are common test automation hindrances that universally affect all test organizations regardless of their business domain or company size). Even though the case organizations were selected to represent different types of organizations, the common theme was that the main obstacles in automation adoption were development expenses and upkeep costs. It seemed to make no difference whether the organization unit belonged to a small or large company, as in the OU levels they shared common obstacles. Even despite the maintenance and development hindrances, automation was considered a feasible tool in many organizations. For example, Cases I and L pursued the development of some kind of automation to enhance the testing process. Similarly, Cases E and H, which already had a significant number of test automation cases, were actively pursuing a larger role for automated testing.

> *"Well, it [automation] creates a sense of security and controllability, and one thing that is easily underestimated is its effect on performance and optimization. It requires regression tests to confirm that if something is changed, the whole thing does not break down afterwards."*—Designer, Case H

In many cases, the major obstacle for adopting test automation was, in fact, the high requirements for process development resources.

> *"Shortage of time, resources … we have the technical ability to use test automation, but we don't."* —Tester, Case J

> *"Creating and adopting it, all that it takes to make usable automation … I believe that we don't put any effort into it because it will end up being really expensive."* —Designer, Case J

In Case J particularly, the OU saw no incentive in developing test automation as it was considered to offer only little value over manual testing, even if they otherwise had no immediate obstacles other than implementation costs. Also cases F and K reported similar opinions, as they both had scaled down the amount of automation after the initial pilot projects.

> *"It was a huge effort to manually confirm why the results were different, so we took it [automation] down."*—Tester, Case F

> *"Well, we had gotten automation tools from our partner, but they were so slow we decided to go on with manual testing."*—Tester, Case K

Hypothesis 3 (Test automation is applicable to most of the software processes, but requires considerable effort from the organization unit). The case organizations were selected to represent the polar types of software production operating in different business domains. Out of the focus OUs, there were four software development OUs, five IT service OUs, two OUs from the finance sector and one logistics OU. Of these OUs, only two did not have any test automation, and two others had decided to strategically abandon their test automation infrastructure. Still, the business domains for the remaining organizations which applied test automation were

Table 6: Test automation categories affecting the software process in case OUs.

OU	Category					
	Automation application	Role in software process	Test automation strategy	Automation development	Automation tools	Automation issues
Case A	GUI testing, regression testing	Functionality verification	Part of the normal test infrastructure	General increase of application	Individual tools, test suite, in-house development	Complexity of adapting automation to test processes
Case B	Performance, smoke testing	Quality control tool	Part of the normal test infrastructure	GUI testing, unit testing	Individual tools, in-house development	Costs of automation implementation
Case C	Functionality, regression testing, documentation automation	Quality control tool	Part of the normal test infrastructure	General increase of application	Test suite, in-house development	Cost of automation maintenance
Case D	Functionality testing	Quality control for secondary modules	Project-related cases	Upkeep for existing parts	Individual tools	Costs of automation implementation
Case E	System stress testing	Quality assurance tool	Part of the normal test infrastructure	General increase of application	Test suite	Costs of implementing new automation
Case F	Unit and module testing, documentation automation	QC, overall effect harmful	Individual users	Recently scaled down	Individual tools	Manual testing seen more efficient
Case G	Regression testing for use cases	Quality assurance tool	Part of the normal test infrastructure	General increase of application	Test suite	Cost of automation maintenance
Case H	Regression testing for module interfaces	Quality control for secondary modules	Part of the normal test infrastructure	General increase of application	Test suite, in-house development	Underestimation of the effect of automated testing on quality
Case I	Functionality testing	Quality control tool	Project-related cases	Application pilot in development	Proof-of-concept tools	Costs of automation implementation
Case J	Automation not in use	QA, no effect observed	Individual users	Application pilot in development	Proof-of-concept tools	No development incentive
Case K	Small scale system testing	QC, overall effect harmful	Individual users	Recently scaled down	Self-created tools; drivers and stubs	Manual testing seen more efficient
Case L	System stress testing	Verifies module compatibility	Project-related cases	Adapting automation to the testing strategy	Individual tools, in-house development	Complexity of adapting automation to test processes

heterogeneously divided, meaning that the business domain is not a strong indicator of whether or not test automation should be applied.

It seems that test automation is applicable as a test tool in any software process, but the amount of resources required for useful automation compared to the overall development resources is what determines whether or not automation should be used. As automation is oriented towards quality control aspects, it may be unfeasible to implement in small development projects where quality control is manageable with manual confirmation. This is plausible, as the amount of required resources does not seem to vary based on aspects beyond the OU characteristics, such as available company resources or testing policies applied. The feasibility of test automation seems to be rather connected to the actual software process objectives, and fundamentally to the decision whether the quality control aspects gained from test automation supersede the manual effort required for similar results.

"... before anything is automated, we should calculate the maintenance effort and estimate

> *whether we will really save time, instead of just automating for automation's sake."*—Tester, Case G

> *"It always takes a huge amount of resources to implement."*—Designer, Case A

> *"Yes, developing that kind of test automation system is almost as huge an effort as building the actual project."* —Designer, Case I

Hypothesis 4 (The available repertoire of testing automation tools is limited, forcing OUs to develop the tools themselves, which subsequently contributes to the application and maintenance costs). There were only a few case OUs that mentioned any commercial or publicly available test automation programs or suites. The most common approach to test automation tools was to first acquire some sort of tool for proof-of-concept piloting, then develop similar tools as in-house-production or extend the functionalities beyond the original tool with the OU's own resources. These resources for in-house-development and upkeep for self-made products are one of the components that contribute to the costs of applying and maintaining test automation.

> *"Yes, yes. That sort of [automation] tools have been used, and then there's a lot of work that we do ourselves. For example, this stress test tool ..."*—Designer, Case E

> *"We have this 3rd party library for the automation. Well, actually, we have created our own architecture on top of it ..."*—Designer, Case H

> *"Well, in [company name], we've-, we developed our own framework to, to try and get around some of these, picking which tests, which group of tests should be automated."*—Designer, Case C

However, it should be noted that even if the automation tools were well-suited for the automation tasks, the maintenance still required significant resources if the software product to which it was connected was developing rapidly.

> *"Well, there is the problem [with automation tool] that sometimes the upkeep takes an incredibly large amount of time."* —Tester, Case G

> *"Our system keeps constantly evolving, so you'd have to be constantly recording [maintaining tools]..."* —Tester, Case K

6. Discussion

An exploratory survey combined with interviews was used as the research method. The objective of this study was to shed light on the status of test automation and to identify improvement needs in and the practice of test automation. The survey revealed that the total effort spent on testing (median 25%) was less than expected. The median percentage (25%) of testing is smaller than the 50%–60% that is often mentioned in the literature [38, 39]. The comparable low percentage may indicate that that the resources needed for software testing are still underestimated even though testing efficiency has grown. The survey also indicated that companies used fewer resources on test automation than expected: on an average 26% of all of the test cases apply automation. However, there seems to be ambiguity as to which activities organizations consider test automation, and how automation should be applied in the test organizations. In the survey, several organizations reported that they have an extensive test automation infrastructure, but this did not reflect on the practical level, as in the interviews with testers particularly, the figures were considerably different. This indicates that the test automation does not have strong strategy in the organization, and has yet to reach maturity in several test organizations. Such concepts as quality assurance testing and stress testing seem to be particularly unambiguous application areas, as Cases E and L demonstrated. In Case E, the management did not consider stress testing an automation application, whereas testers did. Moreover, in Case L the large automation infrastructure did not reflect on the individual project level, meaning that the automation strategy may strongly vary between different projects and products even within one organization unit.

The qualitative study which was based on interviews indicated that some organizations, in fact, actively avoid using test automation, as it is considered to be expensive and to offer only little value for the investment. However, test automation seems to be generally applicable to the software process, but for small projects the investment is obviously oversized. One additional aspect that increases the investment are tools, which unlike in other areas of software testing, tend to be developed in-house or are heavily modified to suit specific automation needs. This development went beyond the localization process which every new software tool requires, extending even to the development of new features and operating frameworks. In this context it also seems plausible that test automation can be created for several different test activities. Regression testing, GUI testing or unit testing, activities which in some form exist in most development projects, all make it possible to create successful automation by creating suitable tools for the task, as in each phase can be found elements that have sufficient stability or unchangeability. Therefore it seems that the decision on applying automation is not only connected to the enablers and disablers of test automation [4], but rather on tradeoff of required effort and acquired benefits; In small projects or with low amount of reuse the effort becomes too much for such investment as applying automation to be feasible.

The investment size and requirements of the effort can also be observed on two other occasions. First, test automation should not be considered as an active testing tool for finding errors, but as a tool to guarantee the functionality of already existing systems. This observation is in line with those of Ramler and Wolfmaier [3], who discuss the necessity of a large number of repetitive tasks for the automation to supersede manual testing in cost-effectiveness, and of

Berner et al. [8], who notify that the automation requires a sound application plan and well-documented, simulatable and testable objects. For both of these requirements, quality control at module interfaces and quality assurance on system operability are ideal, and as it seems, they are the most commonly used application areas for test automation. In fact, Kaner [56] states that 60%–80% of the errors found with test automation are found in the development phase for the test cases, further supporting the quality control aspect over error discovery.

Other phenomena that increase the investment are the limited availability and applicability of automation tools. On several occasions, the development of the automation tools was an additional task for the automation-building organization that required the organization to allocate their limited resources to the test automation tool implementation. From this viewpoint it is easy to understand why some case organizations thought that manual testing is sufficient and even more efficient when measured in resource allocation per test case. Another approach which could explain the observed resistance to applying or using test automation was also discussed in detail by Berner et al. [8], who stated that organizations tend to have inappropriate strategies and overly ambitious objectives for test automation development, leading to results that do not live up to their expectations, causing the introduction of automation to fail. Based on the observations regarding the development plans beyond piloting, it can also be argued that the lack of objectives and strategy also affect the successful introduction processes. Similar observations of "automation pitfalls" were also discussed by Persson and Yilmaztürk [26] and Mosley and Posey [57].

Overall, it seems that the main disadvantages of testing automation are the costs, which include implementation costs, maintenance costs, and training costs. Implementation costs included direct investment costs, time, and human resources. The correlation between these test automation costs and the effectiveness of the infrastructure are discussed by Fewster [24]. If the maintenance of testing automation is ignored, updating an entire automated test suite can cost as much, or even more than the cost of performing all the tests manually, making automation a bad investment for the organization. We observed this phenomenon in two case organizations. There is also a connection between implementation costs and maintenance costs [24]. If the testing automation system is designed with the minimization of maintenance costs in mind, the implementation costs increase, and vice versa. We noticed the phenomenon of costs preventing test automation development in six cases. The implementation of test automation seems to be possible to accomplish with two different approaches: by promoting either maintainability or easy implementation. If the selected focus is on maintainability, test automation is expensive, but if the approach promotes easy implementation, the process of adopting testing automation has a larger possibility for failure. This may well be due to the higher expectations and assumption that the automation could yield results faster when promoting implementation over maintainability, often leading to one of the automation pitfalls [26] or at least a low percentage of reusable automation components with high maintenance costs.

7. Conclusions

The objective of this study was to observe and identify factors that affect the state of testing, with automation as the central aspect, in different types of organizations. Our study included a survey in 31 organizations and a qualitative study in 12 focus organizations. We interviewed employees from different organizational positions in each of the cases.

This study included follow-up research on prior observations [4, 5, 12–14] on testing process difficulties and enhancement proposals, and on our observations on industrial test automation [4]. In this study we further elaborated on the test automation phenomena with a larger sample of polar type OUs, and more focused approach on acquiring knowledge on test process-related subjects. The survey revealed that test organizations use test automation only in 26% of their test cases, which was considerably less than could be expected based on the literature. However, test automation tools were the third most common category of test-related tools, commonly intended to implement unit and regression testing. The results indicate that adopting test automation in software organization is a demanding effort. The lack of existing software repertoire, unclear objectives for overall development and demands for resource allocation both for design and upkeep create a large threshold to overcome.

Test automation was most commonly used for quality control and quality assurance. In fact, test automation was observed to be best suited to such tasks, where the purpose was to secure working features, such as check module interfaces for backwards compatibility. However, the high implementation and maintenance requirements were considered the most important issues hindering test automation development, limiting the application of test automation in most OUs. Furthermore, the limited availability of test automation tools and the level of commitment required to develop a suitable automation infrastructure caused additional expenses. Due to the high maintenance requirements and low return on investments in small-scale application, some organizations had actually discarded their automation systems or decided not to implement test automation. The lack of a common strategy for applying automation was also evident in many interviewed OUs. Automation applications varied even within the organization, as was observable in the differences when comparing results from different stakeholders. In addition, the development strategies were vague and lacked actual objectives. These observations can also indicate communication gaps [58] between stakeholders of the overall testing strategy, especially between developers and testers.

The data also suggested that the OUs that had successfully implemented test automation infrastructure to cover the entire organization seemed to have difficulties in creating a continuance plan for their test automation development. After the adoption phases were over, there was an ambiguity about how to continue, even if the organization had decided to further develop their test automation infrastructure. The overall objectives were usually clear and obvious—cost savings and better test coverage—but in practise there were only few actual development ideas and novel concepts. In the case organizations this was observed in the vagueness of the development plans: only one of the five OUs which used automation as a part of their normal test processes had development plans beyond the general will to increase the application.

The survey established that 61% of the software companies followed some form of a systematic process or method in testing, with an additional 13% using some established procedures or measurements to follow the process efficiency. The main source of software quality was considered to reside in the development process, with testing having much smaller impact in the product outcome. In retrospect of the test levels introduced in the ISO/IEC29119 standard, there seems to be no one particular level of the testing which should be the research and development interest for best result enhancements. However, the results from the self-assessment of the test phases indicate that low-level testing could have more potential for testing process development.

Based on these notions, the research and development should focus on uniform test process enhancements, such as applying a new testing approach and creating an organization-wide strategy for test automation. Another focus area should be the development of better tools to support test organizations and test processes in the low-level test phases such as unit or integration testing. As for automation, one tool project could be the development of a customizable test environment with a common core and with an objective to introduce less resource-intensive, transferable and customizable test cases for regression and module testing.

Appendix

Case Descriptions

Case A (Manufacturing execution system (MES) producer and electronics manufacturer). Case A produces software as a service (SaaS) for their product. The company is a small-sized, nationally operating company that has mainly industrial customers. Their software process is a plan-driven cyclic process, where the testing is embedded to the development itself, having only little amount of dedicated resources. This organization unit applied test automation as a user interface and regression testing tool, using it for product quality control. Test automation was seen as a part of the normal test strategy, universally used in all software projects. The development plan for automation was to generally increase the application, although the complexity of the software- and module architecture was considered major obstacle on the automation process.

Case B (Internet service developer and consultant). Case B organization offers two types of services; development of Internet service portals for the customers like communities and public sector, and consultation in the Internet service business domain. The origination company is small and operates on a national level. Their main resource on the test automation is in the performance testing as a quality control tool, although addition of GUI test automation has also been proposed. The automated tests are part of the normal test process, and the overall development plan was to increase the automation levels especially to the GUI test cases. However, this development has been hindered by the cost of designing and developing test automation architecture.

Case C (Logistics software developer). Case C organization focuses on creating software and services for their origin company and its customers. This organization unit is a part of a large-sized, nationally operating company with large, highly distributed network and several clients. The test automation is widely used in several testing phases like functionality testing, regression testing and document generation automation. These investments are used for quality control to ensure the software usability and correctness. Although the OU is still aiming for larger test automation infrastructure, the large amount of related systems and constant changes within the inter-module communications is causing difficulties in development and maintenance of the new automation cases.

Case D (ICT consultant). Case D organization is a small, regional software consultant company, whose customers mainly compose of small business companies and the public sector. Their organization does some software development projects, in which the company develops services and ICT products for their customers. The test automation comes mainly trough this channel, as the test automation is mainly used as a conformation test tool for the third party modules. This also restricts the amount of test automation to the projects, in which these modules are used. The company currently does not have development plans for the test automation as it is considered unfeasible investment for the OU this size, but they do invest on the upkeep of the existing tools as they have usage as a quality control tool for the acquired outsider modules.

Case E (Safety and logistics system developer). Case E organization is a software system developer for safety and logistics systems. Their products have high amount of safety critical features and have several interfaces on which to communicate with. The test automation is used as a major quality assurance component, as the service stress tests are automated to a large degree. Therefore the test automation is also a central part of the testing strategy, and each project has defined set of automation cases. The organization is aiming to increase the amount of test automation and simultaneously develop new test cases and automation applications for the testing process. The main obstacle for this development has so far been the costs of creating new automation tools and extending the existing automation application areas.

Case F (Naval software system developer). The Case F organization unit is responsible for developing and testing naval service software systems. Their product is based on a common core, and has considerable requirements for compatibility with the legacy systems. This OU has tried test automation on several cases with application areas such as unit- and module testing, but has recently scaled down test automation for only support aspects such as the documentation automation. This decision was based on the resource requirements for developing and especially maintaining the automation system, and because the manual testing was in this context considered much more efficient as there were too much ambiguity in the automation-based test results.

Case G (Financial software developer). Case G is a part of a large financial organization, which operates nationally but has several internationally connected services due to their business domain. Their software projects are always aimed as a service portal for their own products, and have to pass considerable verification and validation tests before being introduced to the public. Because of this, the case organization has sizable test department when compared to other case companies in this study, and follows rigorous test process plan in all of their projects. The test automation is used in the regression tests as a quality assurance tool for user interfaces and interface events, and therefore embedded to the testing strategy as a normal testing environment. The development plans for the test automation is aimed to generally increase the amount of test cases, but even the existing test automation infrastructure is considered expensive to upkeep and maintain.

Case H (Manufacturing execution system (MES) producer and logistics service system provider). Case H organization is a medium-sized company, whose software development is a component for the company product. Case organization products are used in logistics service systems, usually working as a part of automated processes. The case organization applies automated testing as a module interface testing tool, applying it as a quality control tool in the test strategy. The test automation infrastructure relies on the in-house-developed testing suite, which enables organization to use the test automation to run daily tests to validate module conformance. Their approach on the test automation has been seen as a positive enabler, and the general trend is towards increasing automation cases. The main test automation disability is considered to be that the quality control aspect is not visible when working correctly and therefore the effect of test automation may be underestimated in the wider organization.

Case I (Small and medium-sized enterprise (SME) business and agriculture ICT-service provider). The case I organization is a small, nationally operating software company which operates on multiple business domain. Their customer base is heterogeneous, varying from finances to the agriculture and government services. The company is currently not utilizing test automation in their test process, but they have development plans for designing quality control automation. For this development they have had some individual proof-of-concept tools, but currently the overall testing resources limit the application process.

Case J (Modeling software developer). Case J organization develops software products for civil engineering and architectural design. Their software process is largely plan-driven with rigorous verification and validation processes in the latter parts of an individual project. Even though the case organization itself has not implemented test automation, on the corporate level there are some pilot projects where regression tests have been automated. These proof-of-concept-tools have been introduced to the case OU and there are intentions to apply them in the future, but there has so far been no incentive for adoption of the automation tools, delaying the application process.

Case K (ICT developer and consultant). Case K organization is a large, international software company which offers software products for several business domains and government services. Case organization has previously piloted test automation, but decided against adopting the system as it was considered too expensive and resource-intensive to maintain when compared to the manual testing. However, some of these tools still exist, used by individual developers along with test drivers and interface studs in unit- and regression testing.

Case L (Financial software developer). Case L organization is a large software provider for their corporate customer which operates on the finance sector. Their current approach on software process is plan-driven, although some automation features has been tested on a few secondary processes. The case organization does not apply test automation as is, although some module stress test cases have been automated as pilot tests. The development plan for test automation is to generally implement test automation as a part of their testing strategy, although amount of variability and interaction in the module interfaces is considered difficult to implement in test automation cases.

Acknowledgment

This study is a part of the ESPA project (http://www.soberit.hut.fi/espa/), funded by the Finnish Funding Agency for Technology and Innovation (project number 40125/08) and by the participating companies listed on the project web site.

References

[1] E. Kit, *Software Testing in the Real World: Improving the Process*, Addison-Wesley, Reading, Mass, USA, 1995.

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," RTI Project 7007.011, U.S. National Institute of Standards and Technology, Gaithersburg, Md, USA, 2002.

[3] R. Ramler and K. Wolfmaier, "Observations and lessons learned from automated testing," in *Proceedings of the International Workshop on Automation of Software Testing (AST '06)*, pp. 85–91, Shanghai, China, May 2006.

[4] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST '09)*, pp. 201–209, Denver, Colo, USA, April 2009.

[5] O. Taipale and K. Smolander, "Improving software testing by observing causes, effects, and associations from practice," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE '06)*, Rio de Janeiro, Brazil, September 2006.

[6] B. Shea, "Sofware testing gets new respect," *InformationWeek*, July 2000.

[7] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, Boston, Mass, USA, 1999.

[8] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 571–579, St. Louis, Mo, USA, May 2005.

[9] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70–79, 2000.

[10] L. J. Osterweil, "Software processes are software too, revisited: an invited talk on the most influential paper of ICSE 9," in *Proceedings of the 19th IEEE International Conference on Software Engineering*, pp. 540–548, Boston, Mass, USA, May 1997.

[11] ISO/IEC and ISO/IEC 29119-2, "Software Testing Standard—Activity Descriptions for Test Process Diagram," 2008.

[12] O. Taipale, K. Smolander, and H. Kälviäinen, "Cost reduction and quality improvement in software testing," in *Proceedings of the 14th International Software Quality Management Conference (SQM '06)*, Southampton, UK, April 2006.

[13] O. Taipale, K. Smolander, and H. Kälviäinen, "Factors affecting software testing time schedule," in *Proceedings of the Australian Software Engineering Conference (ASWEC '06)*, pp. 283–291, Sydney, Australia, April 2006.

[14] O. Taipale, K. Smolander, and H. Kälviäinen, "A survey on software testing," in *Proceedings of the 6th International SPICE Conference on Software Process Improvement and Capability dEtermination (SPICE '06)*, Luxembourg, May 2006.

[15] N. C. Dalkey, *The Delphi Method: An Experimental Study of Group Opinion*, RAND, Santa Monica, Calif, USA, 1969.

[16] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen, "A preliminary survey on software testing practices in Australia," in *Proceedings of the Australian Software Engineering Conference (ASWEC '04)*, pp. 116–125, Melbourne, Australia, April 2004.

[17] R. Torkar and S. Mankefors, "A survey on testing and reuse," in *Proceedings of IEEE International Conference on Software—Science, Technology and Engineering (SwSTE '03)*, Herzlia, Israel, November 2003.

[18] C. Ferreira and J. Cohen, "Agile systems development and stakeholder satisfaction: a South African empirical study," in *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '08)*, pp. 48–55, Wilderness, South Africa, October 2008.

[19] J. Li, F. O. Bjørnson, R. Conradi, and V. B. Kampenes, "An empirical study of variations in COTS-based software development processes in the Norwegian IT industry," *Empirical Software Engineering*, vol. 11, no. 3, pp. 433–461, 2006.

[20] W. Chen, J. Li, J. Ma, R. Conradi, J. Ji, and C. Liu, "An empirical study on software development with open source components in the Chinese software industry," *Software Process Improvement and Practice*, vol. 13, no. 1, pp. 89–100, 2008.

[21] R. Dossani and N. Denny, "The Internet's role in offshored services: a case study of India," *ACM Transactions on Internet Technology*, vol. 7, no. 3, 2007.

[22] K. Y. Wong, "An exploratory study on knowledge management adoption in the Malaysian industry," *International Journal of Business Information Systems*, vol. 3, no. 3, pp. 272–283, 2008.

[23] J. Bach, "Test automation snake oil," in *Proceedings of the 14th International Conference and Exposition on Testing Computer Software (TCS '99)*, Washington, DC, USA, June 1999.

[24] M. Fewster, *Common Mistakes in Test Automation*, Grove Consultants, 2001.

[25] A. Hartman, M. Katara, and A. Paradkar, "Domain specific approaches to software test automation," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, pp. 621–622, Dubrovnik, Croatia, September 2007.

[26] C. Persson and N. Yilmaztürk, "Establishment of automated regression testing at ABB: industrial experience report on 'avoiding the pitfalls'," in *Proceedings of the 19th International Conference on Automated Software Engineering (ASE '04)*, pp. 112–121, Linz, Austria, September 2004.

[27] M. Auguston, J. B. Michael, and M.-T. Shing, "Test automation and safety assessment in rapid systems prototyping," in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP '05)*, pp. 188–194, Montreal, Canada, June 2005.

[28] A. Cavarra, J. Davies, T. Jeron, L. Mournier, A. Hartman, and S. Olvovsky, "Using UML for automatic test generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, Roma, Italy, July 2002.

[29] M. Vieira, J. Leduc, R. Subramanyan, and J. Kazmeier, "Automation of GUI testing using a model-driven approach," in *Proceedings of the International Workshop on Automation of Software Testing*, pp. 9–14, Shanghai, China, May 2006.

[30] Z. Xiaochun, Z. Bo, L. Juefeng, and G. Qiu, "A test automation solution on gui functional test," in *Proceedings of the 6th IEEE International Conference on Industrial Informatics (INDIN '08)*, pp. 1413–1418, Daejeon, Korea, July 2008.

[31] D. Kreuer, "Applying test automation to type acceptance testing of telecom networks: a case study with customer participation," in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pp. 216–223, Cocoa Beach, Fla, USA, October 1999.

[32] W. D. Yu and G. Patil, "A workflow-based test automation framework for web based systems," in *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC '07)*, pp. 333–339, Aveiro, Portugal, July 2007.

[33] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *Proceedings of the Future of Software Engineering (FoSE '07)*, pp. 85–103, Minneapolis, Minn, USA, May 2007.

[34] M. Blackburn, R. Busser, and A. Nauman, "Why model-based test automation is different and what you should know to get started," in *Proceedings of the International Conference on Practical Software Quality*, Braunschweig, Germany, September 2004.

[35] P. Santos-Neto, R. Resende, and C. Pádua, "Requirements for information systems model-based testing," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 1409–1415, Seoul, Korea, March 2007.

[36] ISO/IEC and ISO/IEC 15504-1, “Information Technology—Process Assessment—Part 1: Concepts and Vocabulary,” 2002.

[37] K. M. Eisenhardt, “Building theories from case study research,” *The Academy of Management Review*, vol. 14, no. 4, pp. 532–550, 1989.

[38] EU and European Commission, “The new SME definition: user guide and model declaration,” 2003.

[39] G. Paré and J. J. Elam, “Using case study research to build theories of IT implementation,” in *Proceedings of the IFIP TC8 WG 8.2 International Conference on Information Systems and Qualitative Research*, pp. 542–568, Chapman & Hall, Philadelphia, Pa, USA, May-June 1997.

[40] A. Strauss and J. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, SAGE, Newbury Park, Calif, USA, 1990.

[41] ATLAS.ti, The Knowledge Workbench, Scientific Software Development, 2005.

[42] M. B. Miles and A. M. Huberman, *Qualitative Data Analysis*, SAGE, Thousand Oaks, Calif, USA, 1994.

[43] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[44] C. Robson, *Real World Research*, Blackwell, Oxford, UK, 2nd edition, 2002.

[45] N. K. Denzin, *The Research Act: A Theoretical Introduction to Sociological Methods*, McGraw-Hill, New York, NY, USA, 1978.

[46] A. Fink and J. Kosecoff, *How to Conduct Surveys: A Step-by-Step Guide*, SAGE, Beverly Hills, Calif, USA, 1985.

[47] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, et al., “Preliminary guidelines for empirical research in software engineering,” *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[48] T. Dybå, “An instrument for measuring the key factors of success in software process improvement,” *Empirical Software Engineering*, vol. 5, no. 4, pp. 357–390, 2000.

[49] ISO/IEC and ISO/IEC 25010-2, “Software Engineering—Software product Quality Requirements and Evaluation (SQuaRE) Quality Model,” 2008.

[50] Y. Baruch, “Response rate in academic studies—a comparative analysis,” *Human Relations*, vol. 52, no. 4, pp. 421–438, 1999.

[51] T. Koomen and M. Pol, *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*, Addison-Wesley, Reading, Mass, USA, 1999.

[52] P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, Reading, Mass, USA, 2nd edition, 1998.

[53] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, Prentice-Hall, Upper Saddle River, NJ, USA, 2001.

[54] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Mass, USA, 2000.

[55] B. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine, Chicago, Ill, USA, 1967.

[56] C. Kaner, “Improving the maintainability of automated test suites,” *Software QA*, vol. 4, no. 4, 1997.

[57] D. J. Mosley and B. A. Posey, *Just Enough Software Test Automation*, Prentice-Hall, Upper Saddle River, NJ, USA, 2002.

[58] D. Foray, *Economics of Knowledge*, MIT Press, Cambridge, Mass, USA, 2004.