Regression Testing of Object-Oriented Software: Towards a Hybrid Technique

Pierre-Luc Vincent, Linda Badri and Mourad Badri

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

{PierreLuc.Vincent, Linda.Badri, Mourad.Badri}uqtr.ca

Abstract

We propose, in this paper, a hybrid regression testing technique and associated tool for object-oriented software. The technique combines, in fact, the analysis of UML models to a simple static analysis of the source code of the modified program. The basic models we use are use cases model and corresponding UML statechart and collaboration diagrams. The goal of the static analysis of the source code is to identify changes that are not visible in design models. The developed tool identifies the modified (and/or impacted by modifications) use cases and selects the appropriate test cases from an existing test suite. New (JUnit) test cases, covering new scenarios or those whose structure has been modified after changes, are generated when necessary. In this way, the technique supports an incremental update of the test suite. The selected JUnit test cases, including the new ones, are automatically executed. A case study is reported to provide evidence of the feasibility of the approach and its benefits in terms of reduction of regression testing effort.

Keywords: *Software Maintenance, Regression Testing, Object-Oriented, Test Case Selection, Test Case Generation, Use Cases, UML Diagrams, Static Analysis*

1. Introduction

Modern software development is becoming more and more complex. Moreover, as software systems are used for a long period of time, software evolution is ineluctable. Software systems need, indeed, to continually evolve during their lifecycle for various reasons: adding new features to satisfy user requirements, changing business needs, introducing novel technologies, correcting faults, improving quality, *etc*. So, as software evolves, the changes made to the software must be carefully managed. In particular, it is important to ensure that modifications do not adversely affect the software. Software maintenance (and evolution) plays a key role in the overall lifecycle of software. Furthermore, it takes a large part of the lifecycle costs. 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 a process that consists of determining if a modified software system still verifies its specifications and whether new errors were introduced inadvertently [1-3]. 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 [4]. Moreover, it does not consider obsolete (no longer valid) and new test cases. In addition, it is often impractical due to the development cost and delivery schedule constraints [5]. An alternative approach, known as a selective retest approach, assumes that not all parts of a software system are affected by changes [6]. Regression test selection in this case consists in selecting and running, from an initial test suite, a reduced subset of appropriate test cases in order to verify the behavior of modified software and provide confidence that modifications, and parts of the software affected by modifications, are correct [7, 8]. This leads to a reduction in the cost of (regression testing and) software maintenance. If a selective retest approach reveals the same faults as a retest-all approach, then it is considered to be safe [9-13]. Regression test selection techniques can, in fact, discard test cases that could reveal faults, possibly reducing faults detection effectiveness [7]. Furthermore, regression test techniques also need to determine if additional test cases are required.

Regression testing techniques need to address different important issues [6, 12]: 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). Many researchers have addressed the regression testing problem (and particularly the regression test selection problem) in the literature [1-3, 5, 6, 8, 12-23]. These techniques, adopting different approaches, attempt to reduce the effort required to test a modified program by selecting a suitable set of test cases from a test suite used during development. The reuse of test cases offers, indeed, major advantages because the creation of new test cases is a costly activity [6, 14]. Moreover, changes in software can introduce new scenarios, and/or change the structure of existing ones. Original test cases do not cover these changes (and their impact). New test cases are often needed. According to Engström et al. [12], 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.

Most of the regression testing techniques proposed in the literature are code-based [1-3, 5, 17, 20, 22, 23]. Engström *et al.*, [12] argue that 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 have, however, certain shortcomings: usually quite costly (particularly when working with large and complex software systems) and may be prone to comprehension errors since the testers need to access and understand the source code [8, 24]. Chen *et al.*, [25] argue that code-based techniques are good for unit testing but have a scalability problem. In addition, some of these techniques are not extensible enough to apply to large components [22]. Fahad *et al.*, [26] argue that it is more difficult to identify changes from the code than from the models. Finally, these techniques are dependent on the programming language used and in some cases don't support all of its constructions [1].

A limited number of regression testing approaches are based on models, especially for object-oriented software (OOS) [6, 18, 21]. These approaches are independent from programming languages, which gives them more applicability. According to Fahad *et al.*, [26], model-based regression testing techniques have many advantages over code-based techniques. However, model-based approaches also have some limitations. Particularly, models must be complete and up-to-date. Advances in the field of reverse engineering allow, however, reducing the effects of such a drawback [27]. Different models can, actually, be generated automatically from the source code of a program. Moreover, according to Briand *et al.*, [21], techniques based only on models may not be as accurate as code-based techniques (incompatibility between model and code). Some changes in the source code of the programs may not have impact on models. In fact, models being abstraction are often insensitive to minor code changes (changes in a method's body for example). Regression testing techniques based solely on models cannot capture this type of change.

In this paper, we present a hybrid regression testing technique, and associated tool, for OOS. The proposed technique combines, in fact, the analysis of UML models to a simple static analysis of the source code of the modified program. The basic models we use in our approach are use cases model and corresponding UML (Unified Modeling Language) statechart and collaboration diagrams. The static analysis of the source code aims basically at identifying changes (in a method's body) that are not visible in design models. In this way, the technique combines the advantages of both model-based and code-based approaches to improve the accuracy in the selection of adequate test cases. The developed tool identifies the modified (and/or impacted by modifications) use cases (parts impacted by modifications) and selects the appropriate test cases, from an existing test suite, that must be retested. New JUnit test cases, covering new scenarios or those whose structure has been modified after changes, are also generated when necessary. In this way, the technique supports an incremental update of the test suite (test suite maintenance, which is a crucial issue in regression testing). The selected JUnit test cases, including the new ones, are automatically executed. A case study is reported to provide evidence of the feasibility of the methodology and its ability to reduce the regression testing effort.

The article is organized as follows: A brief review of the literature on regression testing techniques for OOS is presented in Section 2. Section 3 presents the methodology of the proposed approach and associated tool. Section 4 presents the case study, the definition of the evaluation criteria and a discussion of the results. Finally, Section 5 concludes the paper and gives some future work directions.

2. Related Work

Rothermel *et al.*, [2] present a regression testing technique using both static and dynamic analysis of programs. The code of the program under test is instrumented. The test cases that cover modified code are executed. The execution time varies depending on several factors (for example, cases where the modifications change the control graph's path structure). Harrold *et al.*, [3] present the first regression testing technique to support the Java language. The technique uses both static and dynamic analysis. The code here also is instrumented. It is, in fact, an extension of Rothermel's DejaVu technique [2]. 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 (RETEST) allows the automation of the process. Rothermel *et al.*, [20] present an extension of the DejaVu technique adapted to the C++ language. The test selection process is supported by a tool (DejaVOO).

Kung *et al.*, [1] present an algorithm based on the concept of firewall. The technique isolates modules that need to be re-tested after a change. 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 *et al.*, [17] elaborate the concept of firewall presented by Kung *et al.*, [1]. The main novelty of the approach is that a distinction is made between high level 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.*, [23] present an extension to Adbullah's approach [17]. The extended firewall takes into account, in addition to the elements of the standard firewall, global variables, cycles and paths. In regression testing techniques based on the concept of firewall, 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. This work does not cover the generation of new test cases. Skoglund *et al.*, [22] evaluate the firewall technique on a large system. The authors conclude that the time required for extraction and analysis of the data is more important than retesting all. Firewall techniques are simple and easy to use especially with small changes [9].

Wu *et al.*, [5] propose a technique based on dependency relationships (Affected Function Dependency Graphs) to identify variables and functions affected by changes. This technique addresses only the test case selection problem. Chen *et al.*, [25] use an activity diagram (control flow graph) to describe system requirements, behaviors and workflows of underlying system to test. The paths that correspond to the affected graph nodes determine the tests to be rerun. Wu *et al.*, [28] use class, collaboration and statechart UML diagrams for regression component-based software. Pilskalns *et al.*, [6] present a regression test selection technique based on UML class and sequence diagrams. The technique takes into account OCL (Object Constraint Language) expressions. The approach combines information from class diagrams and sequence diagrams in a direct acyclic graph. No tool is mentioned to automate the approach.

Briand *et al.*, [21] present an impact analysis and regression test selection technique based on UML designs. The used models are class, sequence and use cases diagrams. After a change, the two versions of the different models are compared and the test cases are classified into: obsolete, re-testable and reusable. A tool (RTSTool) is used to automate the approach. The authors mention that it is likely that the approach is not as accurate as if it was based on the source code. Mansour *et al.*, [24] present also a regression test selection technique based on UML models. The used models are class, interaction and interaction overview diagrams.

3. Regression Testing Methodology

Use cases are used to describe functional requirements. Informally, a use case is a collection of related success and failure scenarios that describe actors using a system to support a goal [29]. A *scenario*, also called a *use case instance*, is a specific sequence of actions and interactions between actors and the system. It is one particular story of using the system, or one path through the use case. The development (and testing) process is driven by use cases. Use cases can be described by several UML models. A useful application of statechart diagrams is to describe the legal sequence of external system events that are recognized and handled by a system in the context of a use case. A statechart diagram that depicts the overall system events and their sequence within a use case is a kind of *use case statechart diagram* [29]. Moreover, in OOS, objects interact in order to implement the behavior. The dynamic interactions between groups of objects may be specified using UML collaboration diagrams. Collaboration defines, in fact, the roles a group of objects play when performing a particular task (a complex operation for example). The specification described in a collaboration diagram must be preserved during the transformation process into an implementation [30, 31].

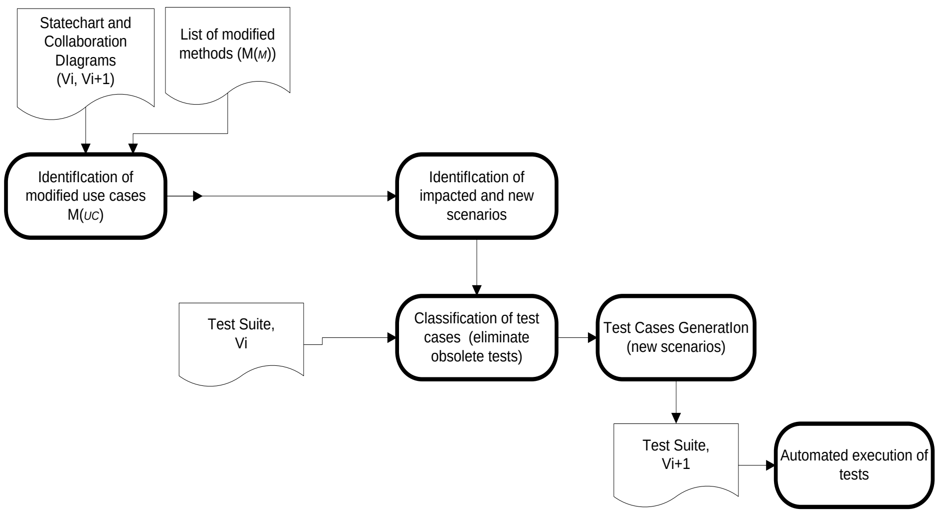

Statechart and Collaboration DIagrams (Vi, Vi+1)
List of modified methods (M(M))
IdentifIcation of modified use cases M(UC)
IdentifIcation of impacted and new scenarios
Test Suite, Vi
Classification of test cases (eliminate obsolete tests)
Test Cases GeneratIon (new scenarios)
Test Suite, Vi+1
Automated execution of tests

Figure 1. Iterative Methodology

The proposed approach, illustrated by Figure 1, covers the important issues that regression testing strategies need to address: change identification, test selection, test execution and test suite maintenance. The technique supports the identification of use cases affected by changes (scenarios to be (re-) tested). Moreover, it allows the selection, from an existing test suite, of the test cases appropriate to cover the modified (impacted) scenarios. It also supports the generation of new test cases when necessary. The different test cases selected (or newly created) are executed automatically. We assume in our approach that UML models are updated after modifications. Let V_i and V_{i+1} be two versions of a program P (models and source code). The version V_{i+1} is obtained following changes instantiated to the version V_i. We focus on the scenarios impacted by changes (and new ones). The approach is organized in several steps.

3.1. Identification of Modified Methods

In order to identify the modified methods, we perform a static analysis of the source code of the two versions V_i and V_{i+1} of the program (Figure 2). We use, in fact, an impact analysis tool that we developed in a previous work [32]. The obtained list M(*M*) of modified methods also contains the removed and added methods. This is particularly useful because it allows identifying the changes (change in a method's body) that are made to the source code of the program that do not require an update of the design models (not visible in design models).

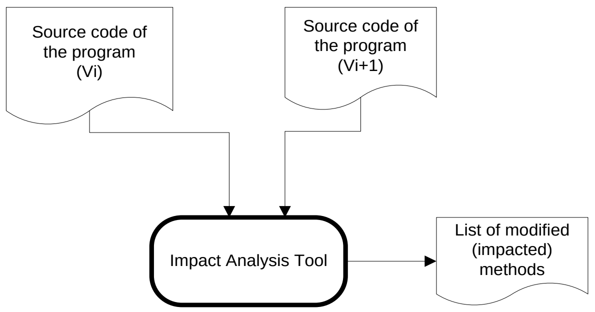

Source code of the program (Vi)
Source code of the program (Vi+1)
Impact Analysis Tool
List of modified (impacted) methods

Figure 2. Identification of Modified (impacted) Methods

3.2. Identification of Impacted Use Cases

We determine the set M(*UC*) of use cases that have been affected by one or more modifications to assist in identifying (and classifying) appropriate test cases. Each use case identified as impacted by changes will be marked (marking of the corresponding statechart and collaboration diagrams - version V_{i+1}). The goal is to identify the set M(*S*) of modified (impacted and/or new) scenarios of a use case. In order to ensure that modifications have not adversely affected the system, all these scenarios will be (re-) tested. This step is essentially based on the comparison of the diagrams corresponding to the different use cases (statechart and collaboration diagrams) of the two versions V_i and V_{i+1} (creating a mapping of the changes between the two versions). UML diagrams are described in our approach using XML. The comparison uses also the list M(*M*) of modified methods. Figure 3 shows an example of a use case statechart diagram for the use case "*Process a sale*" of a sales management application [29]. Figure 4 shows the collaboration diagram of the *enterArticle* method. The use case statechart diagram of Figure 3 contains a transition named *enterArticle*.

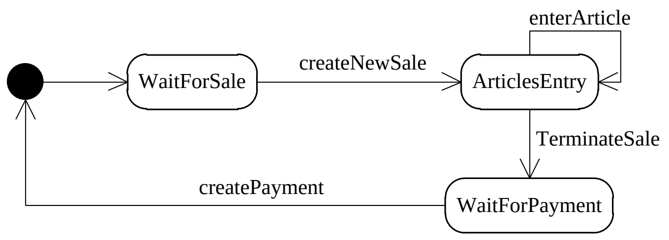


Figure 3. A Use Case Statechart Diagram - *Process a Sale* [29]

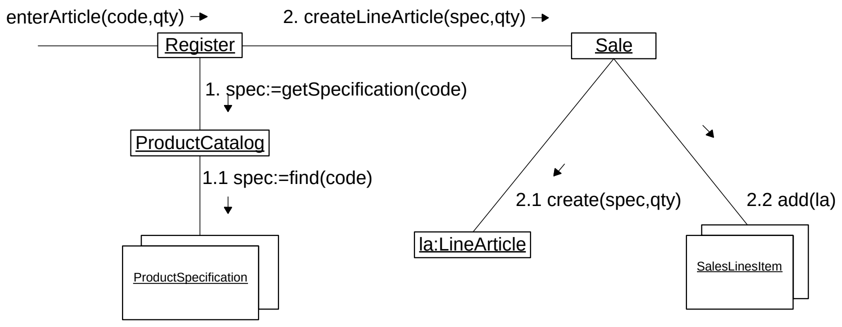


Figure 4. Collaboration Diagram of the *enterArticle* Operation [29]

3.3. Identification of Impacted Scenarios

For each modified (impacted) use case UC_i, the corresponding statechart diagram (version V_{i+1}) is transformed into a tree T_{SD} (the modified scenarios are marked). Each collaboration diagram corresponding to a modified method is also transformed into a tree T_{CD} (the modified sequences of calls of the methods are marked - collaboration). Each modified method, included in a scenario of the use case, will be replaced by its own tree (sequence of calls). The different (complete) scenarios of the use case (from T_{SD}) that have been modified M(*S*) (impacted scenarios, new scenarios and scenarios whose structure has been changed) must be (re-) tested. We draw, in this process, on the approaches that we have developed in previous work [31, 33, 34].

Let us consider the used example. If the collaboration "*enterArticle*" (Figure 4) is changed, then the use case statechart diagram "*Process a sale*" (Figure 3) will be marked at the level of the transition "*enterArticle*". The tree corresponding to the use case "*Process a sale*" of Figure 3 is shown in Figure 5. In this example, we can clearly see that there are two possible scenarios. The first scenario is one where the *enterArticle* method is called once. The second scenario is one where the *enterArticle* method is called several times.

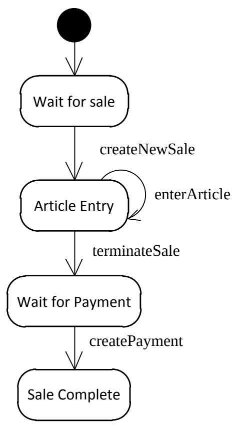


Figure 5. Message Tree for the Use Case *Process a Sale*

The message tree of the '*enterArticle*' collaboration is shown in Figure 6. As it is a trivial collaboration, there is only one possible path. As '*enterArticle*' is changed, then the path of the *'Process a sale'* use case that contains the '*enterArticle*' transition should be re-tested and '*enterArticle*' will be replaced by its own sequence appearing at the level of the message tree (Figure 5).

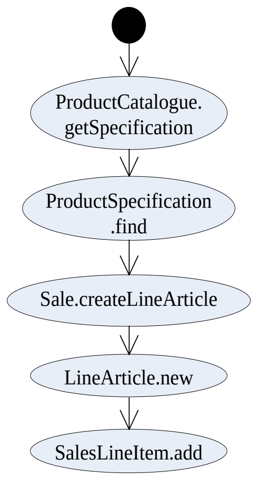


Figure 6: Message Tree for the 'enterArticle' Collaboration

3.4. Classification of Test Cases

We focus, in this step, on the identification (classification and eventually generation) of the test cases corresponding to the set of impacted scenarios M(*S*). With the set of impacted scenarios M(*S*) identified previously, this step allows to select, from the existing test suite, the test cases covering the impacted scenarios (reuse). We perform, in fact, a static analysis of the XML descriptions of the models combined to a static analysis of the source code of the test cases (JUnit code). We also identify the scenarios that are not covered by the existing suite for which new test cases are generated. The initial test suite is thus updated incrementally. In this step, the various test cases are analyzed and classified into different categories: Obsolete (test cases that are no longer valid - deleted), Retestable (test cases that cover scenarios that have been modified), Reusable (test cases that cover scenarios that have not been modified – kept in the test suite but not used for regression testing) and New (new test cases that cover new scenarios or scenarios whose structure has been modified by changes).

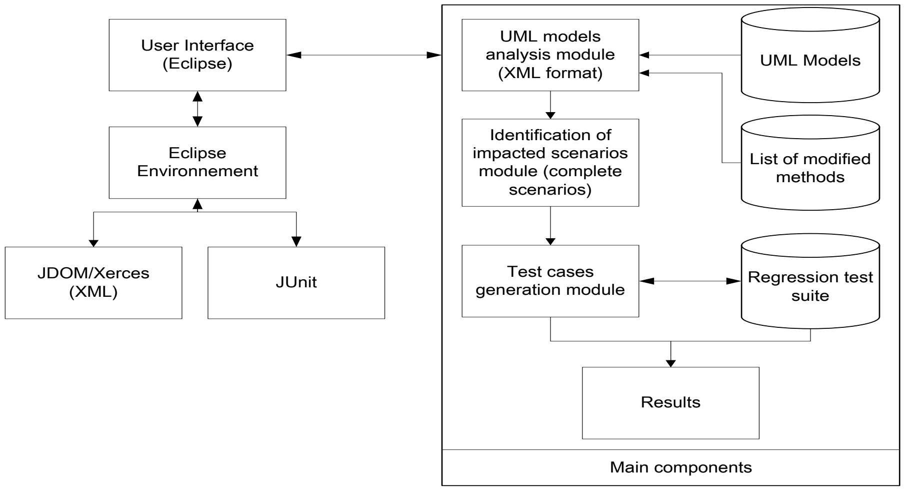


Figure 7. Architecture of the Tool

The approach we propose considers both unit testing (unit test cases - methods) and integration testing (integration test cases - use cases – impacted, modified and new scenarios). The (prototype) tool that we developed, based in part on an extension of the JUnit Framework (http://www.junit.org), allows the generation and automatic execution of test cases. All new and re-testable test cases are tested. The architecture of the tool is given in Figure 7. It is composed of several modules. It supports all phases of the methodology.

4. Case Study

4.1. The case study

In order to provide evidence of the feasibility of the methodology and its benefits in terms of regression testing effort reduction, we used our approach (and associated tool) on a case study. The case study is an ATM (simulator) system (taken from the literature), allowing to perform basic banking operations (withdrawal, deposit, transfer, balance, *etc*.). We adapted the case study for our purposes. Figure 8 gives the use cases model of the application.

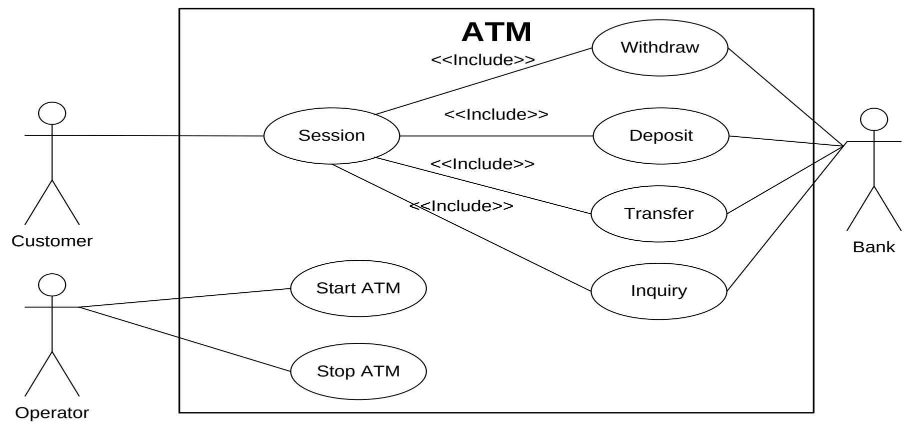


Figure 8. Use Cases Model

To evaluate our approach, we have made several changes to the different models of the original application (V_1) to produce three successive versions (V_2, V_3 and V_4). In addition, we have also made several changes on some methods in the source code of the version V_3. We have deliberately made changes that are not visible in the design models. The objective was basically to test the ability of our approach (and associated tool) to detect these changes and select the appropriate test cases (from the original test suite). We have, in fact, developed the necessary JUnit test cases (original tests) for our application (before instantiating changes). Subsequently, the evaluation is performed by applying our technique on each pair of successive versions ((V_1, V_2), (V_2, V_3) and (V_3, V_4)). The evaluation is performed in three iterations. Each iteration includes data collection and analysis and interpretation of results. We compared our methodology with the retest-all strategy.

Table 1. Changes made between Versions 1 and 2

	Total (V.1)	Added	Changed	Deleted	Total (V.2)
Methods	**32**	5	2	3	**34**
Classes	**13**	1	5	0	**14**
Use Cases	**7**	0	3	0	**7**

Table 2. Changes made between Versions 2 and 3

	Total (V.2)	Added	Changed	Deleted	Total (V.3)
Methods	**34**	3	5	1	**36**
Classes	**14**	0	3	1	**13**
Use Cases	**7**	0	5	0	**7**

Table 3. Changes made between Versions 3 and 4

	Total (V.3)	Added	Changed	Deleted	Total (V.4)
Methods	**36**	0	4	0	**36**
Classes	**13**	0	0	0	**13**
Use Cases	**7**	0	0	0	**7**

The initial specification of the application has 7 use cases, 7 statechart diagrams and 4 collaboration diagrams. The implementation (in Java) has a total of 13 classes and 32 methods. The second version includes changes made on 3 of the 7 use cases. Two methods have been renamed, a method was moved to another class, a transition was added to the statechart diagram describing one use case (*Inquiry*) and a message has been added in the collaboration diagram of one method. Table 1 presents the detailed statistics on the differences between version V_1 and version V_2 of the application. The third version of the application includes changes made on 5 of the 7 use cases. A new collaboration diagram is added to keep the credit card after three unsuccessful attempts by the user to enter his password. The statechart diagrams of the 4 possible transactions are modified to no longer to eject the card after a transaction is done to allow more than one transaction per session. Finally, a method is moved to another class and a transition is added to the statechart diagram of one use case (*Session*) to allow users to make more than one transaction per session. Table 2 presents the detailed statistics on the differences between version V_2 and version V_3 of the application. The fourth version of the application includes changes made only to the source code of the program that do not require changes to the models. The following methods have been changed: *SecurityAudit.setFailedPinCount*, *Transaction.invalidPin*, *Deposit* (constructor), and *Inquiry* (constructor). Table 3 presents the detailed statistics on the differences between version V_3 and version V_4 of the application.

4.2. Evaluation Criteria

A review of the literature on the evaluation criteria used by different researchers 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* [8]. Although both classes are important, in this paper we concentrate on test suite size reduction criteria. We adapted some criteria defined in the literature to evaluate: the reduction of the number of test cases to re-test both at the integration level (in terms of scenarios) and at the unit level (in terms of methods), and the reuse rate of test cases. Let P be a program.

Definition 1: The reduction of the test suite at the integration testing level is given by: *ReductInteg*(P) = 1 – (STC/TC), where STC represents the selected test cases and TC represents the set of all test cases of the program.

Definition 2: The reduction of the test suite at the unit level is given by: *ReductUnit*(P) = 1 – (M/AM), where M represents the methods that are part of the selected test cases and AM represents the set of all methods.

Definition 3: The reuse rate of the test suite at the integration level is given by: *ReuseInteg*(P) = RITC / ITC, where RITC represents the number of integration test cases (sequences) that are classified reusable or re-testable and ITC represents the total number of integration test cases (sequences).

Definition 4: The reuse rate of the test suite at the unit level is given by: *ReuseUnit*(P) = RUTC / UTC, where RUTC represents the number of unit test cases (methods) that are classified reusable or re-testable and UTC represents the total number of unit test cases (methods).

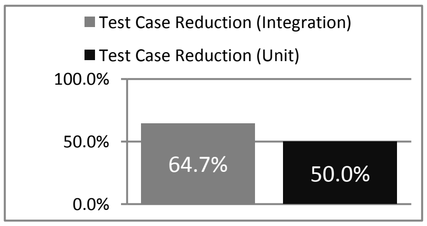
4.3. Results and Discussion

Iteration 1: From version V_1 to version V_2

At this iteration, for integration testing, there are 6 test cases (complete test sequences) selected on a total of 17. The 6 test cases correspond, in fact, to new scenarios. For unit tests, there are 17 unit test cases selected on a total of 34 test cases (see Table 4), including 5 new and 12 re-testable. Figure 9 shows the evaluation results based on the two criteria *ReductInteg* and *ReductUnit* for test suite reduction. The reduction of the test suite is significant for both integration (64.7%) and unit (50%) levels when compared to retest-all strategy although the changes we made (from version 1 to version 2) affect the majority of use case scenarios. Figure 10 shows the evaluation results based on the two other criteria *ReuseInteg* and *ReuseUnit* for the reuse rate of test cases of our approach. The reuse rate of the test suite is here also significant for both integration (64.7%) and unit (85.3%) levels.

Table 4. Classification of Test Cases (Iteration 1)

	Total (V.1)	Obsolete	Retestable	Reusable	New	Total (V.2)
Unit	**32**	3	12	17	5	**34**
Integration	**15**	4	0	11	6	**17**



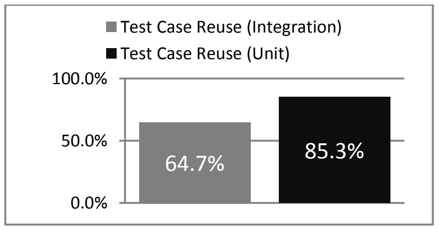


Figure 9. Test Suite Reduction **Figure 10. Test Case Reuse Rate**

Iteration 2: From version V_2 to version V_3

At this iteration, for integration testing, there are 10 test cases (complete test sequences) selected on a total of 20. The 10 test cases correspond to new scenarios. For unit tests, there are 23 unit test cases selected on a total of 37 test cases (see Table 5), including 4 new and 19 re-testable. Figure 11 shows the evaluation results based on the two criteria *ReductInteg* and *ReductUnit* for test suite reduction. The reduction of the test suite is significant for both integration (50%) and unit (38%) levels when compared to retest-all strategy. Figure 12 shows the evaluation results based on the two other criteria *ReuseInteg* and *ReuseUnit* for the reuse rate of test cases of our approach. The reuse rate of the test suite is here also significant for both integration (50.0%) and unit (89.2%) levels.

Table 5. Classification of Test Cases (Iteration 2)

	Total (V.2)	Obsolete	Retestable	Reusable	New	Total (V.3)
Unit	**34**	1	19	14	4	**37**
Integration	**17**	7	0	10	10	**20**

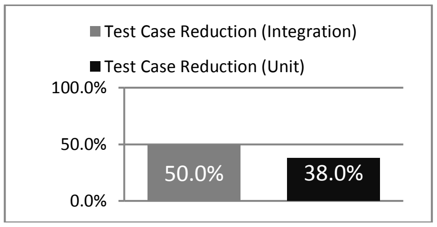


Figure 11. Test Suite Reduction

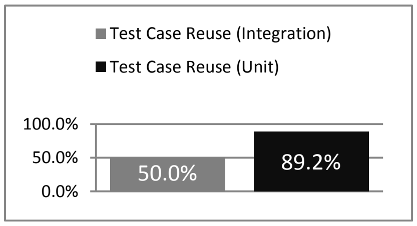


Figure 12. Test Case Reuse Rate

Iteration 3: From version V_3 to version V_4

At this iteration, for integration testing, there are 4 test cases (complete test sequences) selected on a total of 20. The 4 test cases correspond to existing scenarios where the source code of one or more methods has been modified. For unit tests, there are 15 unit test cases selected on a total of 37 test cases (see Table 6). If the same evaluation would have been done using only a model-based regression testing approach that does not consider changes to source code, then no test cases would be selected at all. As mentioned previously, our technique uses an impact analysis tool (Badri, 2005) to identify the list of modified methods. This list is then used to identify the impacted test sequences. In this iteration, the 4 selected test sequences contain a total of 15 methods that must be retested. Figure 13 shows the evaluation results based on the two criteria *ReductInteg* and *ReductUnit* for test suite reduction. The reduction of the test suite is significant for both integration (80%) and unit (59%) levels when compared to retest-all strategy. All test cases in this iteration are reusable since no test case was made obsolete and the complete test sequences are exactly the same as for version V_3. Figure 14 shows the evaluation results based on the two other criteria *ReuseInteg* and *ReuseUnit* for the reuse rate of test cases of our approach. The reuse rate in this iteration is equal to 100% for both integration and unit levels.

Table 6. Classification of Test Cases (Iteration 3)

	Total (V.2)	Obsolete	Retestable	Reusable	New	**Total (V.3)**
Unit	**37**	0	15	12	0	**37**
Integration	**20**	0	4	16	0	**20**

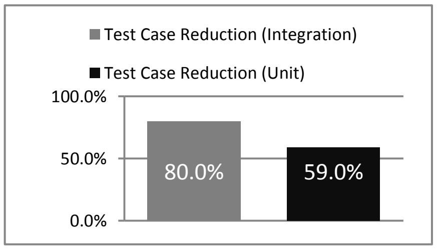


Figure 13. Test Suite Reduction

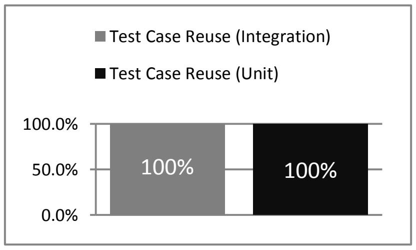


Figure 14. Test Case Reuse Rate

5. Conclusions and Future Work

We have presented a regression testing technique and associated tool for object-oriented systems. The technique combines, in fact, the analysis of UML models to a simple static analysis of the source code of the modified program. The UML models we used are use cases model and corresponding statechart and collaboration diagrams. The goal of the static analysis of the source code is to identify changes that are not visible in the design models.

The technique 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 identifies modified use cases (parts impacted by modifications) and selects the appropriate test cases from an existing test suite. New test cases are generated when necessary. In this way, the test suite is updated incrementally.

In order to evaluate the proposed technique, we used the tool we developed on a case study. We focused on the reduction of the cost of regression testing. We have made several changes to the different models of the original application to produce successive versions. We focused also on changes made on some methods in the source code of the application, which are not visible in the design models. Results provide evidence of the feasibility of the methodology and its ability to reduce the regression testing effort (reducing test suite size). The achieved results are, however, based on the data set we collected from only one subject system. The performed study should be replicated using many other OO software systems in order to draw more general conclusions. As future work, we plan to: extend the approach to more UML views, further explore the combination of model-based and code-based techniques in order to increase the accuracy of test case selection (and generation), use other criteria to improve the evaluation of the approach, and finally replicate the study on other OO software systems to be able to give generalized results.

Acknowledgements

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

References

[1] D. Kung, J. Gao and P. Hsia, “Class firewall, Test Order and Regression Testing of Object Oriented Programs”, Journal of Object Oriented Programming, **(1995)**, pp. 51-65.

[2] G. Rothermel, M. J. Harrold, “A safe, efficient regression test selection technique”, ACM Transactions on Software Engineering Methodology, vol. 6, no. 2, **(1997)** April, pp. 173-210.

[3] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon and A. Gujarathi, “Regression test selection for Java software”, Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, **(2001)**, pp. 312-326.

[4] G. Rothermel and M. J. Harrold, “Analyzing Regression Test Selection Techniques”, IEEE Transactions on Software Engineering, vol. 22, no. 8, **(1996)**, pp. 529-551.

[5] Y. Wu, M.-H. Chen and H. M. Kao, “Regression testing on object-oriented programs”, Proc. of the 10th International Symposium on Software Reliability Engineering, **(1999)**, pp. 270-279.

[6] O. Pilskalns, G. Uyan and A. Andrews, “Regression Testing UML Designs”, Proc. of the 22nd IEEE International Conference on Software Maintenance, ICSM '06'. **(2006)**, pp. 254-264.

[7] 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, **(2001)**, pp. 184-208.

[8] E. Engström, M. Skoglund and P. Runeson, “Empirical evaluations of regression test selection techniques: A systematic review”, Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. **(2008)**, pp. 22-31.

[9] M. J. Harrold, “Testing Evolving Software”, Journal of Systems and Software, vol. 47, **(1999)**, pp. 173-181.

[10] H. K. Leung and L. White, “A study of integration testing and software regression at the integration level”, Proceedings of the Conference on Software Maintenance, **(1990)**.

[11] J. Laski and W. Szermer, “Identification of program modifications and its applications in software maintenance”, Software Maintenance, **(1992)**.

[12] E. Engström, P. Runeson and M. Skoglund, “A systematic review on regression test selection techniques”, Information and Software Technology, **(2009)**.

[13] H. K. Leung and L. White, “Insights into testing and regression testing global variables”, Journal of Software Maintenance: Research and Practice, vol. 2, no. 4, **(1990)** December, pp. 209-222.

[14] H. K. Leung and L. White, "A Cost Model to Compare Regression Test Strategies", Proceedings of the Conference on Software Maintenance, vol. 91, **(1991)**, pp. 201-208.
[15] G. Rothermel and M. J. Harrold, "A comparison of regression test selection techniques", Technical report 114, Clemson University, Clemson, SC, **(1993)** April.
[16] D. Kung, "On regression testing of object-oriented programs", Journal of Systems and Software, vol. 32, **(1996)**, pp. 21-40.
[17] K. Abdullah, "The Firewall Concept for Regression Testing and Impact Analysis of Object Oriented Systems", PhD thesis, Case Western Reserve University, **(1998)**.
[18] A. Von Mayrhauser and N. Zhang, "Automated regression testing using DBT and Sleuth", Journal of Software Maintenance, vol. 11, no. 2, **(1999)** March, pp. 93-116.
[19] Y. Le Traon, T. Jéron, J. M. Jézéquel and P. Morel, "Efficient Object-Oriented Integration and Regression Testing", IEEE Transactions on Reliability, vol. 49, no. 1, **(2000)** March.
[20] G. Rothermel, M. J. Harrold, J. Dedhia, "Regression Test Selection for C++ Software", Journal of Software Testing Verification and Reliability, vol. 10, **(2000)**.
[21] L. C. Briand, Y. Labiche and G. Soccar, "Automating impact analysis and regression test selection based on UML designs", International Conference on Software Maintenance, **(2002)**, pp. 252-261.
[22] M. Skoglund and P. Runeson, "A case study of the class firewall regression test selection technique on a large scale distributed software system", Proc. of the ISESE, **(2005)**.
[23] L. White, K. Jaber and B. Robinson, "Utilization of extended firewall for object-oriented regression testing", Proc. of the IEEE International Conference on Software Maintenance, **(2005)**, pp. 695-698.
[24] N. Mansour, H. Takkoush and A. Nehme, "UML-based regression testing for OO software", Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, **(2011)**, pp. 51-68.
[25] Y. Chen, R. L. Probert and D. P. Sims, "Specification-based Regression Test Selection with Risk Analysis", IBM Center Advanced Studies Conference, **(2002)**.
[26] M. Fahad and A. Nadeem, "A survey of UML Based Regression Testing", IFIP International Federation for Information Processing, Intelligent Information Processing IV; Zhongzhi Shi, E. Mercier-Laurent, D. Leake; (Boston: Springer), vol. 288, **(2008)**, pp. 200-210.
[27] L.C. Briand, Y. Labiche and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software", IEEE Transactions on Software Engineering, **(2006)**, pp. 642-663.
[28] Y. Wu and J. Offutt, "Maintaining evolving component based software with UML", Proceedings of the 7th European Conference on Software Maintenance and Reengineering, **(2003)**.
[29] C. Larman, "Applying UML and Design Patterns", An introduction to object-oriented analysis and design and the unified process, Prentice Hall, **(2004)**.
[30] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications", Second International Conference on the Unified Modeling Language (UML '99). Fort Collins, CO, **(1999)** October.
[31] M. Badri, L. Badri and N. Naha, "A Use Case Driven Testing Process: Towards a formal approach based on UML collaboration diagrams", Post-Proceedings (selected and revised papers) of FATES *(Formal Approaches to Testing of Software) 2003*, in LNCS (Lecture Notes in Computer Science) 2931, Springer-Verlag, **(2003)**.
[32] L. Badri, M. Badri, D. St-Yves, "Supporting predictive change impact analysis: A control call graph based technique", Proc. of the 12th Asia-Pacific Software Engineering Conference, **(2005)**, pp. 167-175.
[33] P. Massicotte, L. Badri and M. Badri, "Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs", Journal of Object Technology, vol. 6, no. 1, **(2007)**, pp. 671- 89.
[34] M. Badri, L. Badri and M. Bourque-Fortin, "Automated State-Based Unit Testing for Aspect-Oriented Programs: A Supporting Framework", Journal of Object Technology, vol. 8, no. 3, **(2009)**, pp. 121-126.