Hindawi Publishing Corporation
ISRN Software Engineering
Volume 2013, Article ID 420394, 10 pages
http://dx.doi.org/10.1155/2013/420394

Research Article

Regression Test Reduction for Object-Oriented Software: A Control Call Graph Based Technique and Associated Tool

Nicolas Frechette, Linda Badri, and Mourad Badri

Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, QC, Canada G9A 5H7

Correspondence should be addressed to Mourad Badri; mourad.badri@uqtr.ca

Received 5 February 2013; Accepted 5 March 2013

Academic Editors: K. Framling, F. Ipate, and S. K. Shukla

This paper presents a selective regression testing technique and an associated tool for object-oriented software. The technique is based on the concept of *Control Call Graphs,* which are a reduced form of traditional *Control Flow Graphs*. It uses static analysis of the source code of the program. The developed tool (1) identifies the *Control Call Paths* potentially impacted by changes, (2) selects, from an existing test suite, the appropriate test cases, and (3) generates new JUnit test cases for control call paths that are not covered by existing tests (new ones, or those whose structure has been modified after changes). In this way, the approach supports an incremental update of the test suite. The selected JUnit test cases, including the new ones, are automatically executed. Three concrete case studies are reported to provide evidence of the feasibility of the approach and its benefits in terms of reduction of regression testing effort.

1. Introduction

Software systems need to continually evolve for various reasons: adding new features to satisfy user requirements, changing business needs, introducing novel technologies, correcting faults, improving quality, and so forth. It is, therefore, important to ensure that modifications do not adversely affect the software. Software maintenance is an essential activity [1]. The costs of software maintenance keep, however, increasing [2]. Among the various maintenance activities, regression testing represents a crucial one. Regression testing is actually an important activity to ensure software quality, particularly when software is actively maintained and updated. It can also be used in the testing release phase of software development.

Regression testing is the process that consists of determining if a modified software system still verifies its specification and whether new errors were introduced inadvertently [3–5]. For obvious reasons, the retest-all approach that consists in rerunning every test case in the initial test suite produced during initial development is inefficient, costly, and unacceptable in the maintenance phase [6]. Moreover, it does not consider obsolete and new test cases. In addition, it is often impractical due to the development cost and delivery schedule constraints [7].

An alternative approach, known as a selective retest strategy, assumes that not all parts of a software system are affected by changes [8]. Regression test selection in this case consists in selecting and running, from an initial test suite, a reduced subset of test cases in order to verify the behavior of modified software and provide confidence that modifications and parts of the software impacted by modifications are correct [9, 10]. This leads obviously to a reduction in the cost of (regression testing and) software maintenance. If a selective retest strategy reveals the same faults as a retest-all strategy, then it is considered to be safe [11, 12]. Indeed, regression test selection techniques can discard test cases that could reveal faults, possibly reducing faults detection effectiveness [9]. Moreover, regression testing also needs to determine if additional test cases are required.

Regression testing strategies need to address different important issues [8, 13]: modification identification (finding where changes occur in a software and parts of the software that are possibly impacted by these modifications), test

selection (deciding which tests are more likely to reveal faults introduced by modifications), test execution (executing test cases and verifying the behavior of software), and test suite maintenance (determining where additional tests may be needed). Several techniques supporting regression testing (and particularly the regression test selection problem) have been proposed in the literature (e.g., [4, 10, 14–19]). These techniques, adopting different approaches, attempt to reduce the effort required to test a modified program by selecting an appropriate subset of test cases from a test suite used during development. The reuse of test cases offers, indeed, major advantages because the development of new test cases is a costly activity [8, 20–22]. Moreover, changes in software systems can introduce new control paths, possibly changing the structure of existing ones. The test cases developed for the original version of the program do not cover these changes. Often, new test cases are needed.

According to Engström et al. [13], no general solution has been put forward since no regression test selection technique could possibly respond adequately to the complexity of the problem and the great diversity in requirements and preconditions in software systems and development organizations. The improvement of the regression testing process aims basically to reduce the cost of maintenance. However, economic considerations are not the only reason. Regression testing also provides a certain confidence level to the modified software and improves (ensures) its reliability [23]. Regression testing involves somewhat a compromise between the cost of rerunning tests and the risk to avoid errors introduced inadvertently by the instantiated changes [13]. The different approaches proposed in the literature can be classified into two major classes [10, 13]: code-based approaches and model-based approaches.

A limited number of model-based approaches, especially for object-oriented software (OOS), have been proposed in the literature (e.g., [8, 21, 24, 25]). These approaches are independent of programming languages, which gives them more applicability. Fahad and Nadeem [26] argue that model-based regression testing techniques have some advantages compared to code-based regression testing techniques. However, model-based approaches have also some serious limitations. Particularly, models must be complete and up to date, which is not always the case in practice. In addition, the problem of traceability of tests must be addressed. According to Briand et al. [21], techniques based only on models may not be as accurate as code-based techniques (incompatibility between models and code). Furthermore, some changes in the source code may not have impact on models (some changes may be in fact not visible in models). Approaches based solely on models cannot capture this type of change [25].

Most of regression testing techniques proposed in the literature are code based (e.g., [3–5, 7, 19, 27–29]). According to Engström et al. [13], these techniques can achieve a high degree of precision in the selection of test cases. These techniques follow different approaches to support the regression testing process and consider different levels of granularity. Code-based techniques also have certain shortcomings, which are usually quite costly (particularly in the case of large and complex software systems) and may be prone to comprehension errors since the testers need to access and understand the source code [10, 30]. According to Chen et al. [31], code-based techniques are good for unit testing.

In this paper, we present a selective retest approach and an associated tool supporting the regression testing of OOS. The approach is based on the concept of *Control Call Graphs*, which are a reduced form of traditional *Control Flow Graphs*. It uses static analysis of the source code of the program. Control call graphs models capture the calls between methods (method level granularity) and related control flow. The control call paths, affected by a change, must be retested. As argued by Law and Rothermel in [32], working at the method level granularity makes the analysis more appropriate in practice, particularly in the case of large and complex software systems. Compared to existing regression testing approaches, the proposed technique (and associated tool) offers several advantages: (1) identifying control call paths using static analysis of the source code (does not require dynamic analysis), (2) covering the new control call paths (new ones, or those whose structure has been modified after changes), and (3) requiring no instrumentation of the source code (or specification unlike some approaches in the literature). In addition, the fact that the approach is based on control call graphs (which are a kind of abstraction), and not directly on code constructs, will reduce the complexity of the regression testing process. The proposed approach has been developed for Java applications. Three concrete case studies are reported in the paper to provide evidence of the feasibility of the approach and its benefits in terms of reduction of regression testing effort.

The rest of the paper is organized as follows. Section 2 introduces the methodology of the approach. Section 3 presents the associated tool. Section 4 presents the empirical study we conducted using three case studies. Section 5 provides a brief overview on related work. Finally, Section 6 gives a conclusion and some future work directions.

2. Regression Testing Methodology

The proposed approach is based on the concept of *Control Call Graphs* (CCG), which are a reduced form of traditional *Control Flow Graphs* (CFG). A CFG is a directed graph where the nodes represent decision points (if-then-else, while, case, etc.), an instruction or a block of sequential instructions. A block of sequential instructions is a sequence of instructions such that, if we execute the first instruction, we are sure to run the others, and always in the same direction. A directed arc linking node N_i to node N_j means that the instructions corresponding to node N_j are executed after the instructions corresponding to node N_i. The arcs of the graph indicate the transfer of control from one node to another.

A CCG is, in fact, a CFG from which the nodes representing instructions (or basic blocs of sequential instructions) not containing (and not leading to) a call to a method are removed. These graphs provide a global overview of the control flow (method calls and distribution of the control flow in the system). Compared to traditional call graphs (CC),

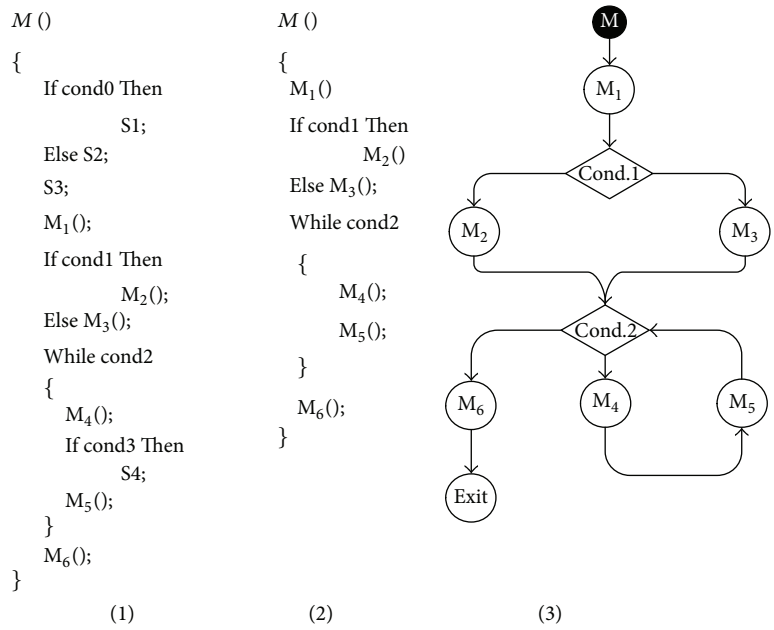

Figure 1: A method and its corresponding control call graph.

CCGs are much more precise models. CCGs capture, in fact, the structure of calls and related control.

Let us consider the example of method M given in Figure 1(1). The S_i represents blocs of instructions that do not contain a call to a method. The code of method M reduced to control call flow is given in Figure 1(2). The instructions (blocs of instructions) not containing a call to a method are removed from the original code of method M. Figure 1(3) gives the corresponding CCG.

The proposed regression testing technique covers the important issues, mentioned previously, that regression testing strategies need to address: change identification, test selection, test execution, and test suite maintenance. The technique is based on a static analysis of the source code of the modified program. It supports the identification of the control call paths (CCPs) impacted by changes, paths that must be tested. Moreover, it allows a conservative selection, from an existing test suite, of the appropriate test cases covering the impacted CCPs. It also supports the generation of new JUnit test cases to cover new CCPs or those whose structure has been modified after changes. In this way, the technique supports an incremental update of the original test suite. The selected test cases, including the new ones, are automatically executed.

Let P be the original (code of the) program and P' the (code of the) program obtained after instantiating changes. We focus on the CCPs impacted (directly or indirectly) by changes and new ones. The approach is organized in several steps (Figure 2).

2.1. Identifying Impacted CCPs. The goal of this step is to identify, by a static analysis of the source code of the two versions of the program P and P', the set I' of impacted CCPs. The set I' will be used later in the process to identify the appropriate test cases that must be retested. We use, in fact, an impact analysis tool that we developed in a previous work [33]. The impact analysis technique supported by this tool considers only the impact that can propagate through any CCP including a modified method M. This approach was previously used in [32] but using dynamic analysis. Dynamic analysis approaches can, indeed, achieve a high degree of precision but are however costly. In addition, dynamic analysis techniques are in general constrained by the quality of the data used [34]. Such techniques require, in fact, reliable data on the operational profile (execution traces) of the program. However, such data are not always available (and are costly to obtain), particularly in the case of large and complex software systems.

Initially, we perform a static analysis of the source code of the version P' of the program in order to build the CCGs of the different methods. From the CCGs, we generate the CCPs in a compacted form. These paths will be used to support change impact analysis. This aspect offers a significant advantage compared to dynamic approaches that obtain these paths by instrumenting the code and performing a dynamic analysis using execution traces. Figure 3 and Table 4 respectively give some examples of methods and corresponding compacted CCPs. We use (for more details see [33]) several notations to describe the control in the sequences of calls. The notation {*sequence*} expresses the iteration in the execution of the sequence of calls. The notation (*sequence 1*/*sequence 2*) expresses an alternative in the execution between *sequence* 1 and *sequence* 2. The notation [*sequence*] expresses the fact that the sequence can be executed or not.

In order to identify the set I' of impacted CCPs, we use the set C' of compacted CCPs of P' and the set M(*M*) of modified methods identified by static analysis of the source code of P

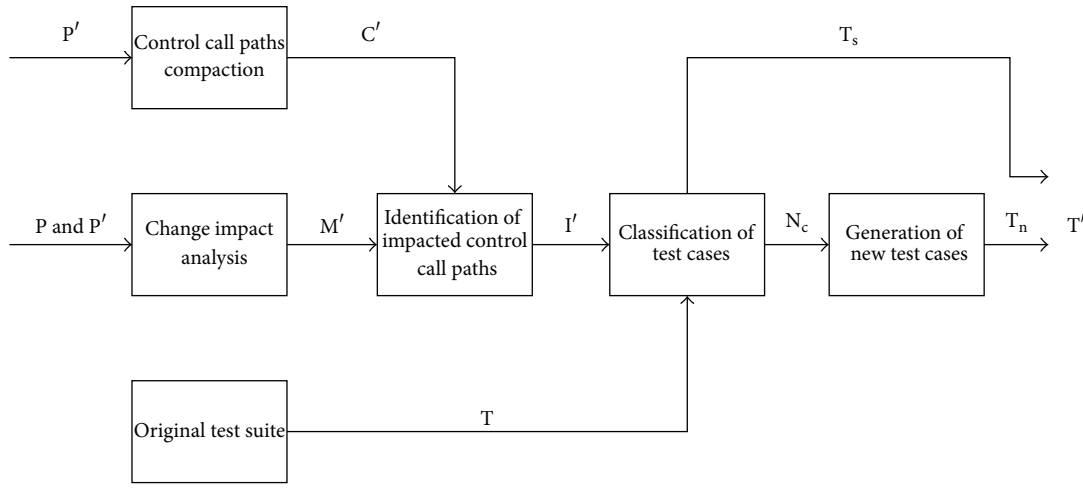

Figure 2: Methodology of the proposed approach.

```
M ()
 {  M1();
    If cond1 Then M2();
             Else M3();
    While cond2 Do
    {
      M4();
      M5();
    }
    M6();
}

M2()
 {  M7();
      If cond3 Alors M8();
 }

M3()
{
        M8();
 }

M6()
 {  If cond4 Then M8();
      M10();
 }

M8()
 {M9();}
```

Figure 3: Example of methods.

and P′. The list M(M) also contains the removed and added methods. The developed tool analyzes each of the CCPs by comparing the signature of the included methods with each of the signatures included in the set of modified methods M(M). The CCPs identified as impacted by modifications (including the new ones and those whose structure has been modified after changes) must be retested.

2.2. Classification of Test Cases. Regression testing is a very expensive activity and can represent more than half of the total cost of software maintenance [4]. The selection of appropriate test cases, from an existing test suite, saves valuable efforts. This avoids having testers to retest all existing test cases to test the modified program. In addition, using only the initial test suite can be inefficient, because it does not consider obsolete and new test cases. This step uses the set I′ of CCPs identified as impacted and the existing test suite T of P. The goal is to identify, using T and I′, the set T_s of test cases covering the impacted CCPs of the program P′. The objective is to ensure a conservative selection of test cases, so that all tests that could reveal a different behavior when executed on P′ are included. In addition to selecting the test cases to be reexecuted, corresponding to impacted CCPs, this step also identifies the set N_c (N_c is a subset of I′) of new CCPs (including the CCPs whose structure has been modified by changes) that are not covered by the existing test suite T. In this step, the various test cases included in T are analyzed and classified into different categories: *Obsolete* (test cases that are no longer valid—deleted), *Retestable* (test cases that cover CCPs that have been modified—noted T_s), and *Reusable* (test cases that cover CCPs that have not been modified—kept in the test suite but not used for regression testing). The *New* test cases, which cover new CCPs or those whose structure has been modified by changes, will be created.

2.3. Generation of New Test Cases. The set of test cases T is designed primarily to test the different CCPs of the initial version P of the program. This implies that, if new features are added to P, the new CCPs (or those whose structure has been modified by changes) of P′ are not covered by the set T. This is critical in industrial systems that are actively maintained and updated. The generation of new test cases is necessary and

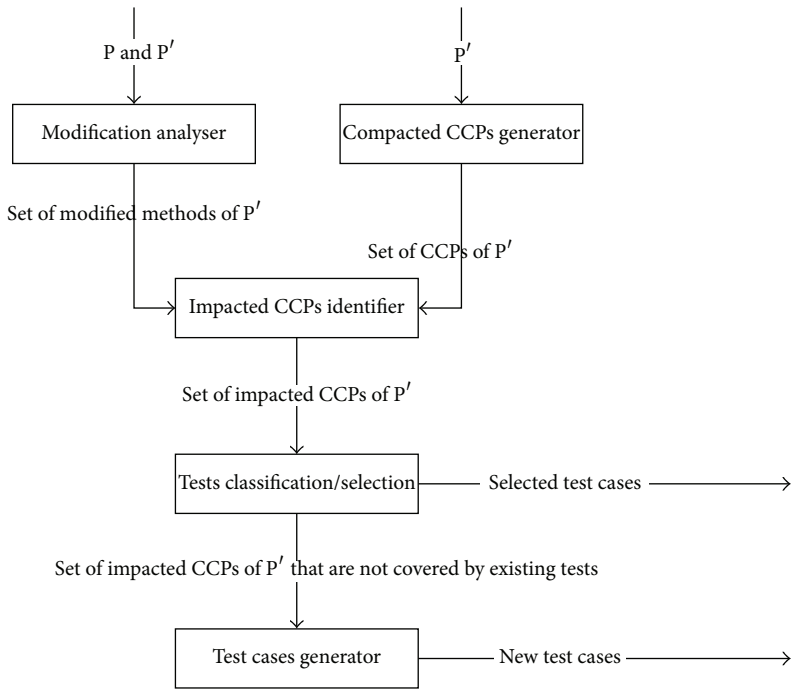

Figure 4: Tool architecture.

essential to conduct a comprehensive coverage of P' (covering each modified path or new one). Furthermore, if the set T initially had gaps in the sense that it did not cover all CCPs (and methods) of P, using only T to cover P' is insufficient even if no CCPs is added to P'. In this case, the generation of new test cases plays an important role in the iterative update of the original test suite. The developed tool supports the identification of CCPs that are not covered by existing tests and allows the automatic generation of new JUnit test cases without instrumentation (and dynamic analysis) of the code. In order to generate the set T_n of new test cases, the tool uses the set N_c of impacted CCPs that are not covered by existing tests. The sets T_n and T_s are paired to form the test suite T' that will be used to test the program P'.

3. Tool Presentation

We developed a (prototype) tool (in Java) to support the proposed approach. The tool consists of several modules. Figure 4 shows the overall architecture. It supports all phases of the methodology described in the previous section: generation of CCPs, identification of impacted CCPs, classification of test cases, and generation of new test cases. The developed tool is based, in part, on an extension of the JUnit Framework (http://www.junit.org/). JUnit is a simple Framework for writing and running automated unit tests for Java classes. The developed tool uses also the impact analysis tool that we developed in a previous work [33].

4. Empirical Study

This section presents the case studies we used in order to evaluate the proposed approach. The experimentation methodology is presented in Section 4.1, the evaluation criteria are presented in Section 4.2, the case studies considered are described in Section 4.3, the changes made to the different case studies are presented in Section 4.4, and the results of the empirical study are presented and discussed in Section 4.5.

4.1. Methodology. In order to provide evidence of the feasibility of the adopted methodology and its benefits in terms of regression testing effort reduction, we used our approach (and associated tool) on three case studies. We have made several changes to the original code of the three subject systems that we have considered so as to produce a new (modified) version for each of the systems. Thus, the original version of each subject system is represented by P and the modified one by P'. We have also developed the necessary JUnit test cases (original tests noted T) for each of the three subject systems (before instantiating changes). We have, however, deliberately omitted constructing some test cases corresponding to certain CCPs. The objective was essentially to test the ability of our approach (and tool) to detect these cases and generate appropriate test cases. Subsequently, the evaluation is performed by applying our technique (and associated tool) on each pair of successive versions (P, P'). For each pair (P, P'), corresponding to a subject system, we collected and analyzed data and interpreted results.

4.2. Evaluation Criteria. A review of the literature on the evaluation criteria used in different approaches allowed us to identify two major classes of criteria: criteria for the *reduction of the cost of regression testing* and criteria for the *effectiveness of the detection of faults* [10]. Although both classes are

important, in this paper we concentrate on test suite size reduction criteria.

The reduction of the size of a test suite, in terms of number of test cases, shows the gain in terms of efficiency achieved by using an approach rather than executing all test cases T. The reduction rate of the set T can be obtained by [35]: $(1 - (T_s/T)) * 100$. A loss of efficiency may result from the reduction of the size of a test suite. This can be measured in terms of precision and recall [35].

Let T be the set of test cases. Suppose that among these tests, n test cases result in different behavior when executed on P and P′. Let us also the set T_s, subset of T, containing m tests ($m \neq 0$). Among these m tests, l test cases can distinguish P from P′. The accuracy of T_s with respect to P, P′, and T is the percentage given by [35]: $100 * (l/m)$. The recall is the percentage given by [35]: $100 * (l/n)$ if $n \neq 0$ or else 100% if $n = 0$. The accuracy of a test suite is the percentage of those tests for which the old and the new programs could produce different results. The recall of a test suite is the percentage of tests selected from those that must be reexecuted. A low percentage of recall means that several test cases that should have been rerun were not selected.

Moreover, in order to evaluate our approach in terms of coverage of test cases, we defined a coverage metric Cov. This definition is inspired from the work of Ammann et al. [36] which was in turn inspired by the work of Wu et al. [37]. Let N be the number of impacted CCPs of the program P′ that are not covered by T_s, and let K be the number of impacted CCPs of the program P′ covered by T_n. The percentage of coverage of T_n on a set of changes is obtained by $\text{Cov} = 100 * (K/N)$, $N \neq 0$.

4.3. Case Studies. The first case study, the NextGen application, is an extension of the application developed in the book of Larman [38]. The original application which includes six classes has been extended for our purposes. The extended version totals 15 classes. We have added features about accounts receivable management, suppliers, and employees. We also added features to support billing and rental payments by debit and credit. The extension of NextGen generated the addition of 73 methods in total.

The second case study, the AgencyManagement application, is a management application, recovered from the website http://www.javafr.com/. It manages, with the console, the various activities of a travel agency. It offers, among other features, managing packages, flights, staff, passengers, and so forth.

The third case study, the LibraryManagement application, is also an application recovered from the website http://www.javafr.com/. It was designed to manage the ongoing operations of a library. Among the features offered, we find management of books, documents, customers, loans, and so forth. Table 1 presents some statistics on the case studies considered.

4.4. Changes Made. For our study, we have made several changes to the original code of the three subject systems that we have considered so as to produce a new version for each of the systems. We have made five different types of modifications on each application:

(i) modification of an instruction,

(ii) adding an instruction,

(iii) removing an instruction,

(iv) adding a new method,

(v) adding a new class.

Table 1: Some statistics on the used systems.

Statistics	NextGen	LibraryManagement	AgencyManagement
Number of classes	15	10	20
Number of methods	100	78	293
Number of initial test cases	13	58	104

Table 2: Case studies results.

Case Studies	Modified methods	Impacted CCPs	Selected test cases	CCPs that are not covered	New generated test cases
Case study 1	6	3	2	1	1
Case study 2	5	9	10	1	1
Case study 3	8	2	2	1	1

4.5. Evaluation. The first evaluation was performed using the NextGen application. The results reveal that there are 6 methods changed in NextGen′ (the modified version of NextGen). These methods are involved in 3 CCPs. The approach selected 2 test cases from the existing test suite to cover 2 impacted CCPs. The approach also generated 1 new test case to cover the remaining CCP. Table 2 summarizes the results. The second evaluation was performed using the application LibraryManagement. The approach identified 5 modified methods in LibraryManagement′. These methods are involved in 9 different CCPs. The approach selected 10 test cases from the initial test suite to cover 8 impacted CCPs. For the remaining CCP, 1 new test case was generated to cover it. The results are summarized in Table 2. The third evaluation was performed using the application AgencyManagement. The results reveal that there are 8 methods changed in AgencyManagement′. These methods are involved in 2 CCPs. The approach selected 2 test cases from the existing test cases to cover an impacted CCP. For the remaining sequence, a new test case was generated to cover it. Table 2 summarizes the results.

Table 3 presents a summary of the results, using the metrics defined in Section 4.2, for the three case studies. The percentage reduction of 85% obtained for the first case study

Table 3: Case studies results.

Evaluation criteria	Case study 1	Case study 2	Case study 3
Reduction	85%	81%	98%
Precision	100%	50%	100%
Recall	100%	100%	100%
Coverage	100%	100%	100%

Table 4: Compacted CCPs.

(1) $M : M_1\ (M_2/M_3)\ \{M_4, M_5\}\ M_6$
(2) $M_2 : M_7\ [M_8]$
(3) $M_3 : M_8$
(4) $M_6 : [M_8]\ M_{10}$
(5) $M_8 : M_9$

demonstrates the gain in terms of effort and cost provided by the proposed approach in contrast to the systematic execution of all existing test cases (retest all strategy). The precision indicates that, among the existing tests selected by our approach, 100% were likely to show a different behavior between the original and modified versions of the program (NextGen and NextGen′). No test was done for nothing. The recall of 100% indicates that all existing test cases actually revealing a different behavior between the original and modified programs (NextGen and NextGen′) have been selected. We have, in this case, the assurance that no useful test has been eliminated. Coverage for the first case study indicates that 100% of the CCPs that are not covered by existing test cases have been covered by the newly generated ones. This result shows that all CCPs of the modified version of the program (NextGen′) are now covered by the new test suite.

Overall, it is possible to make the same observations (and interpretation) from the results obtained for the two other case studies. For the second case study, the precision is 50%. This result means that from the existing test cases selected by our approach, 50% were likely to show a different behavior between the original and modified program. The reason of this result is only the poor quality of the original test suite for this particular case study. The test suite was, in fact, incomplete. However, the recall is 100%, which is a very good result. This means that all test cases that may reveal a different behavior between the two versions of the second case study are selected. For the third case study, the results are the same as those obtained from the first case study with a better reduction (98%).

In summary, the evaluation has shown that the application of our approach (and associated tool) on real case studies has allowed significant savings in terms of regression testing effort (more than 80% in the three case studies) without loss of accuracy (precision). Indeed, in two out of three case studies, the accuracy was 100%, and in the other case the accuracy was 50% (which is mainly due to the poor quality of the original test suite). Moreover, in the three case studies, our approach has a 100% recall (including case study 2). This means that all test cases that can reveal a different behavior between the two versions of the program were selected. In no case useful test cases have been excluded. The case studies have also shown that our approach can cover all CCPs that are not covered by existing test cases. This aspect confirms that all CCPs impacted by the modified version of a program are covered by the final test suite that consists of the union of existing test cases (selected and maintained) and new generated ones.

5. Related Work

Several approaches have been proposed in the literature to support regression testing [3–6, 14, 16–19, 27–29, 39–44]. Some of these approaches have focused more specifically on OOS [5, 16, 17, 28]. In this group, there are approaches based on the concept of "dangerous edges" [5, 16, 28] and others based on the concept of "firewall" [17, 18].

In regression testing techniques based on the concept of "firewall" [3, 17, 18], the elements included in the firewall may be elements that interact with modified elements, elements that are direct ancestors of modified elements, or elements that are direct descendants of the changed elements. In these approaches, the test cases covering the parts included in the firewall are selected to be reexecuted. Firewall techniques are simple and easy to use especially with small changes [6]. Different levels of granularity were used like dependencies between modules, functions, and classes. Kung et al. [3] present an algorithm based on the concept of firewall. Static code analysis is used to identify the classes that have been impacted by changes. This work focuses on the identification of the impacted classes and the determination of a test order but does not address the generation and execution of tests. Abdullah [27] elaborates, the concept of firewall presented previously [3]. The main novelty is that a distinction is made between high and low level changes. In addition, this approach takes into account polymorphism and dynamic binding. The generation of new test cases is discussed but no tool is mentioned. White et al. [29] present an extension to Adbullah's approach [27]. The extended firewall takes into account, in addition to the elements of the standard firewall, global variables, cycles, and paths. The approach does not cover the generation of new test cases. Skoglund and Runeson [19] evaluate the firewall approach when applied to a large system. The authors conclude that the time required for extraction and analysis of the data is more important than retesting all.

Another group of regression testing techniques is based on an approach proposed by Harrold et al. for procedural languages known as "DejaVu1" (unit level) and "DejaVu2" (integration level) [5, 16, 28]. Harrold et al. [5] present the first regression testing technique to support the Java language. The technique uses both static and dynamic analyses. The code of the program under test is instrumented. It is, in fact, an extension of Rothermel's DejaVu technique [4]. This approach selects the test cases that must be retested after a change, but do not address the problem of new test cases generation. A tool named RETEST allows the automation of the process. Rothermel et al. [28] present an extension of

the DejaVu technique adapted to the C++ language. A tool (DejaVOO) supports the approach.

Approaches based on the concept of "dangerous arcs" [5, 16, 28] are accurate and tend to select fewer tests than similar types of approaches to achieve an equivalent level of confidence. These approaches are conservative (safe). However, because of the fine granularity (level instructions) of information that has to be collected by instrumentation, they tend to be very costly. In addition, approaches based on dangerous arcs do not identify the execution paths that are not covered by existing tests and thereby do not support the generation of new test cases.

Firewall-based approaches [3, 17–19, 27, 29] are less accurate than approaches based on dangerous arcs because they do not reflect the control paths and tend to select more tests than dangerous arcs based approaches to achieve the same level of confidence. Such approaches simply work at a higher level of granularity (level classes) to select regression tests. They therefore require a much less expensive instrumentation. However, firewall approaches require the specifications of the original and modified programs. This aspect is a major drawback since in a real industrial context the specifications of modified applications are rarely available (and updated). The proposed firewall-based approaches, such as approaches based on dangerous arcs, do not support the generation of new test cases.

6. Conclusion and Future Work

We have presented a regression testing technique and associated tool for object-oriented software. The technique is essentially based on the concept of *Control Call Graphs*, which are a reduced form of traditional *Control Flow Graphs*. Compared to traditional call graphs, control call graphs are much more precise models. They capture, in fact, the structure of calls and related control. The technique uses a static analysis of the code (comparing the original and modified versions of the program) and does not require any instrumentation and dynamic analysis. It covers the different important issues that regression testing strategies need to address: change identification, test selection, test execution, and test suite maintenance.

The developed tool, combined with an impact analysis tool, identifies impacted control call paths, paths that need to be retested, and selects the appropriate JUnit test cases from an existing test suite. New test cases are generated when necessary. This allows updating the test suite incrementally. The generation of new test cases (to cover new control call paths and those whose structure has been modified after changes), in particular, distinguishes our approach from the main approaches in the literature that consider only execution paths covered by the original test suite. The proposed approach is conservative because all test cases that can reveal a different behavior between the original and modified versions of the program are selected.

In order to evaluate our approach, we used the tool we developed on three case studies. We focused on the reduction of the cost of regression testing. Results provide evidence of the feasibility of the methodology and its ability to reduce the regression testing effort (reducing the test suite size). The obtained results are conclusive and show that it is possible to perform regression testing safely based on the code (static analysis) while minimizing the reuse of existing test cases.

The study performed in this paper should however be replicated using many other object-oriented software systems in order to draw more general conclusions. The achieved results are based on the dataset we collected from the three subject systems. Even if we believe that the analyzed dataset is large enough to allow obtaining significant results, we do not claim that our results can be generalized to all systems. The study should be replicated on a large number of object-oriented software systems to increase the generality of our findings. Moreover, it would be interesting to replicate the study on industrial systems.

As future work, we plan to extend (and explore the potential of) the approach to support the coverage of changes to the class level of granularity, use other criteria to improve the evaluation of the approach, and finally replicate the study on other object-oriented software systems to be able to give generalized results.

Acknowledgment

This work was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

References

[1] K. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, ACM, New York, NY, USA, 2000.

[2] I. Sommerville, *Software Engineering*, Addison Wesley, 9th edition, 2010.

[3] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8, no. 2, pp. 51–65, 1995.

[4] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[5] M. J. Harrold, J. A. Jones, T. Li et al., "Regression test selection for Java software," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01)*, pp. 312–326, October 2001.

[6] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[7] Y. Wu, M. H. Chen, and H. M. Kao, "Regression testing on object-oriented programs," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE '99)*, pp. 270–279, November 1999.

[8] O. Pilskalns, G. Uyan, and A. Andrews, "Regression testing UML designs," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 254–263, September 2006.

[9] T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection

techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, 2001.

[10] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM '08)*, pp. 22–31, October 2008.

[11] H. K. N. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings of the Conference on Software Maintenance*, pp. 290–301, San Diego, Calif, USA, November 1990.

[12] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *Proceedings of the Conference on Software Maintenance*, 1992.

[13] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[14] L. J. White and H. K. N. Leung, "A firewall concept for both control-flow and data-flow in regression integration testing," in *Proceedings of the Conference of Software Maintenance*, pp. 262–271, 1992.

[15] G. Rothermel and M. J. Harrold, "A comparison of regression test selection techniques," Tech. Rep. 114, Clemson University, Clemson, SC, USA, April 1993.

[16] G. Rothermel and M. J. Harrold, "Framework for evaluating regression test selection techniques," in *Proceedings of the 16th International Conference on Software Engineering*, pp. 201–210, May 1994.

[17] P. Hsia, X. Li, D. C. Kung et al., "A technique for the selective revalidation of OO software," *Journal of Software Maintenance and Evolution*, vol. 9, no. 4, pp. 217–233, 1997.

[18] L. White and K. Abdullah, "A firewall approach for the regression testing of object oriented software," *Software Quality Week*, 1997.

[19] M. Skoglund and P. Runeson, "A case study of the class firewall regression test selection technique on a large scale distributed software system," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE '05)*, pp. 74–83, November 2005.

[20] H. K. Leung and L. White, "A cost model to compare regression test strategies," in *Proceedings of the International Conference on Software Maintenance*, pp. 201–208, 1991.

[21] L. C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 252–261, October 2002.

[22] Y. Chen, R. L. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," in *Proceedings of the 3rd International Workshop Advances in Model Based Testing (AMOST '07)*, pp. 54–62, September 2007.

[23] Y. Li and N. J. Wahl, "An overview of regression testing," *ACM Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, 1999.

[24] A. Von Mayrhauser and N. Zhang, "Automated regression testing using DBT and sleuth," *Journal of Software Maintenance and Evolution*, vol. 11, no. 2, pp. 93–116, 1999.

[25] P. L. Vincent, L. Badri, and M. Badri, "Regression testing of object-oriented software: a technique based on use cases and associated tool," in *Proceedings of the International Conference on Advanced Software Engineering & Its Applications (ASEA '12)*, T.-H. Kim, C. Ramos, H. Kim, A. Kiumi, S. Mohammed, and D. Ślęzak, Eds., Lecture Notes in Computer Science, Springer, Jeju, Korea, December 2012.

[26] M. Fahad and A. Nadeem, "A survey of UML based regression testing," in *IFIP International Federation for Information Processing*, Z. Shi, E. Mercier-Laurent, and D. Leake, Eds., vol. 288, pp. 200–210, Springer, Boston, Mass, USA, 2008.

[27] K. Abdullah, *The firewall concept for regression testing and impact analysis of object oriented systems [Ph.D. thesis]*, Case Western Reserve University, 1998.

[28] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for C++ software," *Journal of Software Testing, Verification and Reliability*, vol. 10, no. 2, pp. 77–109, 2000.

[29] L. White, K. Jaber, and B. Robinson, "Utilization of extended firewall for object-oriented regression testing," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 695–698, September 2005.

[30] N. Mansour, H. Takkoush, and A. Nehme, "UML-based regression testing for OO software," *Journal of Software Maintenance and Evolution*, vol. 23, no. 1, pp. 51–68, 2011.

[31] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-based regression test selection with risk analysis," in *IBM Center Advanced Studies Conference*, 2002.

[32] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 308–318, May 2003.

[33] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC '05)*, pp. 167–175, December 2005.

[34] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '11)*, pp. 128–137, Helsinki, Finland, September 2003.

[35] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "Study of effective regression testing in practice," in *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE '97)*, pp. 264–274, November 1997.

[36] P. Ammann, P. E. Black, and W. Ding, "Model checkers in software testing," NIST-IR 6777, National Institute of Standards and Technology, 2002.

[37] D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell, "A practical method for software quality control via program mutation," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pp. 159–170, 1988.

[38] G. Larman, *UML et les Design Pattern*, Campus Press, 2002.

[39] Y. F. Chen, D. S. Rosenblum, and K. P. Vo, "TestTube: a system for selective regression testing," in *Proceedings of the 16th International Conference on Software Engineering*, pp. 211–220, Sorrento, Italy, May 1994.

[40] B. Korel and A. M. Al-Yami, "Automated regression test generation," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 98)*, pp. 143–152, Clearwater Beach, Fla, USA, 1998.

[41] T. Ball, "On the limit of control flow analysis for regression test selection," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 98)*, pp. 134–142, Clearwater Beach, Fla, USA, 1998.

[42] A. K. Onoma, W. T. Tsai, M. H. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[43] L. White, H. Almezen, and S. Sastry, "Firewall regression testing of GUI sequences and their interactions," in *Proceedings of the International Conference on Software Maintenance*, pp. 398–409, September 2003.

[44] T. Koju, S. Takada, and N. Doi, "Regression test selection based on intermediate code for virtual machines," in *Proceedings of the International Conference on Software Maintenance*, pp. 420–429, September 2003.