JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2009; **21**:253–286
Published online 14 April 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.403

Research

Regression testing with UML software designs: A survey

Alireza Mahdian[1], Anneliese Amschler Andrews[1,*,†] and Orest Jacob Pilskalns[2]

[1]*Department of Computer Science, University of Denver, 2360, S. Gaylord St., Denver, CO 80208, U.S.A.*
[2]*School of Engineering and Computer Science, Washington State University Vancouver, 14204 NE Salmon Creek Avenue, Vancouver, WA 98686, U.S.A.*

SUMMARY

The unified modeling language (UML) designs come in a variety of different notations. UML designs can be quite large and interactions between various notations and the models they define can be difficult to assess. During the design phase, and between successive releases of systems, designs change. The impact of such changes and the resulting effect on behavior can be non-obvious and difficult to assess. This survey article explores techniques for such re-evaluation that can be classified as regression testing and suggests regression testing criteria for designs. These might vary depending on testing objectives and include both selective and regenerative regression testing approaches. The article provides a concise overview of regression testing approaches related to various UML design notations including use cases, class diagrams, sequence diagrams, activity diagrams, and statecharts, as well as combinations of these models. It discusses UML-related techniques involving cost and prioritization during selective regression testing. Finally, it evaluates these techniques with respect to inclusiveness, precision, efficiency, generality, accountability, and safety. An example is used throughout to illustrate how the techniques work. Copyright © 2009 John Wiley & Sons, Ltd.

Received 28 October 2008; Revised 29 December 2008; Accepted 7 January 2009

KEY WORDS: design testing; software testing; regression testing

1. INTRODUCTION

The unified modeling language (UML) [1] has become a defacto industry standard. Traditionally, evaluation of UML designs occurs during design inspections and reviews. More recently, UML testing techniques have been developed that either use UML to derive test cases to test code [2–8], or test the design itself [9–12]. Given that designs tend to change, effects of such changes are

*Correspondence to: Anneliese Amschler Andrews, Department of Computer Science, University of Denver, 2360, S. Gaylord St., Denver, CO 80208, U.S.A.
†E-mail: andrews@cs.du.edu

Table I. Classification of existing work on software testing.

Design methods	Non-UML-based	With UML	For UML
Regression testing	[13–29]	[15,30–32]	[33]

often difficult to assess, especially for large and complex designs with multiple notations. Designs not only change during initial development, but also during maintenance and evolution. Whenever changes occur, it becomes important to evaluate the effect of such change and to make sure that the modifications have not introduced errors. At this stage, existing work on regression testing techniques for UML designs or with UML designs is limited. This article provides a survey of techniques that either suggest regression testing techniques that use UML artifacts, or suggest ways in which to regression test design modifications themselves. Our motivation was to provide a concise overview of these techniques so that students, professionals, and researchers can assess existing work in the area, identify, and compare potential solutions to UML regression testing tasks, and see where further work is necessary and fruitful. Table I classifies existing regression testing techniques into three categories:

1. Non-UML-based regression testing techniques for designs that have informed or are being used or adapted in UML-based regression testing.
2. Regression testing techniques that use UML to select regression tests for code.
3. Regression testing techniques that test modifications of UML designs themselves.

Section 2 reviews relevant principles of regression testing code that can be adapted to or are relevant to testing designs, including non-UML-based design reevaluation approaches. Section 3 describes an example design and the changes to it that is used throughout the survey article to illustrate the various approaches. Section 4 describes regression testing techniques that use UML diagrams. These techniques have been defined for activity diagrams, statecharts, and multiple diagrams and their interactions. Section 5 describes a regression testing technique to evaluate UML design changes (DCs) for both structural and behavioral UML models. Section 6 contrasts and evaluates the approaches and points to further work. Section 7 draws conclusions.

2. PRINCIPLES OF REGRESSION TESTING

2.1. Selective vs retest-all strategies

The process of testing software modifications is called regression testing. It involves retesting all or part of a software system after it is modified. *Retest-all* and *selective* regression testing are two basic strategies for regression testing. The set of tests running during a regression test is the regression test suite [27].

In retest-all strategy, the assumption is that changes could have affected any part of the code; therefore, errors may be introduced anywhere in the code; hence, the system needs to be tested all over again. On the other hand a selective regression testing strategy assumes that only parts of the software could have been affected by modifications [27]; thus, a subset of the original test suite needs to be selected for regression testing. The main objective in selective regression testing is to

DOI: 10.1002/smr

reduce the time and effort of regression testing while maintaining the efficacy of the test suite in revealing faults.

For either strategy, the tester may need to develop new tests to exercise new or modified features of the software that are not covered by existing tests. Rothermel and Harrold [34] provided a definition of selective regression testing consisting of the following steps:

1. *Identify changes.*
2. *Determine which of the currently existing test cases will remain valid for the new version of the software* (*eliminate all tests that are no longer applicable—this results in a set of tests* T'*, a subset of the original test suite* T).
3. *Test the modified software with* T'.
4. *If necessary, test parts of the software that are not tested adequately with* T' *by generating new test cases* T''. *What it means to test adequately depends on the test criterion chosen.*
5. *Execute the modified software with* T''.

Steps 2 and 3 test whether the modifications have introduced defects. Steps 4 and 5 test whether the modifications work.

T' is determined based on the classification of existing test cases and the regression testing approach. Leung and White [23] classified existing test cases as *reusable*, *retestable*, and *obsolete*. Reusable test cases test unmodified parts of the specification and the corresponding unmodified code. Retestable test cases test modified code against an unmodified specification. Retestable test cases specify the correct input and output relation, but they execute changed code. Obsolete test cases are those that are no longer valid.

All regression testing strategies fall on a spectrum: on one end is reuse-based selective regression testing and on the other end is regeneration-based regression testing. The number of test cases from each class is determined based on where a regression testing technique falls on the spectrum. A reuse-based selective regression testing technique selects retestable test cases from the existing test suite. A regeneration-based retest-all approach produces test cases from scratch and is equivalent to development testing [27].

In practice there are many factors that influence the selection of a regression testing strategy. Table II describes the types of situations that may call for different regression testing approaches. We distinguish between four different approaches to regression testing. Table III shows their corresponding strategies. They range from minimal selective regression testing to retest-all regeneration-based regression testing.

With the exception of retest-all and regeneration-based regression testing techniques(full new test cycle), all regression testing techniques employ some kind of selective testing strategy. Selective

Table II. Regression testing in various situations [27].

	Situation descriptors			
Approach	Confidence in software quality	Change nature	Change impact	Schedule constraints
Full new test cycle	Low	Key code	Extensive	None
Minimalistic regression test	High	Isolated	Localized	Very tight
Expanded scope of regression test	Moderate	Key code	Extensive	Moderate
Full reuse of existing test suite	Low	Series of fixes	Extensive	Moderate

DOI: 10.1002/smr

Table III. Regression testing approaches [27].

Approaches	Strategies
(1) Full new test cycle	• Redesign entire test plan • Rebuild the whole test suite
(2) Minimum regression test	• Reuse test plan as much as possible • Rerun minimum number of retestable test cases • Generate minimum number of test cases for changes
(3) Expanded scope of regression testing	• Reuse part of test plan • Rerun all retestable test cases • Generate new test cases on the full scope of changes
(4) Full reuse of existing test suite	• Reuse the test plan and test suite

testing strategies can be classified as *Minimization*, *Coverage*, and *Safe* approaches [26]. This classification is based on (1) how parts of the software that have been affected by changes are identified and (2) to what extent and how they have to be retested. Minimization testing techniques have been categorized as follows:

- Selecting minimal number of test cases in terms of test coverage with regard to testing criteria.
- Selecting test cases that would minimize cost while satisfying coverage.
- Selects test cases by prioritizing test sequence in terms of effectiveness (fault detection) [16,20,22,24,28], minimal cost [15,32], or both. For a survey of these techniques refer to [22].

According to Rothermel and Harrold [26], coverage approaches [23,25,27,29,30,33,35–37] are based on meeting specific coverage criteria, not necessarily minimization of the test suite. They aim to rerun tests that likely produce different output. Coverage criteria are used to select such tests. By contrast, safe approaches select all tests that will produce different output than the original program [26].

Rothermel and Harrold [26] outline issues relevant to selective retest approaches, and present a framework within which such approaches can be evaluated. This framework is then used to evaluate and compare several of the existing selective retest algorithms. The original framework analyzes each test selection method based on five properties: *Inclusiveness*, *Precision*, *Efficiency*, *Generality*, and *Accountability*. We extend this framework by adding *Safety* as the sixth property of the framework. The first three measures are quantitative. Generality and accountability are qualitative.

Before we define each measure we adapt some notations from [26]. Let S denote a selective retest strategy, T the initial test set and T' the set of tests selected by using S, and P and P' the original and modified programs, respectively. Inclusiveness measures the extent to which a method chooses tests that will cause the modified program to produce different output.

Inclusiveness can be quantified using the following definition from [26]:

Definition 1. Suppose T contains n modification-revealing tests, and S selects m of these tests. The *inclusiveness* of S relative to P, P', and T is the percentage calculated by the expression $((m/n)*100)$.

Safety is defined in terms of inclusiveness as follows [26]: 'if a regression test strategy S results in 100% inclusiveness it is considered safe.' Precision measures the ability of a method to avoid

DOI: 10.1002/smr

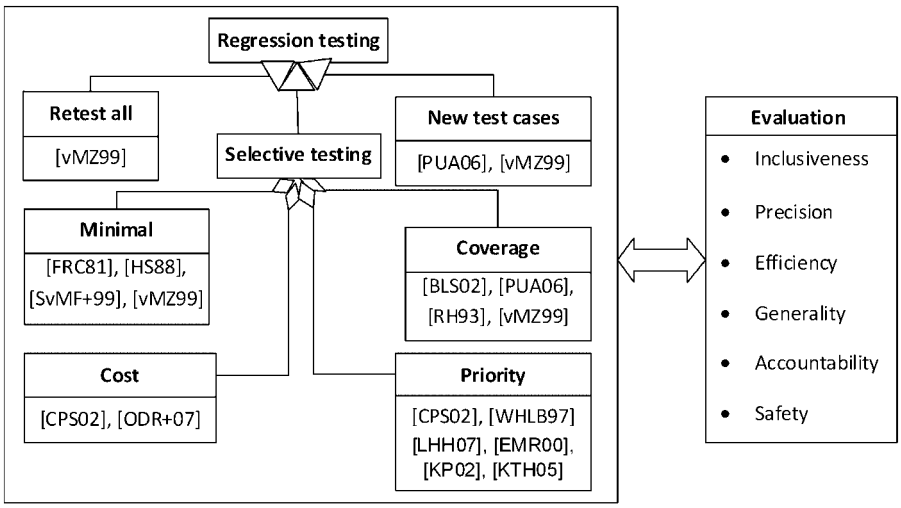


Figure 1. Regression testing domain.

choosing tests that will not cause the modified program to produce different output. Using the notations introduced earlier, precision is defined in [26] as:

Definition 2. Suppose T contains n non-modification-revealing tests, and S omits m of these tests. The $precision$ of S relative to P, P', and T is the percentage calculated by the expression $((m/n)*100)$.

Efficiency in [26] measures the relative effort and thus the cost of regression testing of a selective retest approach. Thus in general a selection strategy is considered efficient, if the cost of running T' is less than the cost of running $T - T'$.

Rothermel and Harrold [26] introduce generality to measure the 'ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications'. It is considered as a qualitative measure. Finally, accountability measures a method's support for coverage criteria, that is, the extent to which the method can aid in the evaluation of test suite adequacy [26].

Although this framework is based largely on qualitative comparisons, it provides a way to evaluate and compare existing selective retest approaches. We use it in Section 6 to evaluate UML-based regression testing techniques.

Figure 1 gives an overview of the regression testing domain and the associated research. Most of the regression testing research has focused on testing code as opposed to design. However, since code is based on design, the general principles mentioned above can be applied to testing designs as well.

2.2. Design-based regression testing

Most of the regression test selection strategies are code-based (i.e., they select the test cases based on code control flow and data flow analysis). Although many of these techniques show very good

DOI: 10.1002/smr

performance during unit testing, they may lack the scalability needed for system testing when they use traceability techniques (e.g., traceability table) to relate test cases to code statements. As the program becomes larger, constructing and maintaining traceability tables become more and more difficult. Another problem with code-based regression testing is the code knowledge required to select the test cases. This can be a time-consuming process. It also requires access to the code that is not always possible (e.g., a system composed of Commercial Off The Shelf (COTS)). Finally, most of the code-based regression testing techniques are language dependent and it becomes more difficult to use them if the program is written in multiple programming languages.

An alternative approach is to use architectural/design information available in design models [27]. In this case, selected test cases execute new or modified model elements or model elements formerly executed but deleted from the original version. The impact of possible changes is first assessed on the design of the last version of the system, by comparing the modified design with the existing design. This forms the basis for regression test selection. The main advantages of a design-based approach are efficiency and the possibility of performing early test planning and effort estimation [30].

There is a whole body of research that uses models other than UML to select regression tests to test code (e.g., [14,21,27]). This survey restricts itself to UML approaches that either test the design itself or use UML to identify regression tests of code. We describe [27] not so much because it is a model-based technique, but because some of the UML-based techniques use concepts first presented in [27]. Von Mayrhauser and Zhang [27] proposed one of the earliest techniques that employ design information to select and regenerate regression test cases. This technique uses the domain model of the system under test to identify changes in the new version of the system. The domain model is the result of application domain analysis. It is similar to a conceptual model in more modern UML notation and combines some structural and dynamic information like one of the UML-based techniques described later. It is more limited than the UML notation.

According to Von Mayrhauser and Zhang [27], domain analysis has three consecutive steps:

1. Object analysis: This step results in object hierarchy definition, which includes the relations between objects and the definition of each object and its elements.
2. Constraint analysis: This step includes the definition of all the constraints on objects and their parameters.
3. Scenario definition: At this step, the sequence of activities that comprise the application of the software are defined.

The approach consists of two phases:

- *Phase* 1 determines what should be regression tested by building a regression subdomain.
- *Phase* 2 takes the regression test subdomain and generates new tests for it automatically, using a tool called *Sleuth*.

Sleuth can be used to select test cases from the existing test suite in case of a selective reuse strategy or it can be used in building a subdomain based on domain model changes and generating test cases in case of a regeneration strategy.

Although this approach is based on a conceptual model, it does not use the standard UML diagrams. The more recent design-based techniques employ UML diagrams to identify changes in the old and new versions of the design and hence selecting and/or generating test cases. In the

DOI: 10.1002/smr

following sections, we introduce these more recent techniques. For better illustration, each technique is demonstrated on our general example design that is introduced in the next section.

3. EXAMPLE DESIGN

As we introduce each regression testing technique, we apply it to an example design of an automatic teller machine (ATM). For simplicity, we only assume three functionalities for the ATM (c.f. Figure 2): users can withdraw cash, get a print of their statements, or deposit cash.

The design deliberately contains several defects. We identify each problem as we apply a relevant testing technique to a UML diagram. The techniques described in Sections 4 and 5 use single or multiple UML diagrams to derive regression test cases. These techniques use UML usecase diagrams, class diagrams, sequence diagrams, activity diagrams, and statecharts. Thus, we include one of each of these diagrams in our example. We start with the class diagram for the example ATM. Figure 3 shows the class diagram for the ATM design. It consists of four classes:

- Controller class: This class is in charge of user authentication and all the other functionalities of the ATM. It also interacts with the bank system through an interface and ensures that a user account is consistent after each transaction.
- Terminal: Terminal works as an interface between the user and the Controller class. The user interacts with the ATM through a simple display and a key pad. In general, users select their transactions through the terminal; in return, the terminal displays the result of user interactions with the ATM.
- Cash dispenser: It is responsible for both accepting deposits and dispensing cash.
- Printer: The printer class is an interface to a printer device that prints statements. For simplicity, we assume printStatement is the only function defined for the printer class that prints a user account statement.
- Card reader: The Card Reader accepts, reads, and returns the ATM card to a user.

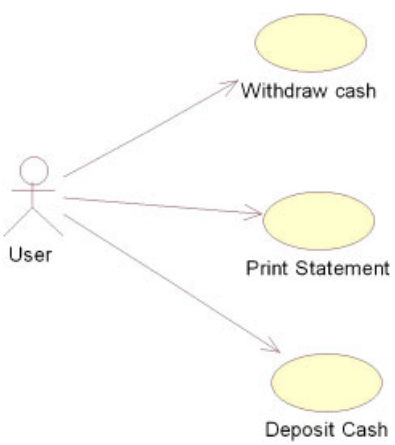


Figure 2. Use case diagram of the example ATM.

DOI: 10.1002/smr

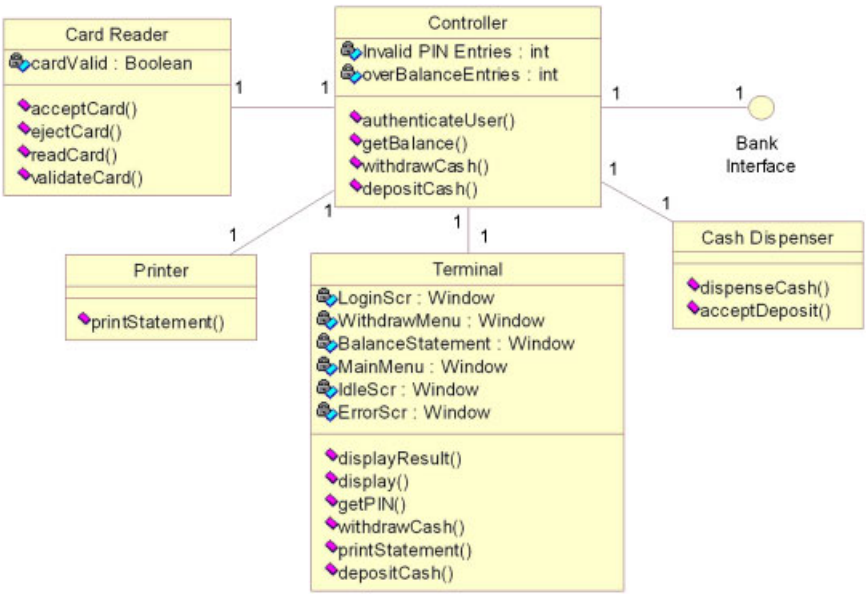


Figure 3. Class diagram for the example ATM.

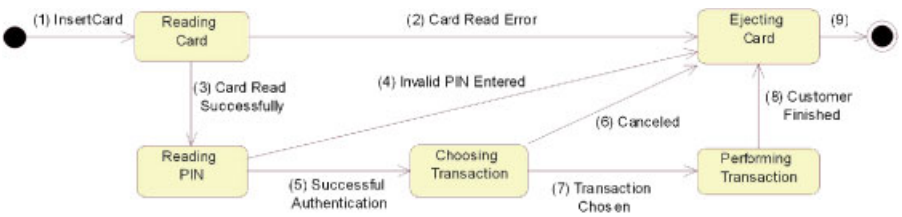


Figure 4. Statechart diagram for the example ATM.

Second, we present statecharts for the example ATM. The ATM can be in one of five states. Figure 4 represents these states and their interactions with each other through the statechart diagram.

Next, we define a sequence diagram for the ATM example. Figure 5 illustrates the Withdraw Cash transaction by means of a sequence diagram. We only consider basic flow. Exception handling is omitted for the sake of simplicity. Finally, Figure 6 demonstrates the basic interactions within the Controller class by means of the UML activity diagram.

The following changes are proposed to improve system functionality:

- Operational errors are to be written into an error log.
- As a security measure, all invalid PIN entries are recorded in the security log.
- After a sequence of three invalid PIN entries, the ATM card is retained in the ATM.
- The users are allowed to cancel their transaction and obtain their card after successful authentication.

DOI: 10.1002/smr

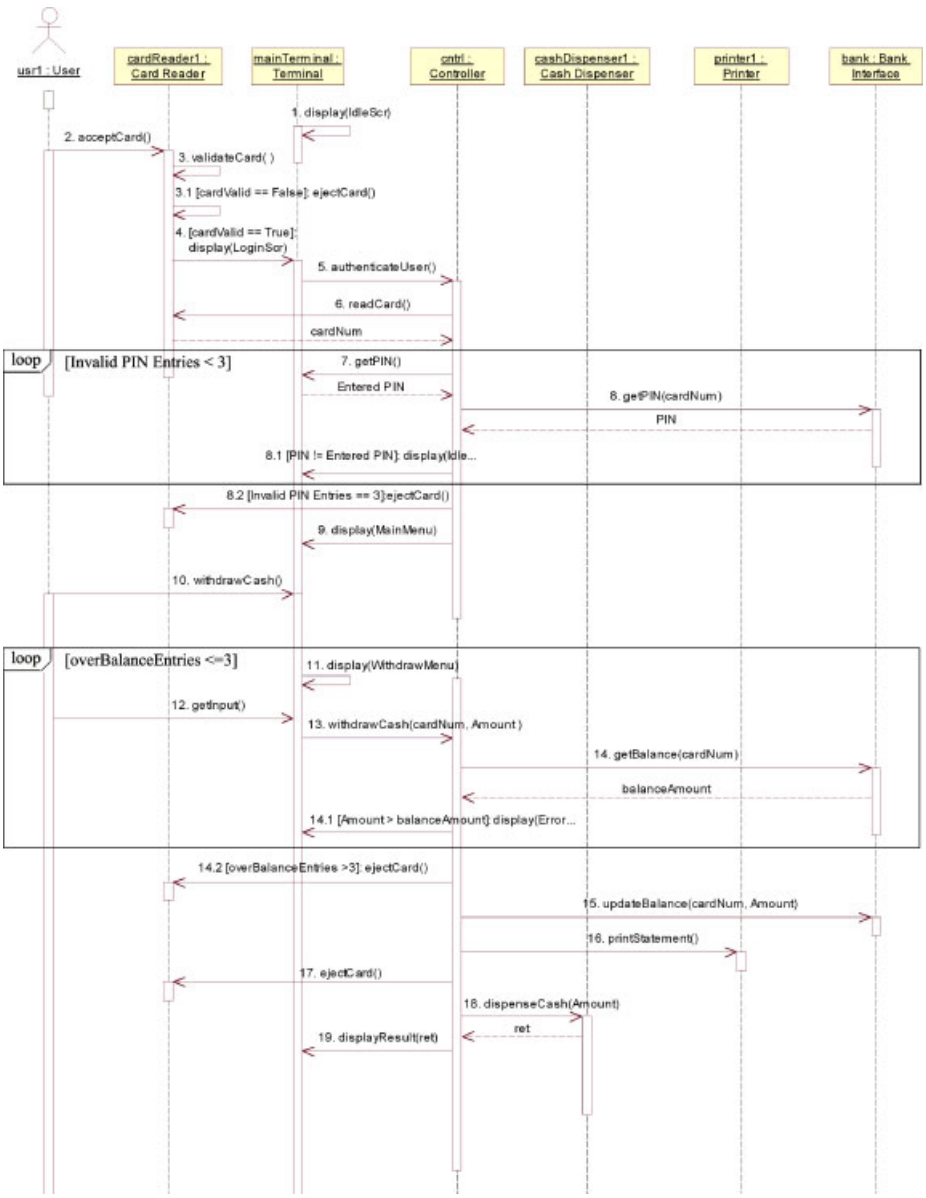


Figure 5. Sequence diagram for the Withdraw Cash transaction.

The design includes two faults that need to be fixed:

- In the activity diagram of the Controller class, the transition from Withdraw Cash activity to Get Balance activity is considered as a fault, since the balance needs to be checked to ensure that there is enough balance before any withdraw transaction.

DOI: 10.1002/smr

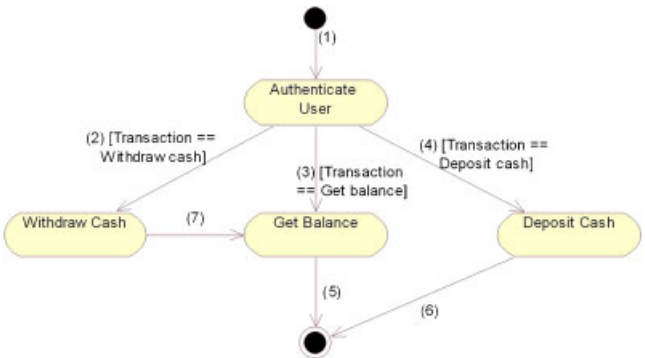


Figure 6. Activity diagram for the Controller class.

- The sequence diagram of the Withdraw Cash transaction also includes a fault that results in an inconsistent account balance in case of an unsuccessful withdraw transaction.

As a result the design is modified. We demonstrate each of the regression testing techniques on this modified design.

4. REGRESSION TESTING TECHNIQUES THAT USE UML DIAGRAMS

In this section we introduce techniques that use design artifacts in the form of UML diagrams to select(derive) test cases for regression testing the code. Note that the techniques introduced in this section select (generate) test cases that execute program code. Hence the implementation still needs to be available. We classify each technique based on the UML diagram used. We illustrate each technique using the ATM example.

4.1. Activity diagram

Chen *et al.* [15] classify regression tests as either *Targeted* or *Safety* tests. Targeted tests are the ones that would test the modified parts of the program to ensure correct functionality. Safety tests are risk oriented. The main motivation for safety tests is the fact that changes in some parts of the code might not be documented and potential defects in those parts might not be detected by targeted tests. Hence safety tests focus on key functions and exception handling to avoid this situation.

Chen *et al.* [15] present a similar approach to the code-based approach of Rothermel *et al.* [38] to select targeted tests. In [38] the control flow graph (CFG) of the original and modified programs are compared. The affected edges are identified. The test cases that traverse the affected edges are identified from the traceability table and are selected as the regression test suite. Chen *et al.* [15] use a UML activity diagram instead of the CFG to select the test cases.

They identify two classes of changes. The first consists of code changes that do not affect the system behavior. These changes cannot be identified through the activity diagram. Rather, the code

DOI: 10.1002/smr

change history maintained by the programmers is used to identify all nodes (i.e., activities) in the activity diagram whose implementation has been changed.

The second group consists of changes that affect system behavior. These changes represent themselves as added/deleted nodes and edges in the activity diagram. The nodes and edges that represent any of the changes described above are considered as affected nodes and edges. The affected edges are identified by comparing the original and modified activity diagrams. In addition to added/deleted edges, all edges that point to an affected node are also considered as affected edges. Finally, based on a traceability table, test cases that traverse affected edges are selected as targeted tests.

Chen *et al.* also introduce three steps to select safety tests. The steps include:

1. Compute the cost of each test case. Two aspects are considered to compute the costs. One is the negative effect of a fault on customer perception of product quality that could result in losing market share. The other is the maintenance cost. Cost is categorized on an one to five scale.
2. Derive a severity index for each test case that is a number between zero and five that indicates the seriousness of the defect.
3. Calculate risk exposure as severity index multiplied by cost.
4. Select those test cases with the highest risk exposure value and execute them on the modified program.

Martins and Vieira [31] propose an approach to select test cases for regression testing a class. This approach is similar to [15]; in that both approaches select test cases based on control flow analysis and use the activity diagram as the main artifact. The main difference is that Martins *et al.'s* approach focuses on a single class instead of a complete system. The approach has four basic steps:

1. Construct a traceability table between test cases and the elements of the behavioral model.
2. Perform a change analysis comparing old and new behavioral models.
3. Identify affected elements of the behavioral model.
4. Select test cases to execute on the new program.

Before performing the above steps, the behavioral model needs to be derived from the activity diagram. The behavioral model is a directed acyclic[‡] graph called behavioral control flow graph (BCFG). The BCFG is constructed from the activity diagram by creating a vertex for each activity or sub-activity and a directed edge for each transition. In addition, there are unique vertices for entry (start) and exit points of the activity diagram. Each vertex has a label composed of the signature (i.e., method name, arguments' names and types, and return value type) for the method that implements the corresponding activity. A sub-activity is considered as a method without any parameters. Entry and exit vertices have labels '*start*' and '*exit*', respectively. The label of vertex v is denoted by $S(v)$.

Edges that correspond to conditional transitions are labeled as `[condition]` or `[else]` according to the corresponding guard conditions in the activity diagram. An edge is represented as $e = (v, w, l)$, where v is the source vertex, w is the target vertex, and l is the edge label. A path with

[‡]Loops are not addressed in this work.

DOI: 10.1002/smr

length n is represented as $p=[e_1,\ldots,e_n]$. A complete path in the BCFG is a sequence of edges that connects the start vertex to the exit vertex. Although a test case always traverses a complete path, a complete path may be traversed by several test cases.

Martins *et al.* suggest the use of built-in test (BIT) capabilities to trace test-cases back to the vertices and edges of the BCFG. More specifically, BIT capabilities are used to determine whether a BCFG path is traversed by a test case or not. BIT capabilities include the following features:

- Assertions that check pre/post conditions and invariants at runtime.
- A method that reports the state of the object.
- An access control mechanism that prevents misuse of the BIT features.

The changes in the activity diagram are classified as either *contract* or *implementation* changes. A contract change is a consequence of a requirement or DC. Examples of contract changes include addition or removal of interface methods, alteration of method signature or changes in the pre/post conditions or invariants. Implementation changes are changes to a class that are not visible to its clients (they do not alter the external behavior of the class). As for [15] there are two types of changes in terms of detectability: first, those that can be detected from the activity diagram, e.g., alteration of a method signature; second, changes that cannot be detected from the activity diagram, e.g., changes in pre/post conditions. The assertion feature of the BIT helps to detect changes of the second type in the vertices of BCFG. Let $V^c \subset V$ denote all the vertices in the activity diagram that are affected by the second type of change. Martins *et al.* propose an algorithm that detects all affected edges.

The algorithm runs on new and old versions of the activity diagram in parallel starting from the vertex labeled 'start'. It traverses the BCFG depth first and marks all edges that are affected by changes. The edges are classified into two sets: D_e (deleted edges) and R_e (retestable edges). Deleted edges are either not present in the new version of the activity diagram or point to a vertex with a changed label. Retestable edges point to vertices that are in the set V^c. This algorithm does not address new edges that only exist in the new version of the activity diagram as this requires new test cases. The algorithm only *selects* test cases from the existing test suite.

Finally, test cases are classified into three mutually exclusive sets: *obsolete*, *retestable*, and *reusable*. Obsolete test cases are those whose path includes at least one edge that is a member of D_e. Retestable test cases are those whose path includes at least one edge that is a member of R_e. All other test cases are considered reusable. To illustrate how the algorithm works, we will execute this algorithm on our example ATM.

We illustrate the techniques in [15,31] to select regression test cases for the activity diagram of the Controller class. Figure 7 represents the new activity diagram of the Controller class. As a result of the changes described in Section 3, there is a transition between the Authenticate User activity and Write Security Log activity, which is added in the new activity diagram in case of invalid PIN entries. Further, the transition between the Withdraw Cash activity and Get Balance activity is removed. The first fault[§] leads to a change in the code of the Withdraw Cash activity.

Figure 8 shows the BCFG of the old and new activity diagrams for the Controller class. Table IV represents the test cases for the original activity diagram. Each test case is composed of a sequence of activities that start at the initial state and end in the final state of the activity diagram. Traceability

[§]Refer to the example section.

DOI: 10.1002/smr

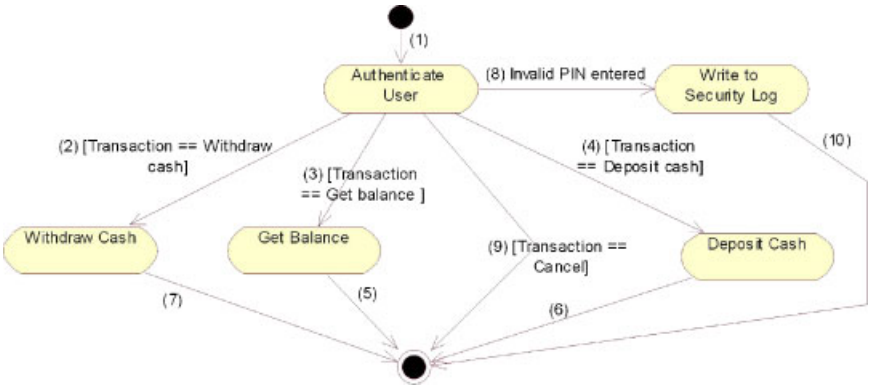


Figure 7. The new activity diagram for the Controller class.

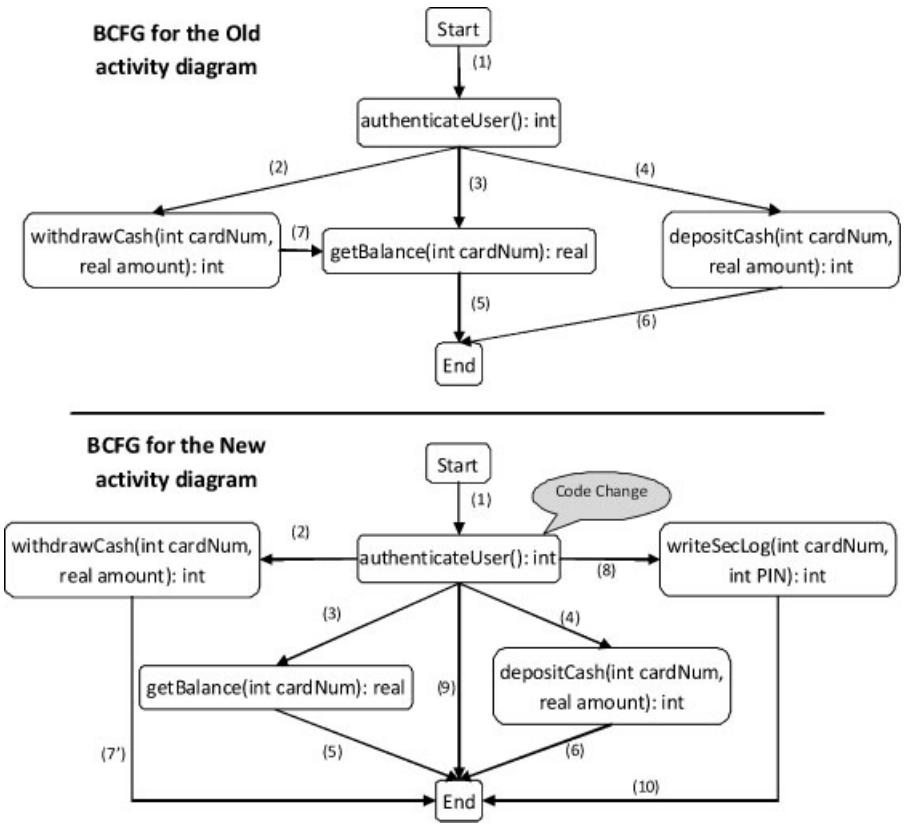


Figure 8. Corresponding BCFGs for the old and new activity diagrams.

DOI: 10.1002/smr

Table IV. Test cases for the old activity diagram of the Controller class.

TC #	Test case	Traversed edges	Severity index	Cost	Risk exposure
1	authenticateUser, withdrawCash, getBalance	(1), (2), (7), (5)	5	5	25
2	authenticateUser, getBalance	(1), (3), (5)	3	3	9
3	authenticateUser, depositCash	(1), (4), (6)	5	4	20

information is provided in the third column of Table IV by denoting the edges that are traversed by each test case. In addition, risk exposure is computed in the last column as the multiplication of the severity index (column 4) and the cost (column 5). Since an error in a cash withdrawal or deposit transaction is considered as more serious than an error in a print balance statement transaction; we give a higher severity index to test cases 1 and 3 with respect to test case 2. In addition, because a cash withdrawal transaction is considered as more complex than the others, the cost of fixing an error in the Withdraw Cash transaction is more than the cost of fixing an error in deposit cash or print statement transactions.

After applying the techniques in [15,31], test cases 2 and 3 are considered retestable (due to the change in the code of function authenticateUser) and the first test case is obsolete since the edge labeled (7) is removed from the old BCFG. Furthermore, test case 3 is favored over test case 2 because of its higher risk exposure value.

4.2. Using statecharts

Orso *et al.* [32] present an approach similar to [15,31] for regression test selection, but they use statechart diagrams instead. This approach has two basic steps:

1. Identifying the differences between the new and old versions of the statechart.
2. Selecting test cases that exercise the changed sections.

The first step uses an adapted version of an algorithm proposed in [25] to find changed sections. The algorithm performs a pairwise synchronous walk of the two versions of the statechart diagrams starting from the initial state. States reached along identically labeled outgoing transitions are compared with each other until changes are found. The changes can be classified as *state* changes, i.e., added/deleted states or *transition* changes, i.e., added/deleted transitions. A transition with changed labels is considered as the result of two changes: (1) a deleted transition from the old version and (2) an added transition. In the next step, all transitions that lead to a changed state are marked as '*dangerous transitions*'. In addition, the changed transitions are marked as '*obsolete transitions*'. In the process of marking, 'obsolete' transitions have precedence over 'dangerous' ones, i.e., if a transition is considered both dangerous and obsolete, it is obsolete.

Each existing test case traverses a path in the statechart diagram. This path is stored in a test coverage table for traceability purposes. For regression test selection, test cases that have at least one obsolete transition in their path are considered as obsolete. From the remaining test cases, those with at least one dangerous transition in their path are considered as retestable. The rest are reusable.

A problem with this technique is that it can be applied to only one statechart diagram. In case of a system with interacting components each with a different statechart, this technique would not be

DOI: 10.1002/smr

Table V. Test cases for the old statechart of the ATM design.

TC #	Test case	Traversed transitions
1	insert card, card read error.	(1), (2), (9)
2	insert card, card read successfully, invalid PIN entered.	(1), (3), (4), (9)
3	insert card, card read successfully, successful authentication, canceled.	(1), (3), (5), (6), (9)
4	insert card, card read successfully, successful authentication, transaction chosen, customer finished.	(1), (3), (5), (7), (8), (9)

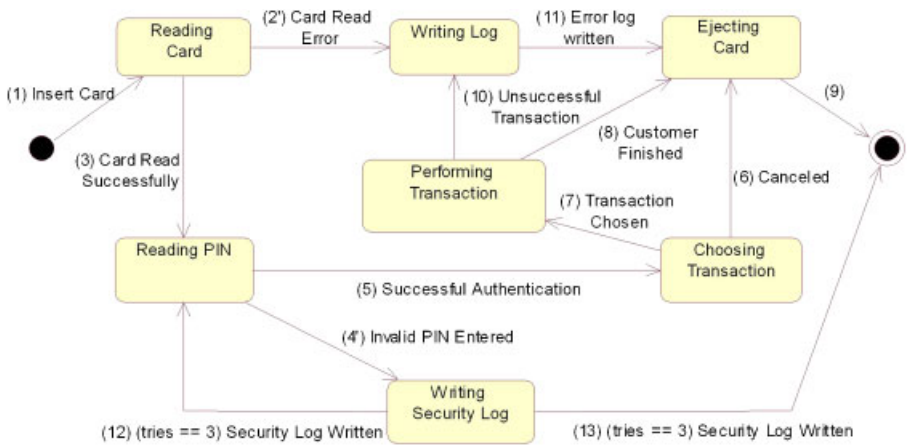


Figure 9. The new statechart diagram for the example ATM.

practical. For this, Orso *et al.* [32] suggest the use of the *Incremental Composition and Reduction* (*ICR*) method proposed by Sabnani *et al.* [39] to construct a global statechart of the system. The ICR method addresses the state explosion problem by avoiding all unreachable states at the composition step and removing all replicated states from the global statechart diagram. If statechart diagrams for components are provided by the vendor, perhaps in the form of '*metadata*' as suggested in [32], this approach could be used to select regression test cases for component-based software.

Table V represents the test cases for the statechart of Figure 4. Each test case is defined as a sequence of events that cause statechart traversal from the start to the end state. In addition, traceability information is embedded in Table V in the form of transitions that are traversed by each test case.

The statechart of the upgraded ATM design is shown in Figure 9. As for the original statechart, the transitions are labeled with numbers 1–13. The differences introduced in the new statechart include the addition of 'Writing log', and 'Writing Security Log' states and four transitions labeled 10–13. The 'Performing Transaction' state is modified so that unsuccessful transactions are written in the log file. Similarly, the 'Reading PIN' state is modified to support writing to the security log. Note that although the 'Invalid PIN Entered' transition is present in the original statechart, because its target state is changed in the new statechart, it is considered as a changed transition (removed

DOI: 10.1002/smr

from the original statechart and added to the new statechart). For this, 'Invalid PIN Entered' is labeled with '4′' to underline this change. Similarly, the 'Card Read Error' transition (labeled with '2′') is a changed transition.

Because these changes are considered minor modifications of the original statechart of Figure 4, we can use the technique in [32] to select some of the existing test cases to test the modified statechart diagram. As mentioned earlier, the 'Performing Transaction' state is a changed state and hence all edges that lead to this state are identified as *dangerous* transitions. Hence, the transition labeled (7) is a dangerous transition. Similarly, 'Reading PIN' is a modified state. Thus, the transition labeled (3) is considered as a *dangerous* transition. In addition, 'Card Read Error' and 'Invalid PIN Entered' transitions labeled (2′) and (4′), respectively, in the new diagram are changed transitions and thus, considered obsolete. Based on [32], test cases 1 and 2 are obsolete and 3 and 4 are retestable.

4.3. Using multiple diagrams

Briand *et al.* [30] use architectural/design information to guide test selection. In this approach the impact of the change is first assessed on the design itself and, depending on its magnitude, a change management group decides whether to implement it in the next version or not.

Change impact analysis in the design phase has several advantages including efficiency, early regression test planning, and early effort estimation. Efficiency is achieved since complex static and dynamic code analyses are avoided and traceability of test case to design is much easier compared with code. Another advantage of design level change analysis is that regression testing tools can be programming language independent. However, design level change analysis requires the design to be complete and up to date [30].

Briand *et al.* [30] propose an approach to select regression test cases for functional and non-functional system testing of code, based on architectural/design artifacts represented in the form of UML diagrams. First the changes are identified between two versions of the design. For this purpose, three types of diagrams are considered: class, sequence, and use case diagrams.

Briand *et al.* categorize changes based on the associated diagrams. Each change category is further classified and for each class of change in each category, mathematical formalization is provided using set operations. Each identified change is assigned to a specific change class based on the diagram that reflected the change and the nature of the change. Because it is assumed that traceability between test cases and design is provided, each change can be related to a test case. Each test case is then categorized into mutually exclusive sets of *obsolete*, *reusable*, or *retestable* test cases.

Change categories are *adding*, *deleting*, or *changing* an artifact or part of a UML artifact in successive versions of the design. The following assumptions are made:

- Class names are unique.
- Changed classes and class names are considered to be deleted classes followed by adding the modified class (deleted/added class).
- Attribute changes show differences in scope, type, or visibility.
- A method's signature is unique (i.e., method name, parameters, and their type in order, return type, visibility, scope).
- A changed method signature is considered as a method delete, followed by a method addition with the modified signature.

DOI: 10.1002/smr

Each change category corresponds to a specific kind of UML diagram, i.e., changes between two versions of the same class diagram, sequence diagram, or use case diagram. Briand *et al.* further classify changes between two versions of the same class diagram into [30] as follows:

1. Added/Deleted attribute: if R^a_{AC} and R^d_{AC} are the sets of added and deleted attributes (identified as pairs (attribute, class)), respectively, then
   $R^a_{AC} = R^2_{AC} - R^1_{AC}$, $R^d_{AC} = R^1_{AC} - R^2_{AC}$ *where the superscripts* 1 *and* 2 *denote the version numbers.*
2. Added/Deleted method: If R^a_{MC} and R^d_{MC} are the sets of added and deleted methods (identified as pairs (method, class)), respectively then
   $R^a_{MC} = R^2_{MC} - R^1_{MC}$, $R^d_{MC} = R^1_{MC} - R^2_{MC}$
3. A changed method is a member of ${R_{MC}}^c$. There are three types of method changes:
   - A changed pre/post condition. Let P be the set of post conditions and R_{PMC} the relation that defined postconditions of methods in classes.
     $(\forall (m,c) \in R_{MC})(\exists p1, p2 \in P)$
     $(p1,m,c) \in R^1_{PMC} \wedge (p2,m,c) \in R^2_{PMC} \wedge$
     $p1 \neq p2 \Rightarrow (m,c) \in R_{MC}$
   - A method accesses a changed attribute. Let R_{ACM} define the relation between methods and the attributes they access. Then
     $(\forall (m,c) \in R_{MC})(\exists a \in A)$
     $a \in A^c \wedge (a,c,m) \in R_{ACM} \Rightarrow (m,c) \in R^c_{MC}$
   - A method navigates a changed relationship. Let R_{PMCR} define relationships that are navigated in the postconditions of methods in classes, then
     $(\forall (m,c) \in R_{MC})(\exists p \in P, r \in R)$
     $(p,m,c,r) \in R^2_{PMCR} \wedge (c,r) \in R^c_{CR} \Rightarrow (m,c) \in R^c_{MC}$
4. Added/Deleted relationships: Using similar notations to the ones above, the set of added/deleted relationships is $R^a_{CR} = R^2_{CR} - R^1_{CR}$, $R^d_{CR} = R^1_{CR} - R^2_{CR}$
5. Changed relationships: The relationship type changes between successive versions. Let C_{ac} be the set of association classes ($C_{ac} \subseteq C$) and C^c the set of changed classes. Then
   $(\forall r \in R)(\exists c \in C) c \in C_{ac} \cap (c,r) \in R_{CR} \Rightarrow r \in R^c_{CR}$
6. Added/Deleted classes:
   $C^a = C^2 - C^1$, $C^d = C^1 - C^2$
7. A changed class exists in both versions of the class diagram, but there are added/changed/deleted attributes, methods, or relationships:
   $C^C = \{c \in C | (\exists a \in A, m \in M, r \in R)$
   $(a,c) \in R^a_{AC} \cup R^d_{AC} \cup R^c_{AC}$
   $\vee (m,c) \in R^a_{MC} \cup R^d_{MC} \cup R^c_{MC}$
   $\vee (c,r) \in R^a_{CR} \cup R^d_{CR} \cup R^c_{CR}$

Further, categories of change are associated with use case diagrams and sequence diagrams. Before we explain these categories in detail, let us introduce some of the notations used in [30]. Let U denote the set of use cases, S denote the set of sequences in all sequence diagrams, and SB denote the set of sequences consisting of only methods belonging to boundary classes. Methods in boundary classes are those that are interacting with the system environment (actors). $S_i \in S$ denotes

DOI: 10.1002/smr

a particular method sequence occurring in one of the sequence diagrams (SB_i can be defined respectively for boundary methods).

The remaining classes of changes are derived from both use case and sequence diagrams. According to [30] these changes are as follows:

1. Added/Deleted use cases are identified via name changes:
   $U^a = U^2 - U^1$, $U^d = U^1 - U^2$
2. Changed use cases exist in both versions, but with changed sequence diagram(s).

A sequence diagram is changed if methods are added, deleted, or changed.

Changed classes are formalized using relations. Let R_{MC} denote the relation that identifies methods in classes, R_{MCS} denote the relation that identifies methods in classes participating in a sequence, and R_{SU} denote the relation that associates each sequence with a use case. The impact of added/deleted/changed methods on changed sequences and use cases are formalized in [30] as follows (these also apply to SB):

- $(\forall s \in S)(\exists (m,c) \in R_{MC})((m,c) \in R^C_{MC})$
  $(m,c,s) \in R_{MCS} \Rightarrow s \in S^C$
- $(\forall s \in S)(\exists (m,c) \in R_{MC})((m,c) \in R^d_{MC})$
  $(m,c,s) \in R_{MCS} \Rightarrow s \in S^1 \cap S^d$
- $(\forall s \in S)(\exists (m,c) \in R_{MC})((m,c) \in R^a_{MC})$
  $(m,c,s) \in R_{MCS} \Rightarrow s \in S^2 \cap S^a$
- $(\forall u \in U)(\exists s \in S) s \in S^c \cup S^a \cup S^d$
  $\wedge (s,u) \in R_{SU} \Rightarrow u \in U^c$

After identification and classification of changes, test cases are classified into *obsolete*, *retestable*, and *reusable*. Let R_{TS} and R_{TSB} denote the relations that trace a test case to a sequence in S and SB, respectively. A test case is considered as *obsolete* if it consists of an invalid execution sequence of boundary class methods. This may be due to deletion of the associated use case, addition/deletion of interface/boundary class methods, or a change in a sequence of boundary method invocations. The set of obsolete test cases T_{o} is formalized as [30]

- $(\forall t \in T) R^1_{TSB}(t) \in SB^d \Rightarrow t \in T_{\mathrm{o}}$

A *retestable* test case is a test case that remains valid in terms of a sequence of boundary method executions, although methods executed by the test case or those that are indirectly triggered by the test case may have changed. The set of retestable test cases T_{rt} is defined as

- $(\forall t \in T) R^1_{TSB}(t) \notin SB^d \wedge R^2_{TS}(t) \in S^c \Rightarrow t \in T_{\mathrm{rt}}$

All other existing test cases are classified as *reusable*. In [30] a prototype tool called regression test selection tool (RTSTool) is introduced that classifies the test cases automatically using two versions of class, sequence, and use case diagrams along with the original test suite.

Figure 10 shows the sequence diagram for the Withdraw Cash transaction of the changed ATM design. As a result the 'writeSecurityLog()' and the 'retainCard()' methods are added to the Controller class and the Card Reader class, respectively. These changes are identifiable in the modified class diagram and according to the classification of changes in [30], they should be included in the $R^a_M C$ set.

DOI: 10.1002/smr

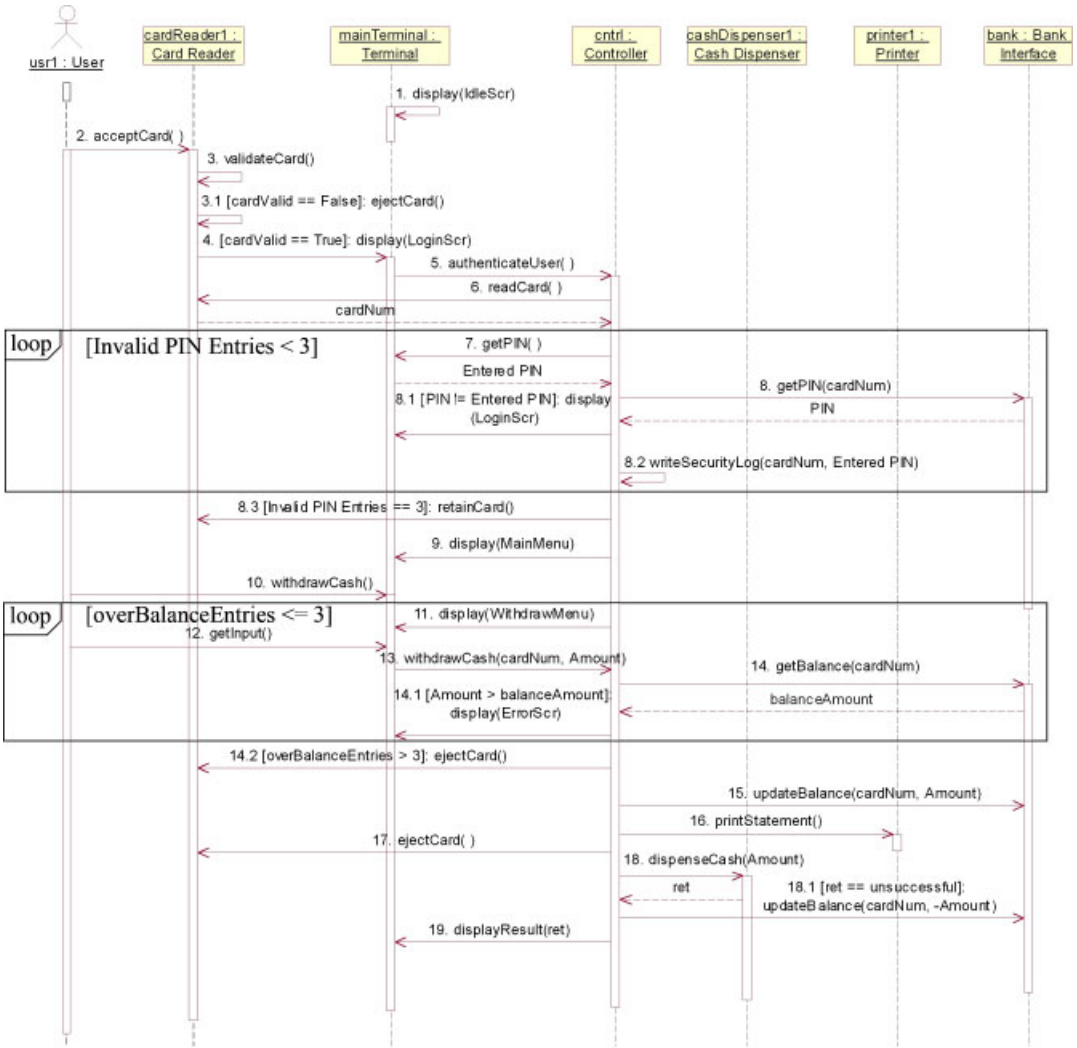


Figure 10. The new sequence diagram for Withdraw Cash transaction.

The new sequence diagram has three new messages labeled with 8.2, 8.3, and 18.1. Based on the classification of change to the sequence diagram, these messages are included in the S^a set. Further, the 'ejectCard()' method labeled with 8.2 in the original sequence diagram of Figure 5 is removed from the new sequence diagram. Hence it is added to S^d. Finally, the 'authenticateUser()' method labeled with 5 is a changed method because it invokes a different sequence of messages, namely, 8.1, 8.2, and 8.3 in the new sequence diagram. Thus, this change is added to the S^c set.

Because all changes in the new design do not affect boundary messages, none of the existing test cases becomes obsolete. Based on the traceability information provided in Table VI, the existing

DOI: 10.1002/smr

Table VI. Traceability between test cases and their path in the original sequence diagram.

TC #	Message sequence in the original sequence diagram
1	(1), (2), (3), (3.1)
2	(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19)
3	(1), (2), (3), (4), (5), (6), (7), (8), (8.1), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19)
4	(1), (2), (3), (4), (5), (6), (7), (8), (8.1), (7), (8), (8.1), (7), (8), (8.1), (8.2)
5	(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (14.1), (11), (12), (13), (14), (15), (16), (17), (18), (19)
6	(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (14.1), (11), (12), (13), (14), (14.1), (11), (12), (13), (14), (14.1), (11), (12), (13), (14), (14.1), (14.2)

test cases are classified as follows:

- Test case 1 is reusable.
- Test cases 2, 3, 4, 5, 6, are retestable.

5. REGRESSION TESTING UML DESIGNS

Recently, Pilskalns *et al.* [33] introduced a regression testing approach that applies to the UML design itself. This means, unlike the techniques introduced in the previous section, this approach does not require implementation and is applied to the design itself as the design is changed. This approach can be summarized in three major phases:

1. Identifying and classifying changes in a similar manner to [30].
2. Adapting a safe and efficient code-based technique [25] to UML designs for selecting UML tests.
3. Generating new UML tests using an approach similar to [27].

This approach is different from [30] as here they confront the problem of identifying changes that affect UML test cases as opposed to code test cases in [30]. Unlike the other techniques, [33] addresses tests for class and sequence diagrams that allow to test interactions between structural and behavioral design artifacts.

Taking a similar approach to that of [25], the problem of regression testing UML designs can be formalized as follows:

Given a design D, a test set T (used to test D), and a modified version of D, D',. Find a way of making use of T to gain sufficient confidence in the correctness of D'.

The proposed approach has the following steps:

1. Identify and classify changes made to D by creating a mapping of changes between D and D'. Based on the identified changes, classify T into mutually exclusive sets of T_o, T_r, and T_u corresponding to obsolete, reusable, and retestable test cases, respectively.
2. Construct $T' \subseteq T_r$ that may reveal change-related faults in D'.
3. Use T' to test D'.

DOI: 10.1002/smr

4. Identify inadequately tested parts and generate a set of new tests T''.
5. Use T'' to test D'.

We now explain the above steps in more detail. An obvious requirement in regression testing is to be able to identify changes in the design. In order to identify changes and relate them to test cases, Pilskans *et al.* [10] use the Object Method Directed Acyclic Graph (OMDAG), which was introduced in their earlier work. The OMDAG is constructed by combining the behavioral information of sequence diagrams with the structural information of class diagrams. The OMDAG is constructed in three steps:

1. Construct a directed graph (DG) from each sequence diagram.
2. Construct class and constraint tuples (CCT) from class diagram and Object Constraint Language (OCL) expressions.
3. Combine DG and CCT into OMDAG.

The construction of DG starts by traversing the first message in the sequence diagram and creating its corresponding vertex. In general if m_i and m_j are two messages in the sequence diagram and v_i and v_j the corresponding vertices, an edge is added from v_i to v_j if it is possible to execute v_j directly after v_i.

A DG is represented by the tuple $G = \langle V, E, s \rangle$, where V is a set of vertices, E is the set of edges, and s is the starting vertex. A vertex in DG can be a simple vertex representing a message or a sub-DG representing a *combined fragment*, hence representing several levels of abstraction. *Combined fragments* allow the developer to describe the control flow of messages with conditions. In the context of [10], three kinds of combined fragments are considered. These are *option* (i.e., 'if' statement), *alternative* (i.e., 'switch' statement), and *loop*. The loop fragment may contain a boolean guard condition, as well as a minimum and maximum number of iterations.

In general, each message vertex, v, is defined by the tuple $v = \langle o, m, lifeline, ARGS, c \rangle$, where o is an object calling m, m is the message, $lifeline$ classifies an object as *new* if it is being created, *deleted* if it is being deleted, and *exists* otherwise. *ARGS* is a set of argument tuples, and c the class name of the instance o. The *ARGS* tuple is composed of $\langle type, name \rangle$, where the $type$ is the argument type, and the *name* is the argument name.

CCTs contain structural and constraint information. Class diagrams and OCL expressions are used to derive CCTs. OCL expressions contain pre/post conditions as well as invariants. OCL invariants can represent association and multiplicity information among classes. CCTs consist of a class name, attributes, from class and superclasses (if applicable), operations for the class and superclasses, and OCL information for both the attributes and the operations (e.g., pre/post conditions). A CCT of a class c has the form

$$CCT(c) = \langle \{\langle Parent\ CCT \rangle\}, \{\langle Attribute \rangle\}, \{\langle Operation \rangle\}, \{[invariant]\} \rangle$$

where c is the class name, $\{\langle ParentCCT \rangle\}$ is a set consisting of parent class CCTs for each parent class of c, $\{\langle Attribute \rangle\}$ is a set of attribute tuples with constraints, $\{\langle Operation \rangle\}$ is a set of operation tuples with constraints, and *invariant* is a set of constraints at the class level (e.g., number of instances). The *Attribute* tuple is defined as follows:

$$Attribute = \langle attribute\ name, attribute\ type, visibility, invariant, \langle CCT \rangle \rangle$$

DOI: 10.1002/smr

The *Operation* tuple is defined as follows:

$$Operation = \langle name, return\ type, visibility, pre\_condition, post\_condition, \langle Parameters \rangle \rangle$$

The final step in building the aggregate model is to combine CCTs and DGs. This is done by replacing class name c in the DGs with their corresponding CCTs. While CCTs and DGs are being combined, static analysis evaluates the consistency of methods and parameters.

In the context of [33], a test case consists of a set of attribute values that satisfy conditions $c_{k_1}, \ldots, c_{k_m}$ for vertices $v_1, \ldots, v_n$ in the OMDAG C for path P_C. A change in the design maps to a change in the OMDAG. This means that a change in two versions of a design can be revealed by examining their corresponding OMDAGs and finding changed vertices and edges. The following properties apply:

Property 1. A path P_t in an OMDAG has a one to many relationship with test cases.

This suggests that a path change (either due to changes in a vertex or an edge) can affect one or more test cases. Many types of changes in UML may affect paths. For example, the addition of a message in a sequence diagram results in a new vertex and new edges in the OMDAG, thus changing paths associated with some test cases. All test cases associated with that path are affected.

Property 2. If a DC affects a path P_t, it also affects the set of test cases TC, associated with the path.

In general changes can be classified based on their effect on the design elements and their effect on paths in OMDAG. Changes are classified into mutually exclusive sets of NEWSET, MODSET, and DELSET according to whether they create, modify, or delete elements in a design, respectively. Using the second property, changes are categorized as changes that affect path PDCSET (path design changes (PDCs)) and those that do not NDCSET (non-path design changes (NDC)).

Classifying a change based on their effect on design elements is straightforward and it can be derived from the DC title. But identifying whether it is PDC or NDC is a more difficult task. In order to facilitate this process Pilskans *et al.* introduce a change table for class and sequence diagram.

Figure 11 represents a class diagram change table. The first column contains the element of the class diagram that may change due to a DC. The lines connecting the elements in the first column show refinement. It can be thought of as a 'consists of' relationship. For example a class has attributes, methods, generalizations, etc. The second column indicates the types of uses of an element in the sequence diagram. Whether or not a change affects or does not affect a path depends on its use in the sequence diagram. For example, a class may *not be used* in a sequence diagram, it may be *instantiated*, or it may be *used in a condition*. For a changed class diagram element, with a use in a sequence diagram according to column 2, column 3 lists whether the change is a PDC or an NDC. This classification will help to determine which nodes in the OMDAG influence retestability and obsolescence for test cases whose execution paths contain these nodes.

Figure 12 represents the sequence diagram change table. The first column in the table contains the element in the sequence diagram. The lines connecting the elements in the first column show refinement, similar to the refinement approach used for the class diagram in Figure 11. For example an object is described by a lifeline, methods, pre-conditions, and post-conditions. The second column indicates the types of uses for each element in the sequence diagram. For instance, an object may be *used in a condition* or not be used in a condition. As before, if an element is changed and is used

DOI: 10.1002/smr

Class Diagram Element Change Type	Sequence Diagram Use	Path Type
Class	Not used	NDC
	Instantiated	PDC
	Used in condition	PDC
Generalization	Not used	NDC
	Instantiated	PDC
Attribute	Not used	NDC
	Instantiated	NDC
	Used in condition	PDC
Primitive	Not used	NDC
	Used in condition	PDC
Class	Not used	NDC
	Instantiated	NDC
	Used in condition	PDC
Visibility	Not used	NDC
	Used	NDC
Method	Not Used	NDC
	Used by object	PDC
Arguments	Not used	NDC
	Used in condition	PDC
Primitive	Not used	NDC
	Used in condition	PDC
Class	Not used	NDC
	Instantiated	NDC
	Used in condition	PDC
Visibility	All uses	N/A*

*: OCL, by definition, cannot affect paths. But since it is a test oracle, all associated tests need to be re-run.

Figure 11. Class diagram change table mapping [33].

Sequence Diagram Element Change Type	Element use	Path Type
Class	Not used in condition	PDC
	Used in condition	PDC
Visibility	All uses	PDC
Method	All uses	PDC
Arguments	Not used in condition	NDC
	Used in condition	PDC
Primitive	Not used in condition	NDC
	Used in condition	PDC
Class	Not used in condition	NDC
	Used in condition	PDC
Pre-condition	All uses *(run associated tests)*	N/A*
Post-condition	All uses *(run associated tests)*	N/A*

*: OCL, by definition, cannot affect paths. But since it is attest oracle, all associated tests need to be re-run

Figure 12. Sequence diagram change table mapping [33].

DOI: 10.1002/smr

according to a usage type in column 2, then its change impact is defined in column 3. Column 3 indicates if the change is a PDC or not. As before, changes in the sequence diagram should be mapped onto the affected nodes and edges of the corresponding OMDAG so as to determine which test cases are retestable and which are obsolete.

The exact mapping between UML diagram elements and the OMDAG is explained in [10]. Mapping DCs to OMDAG vertices is done by finding corresponding vertices for each changed element and placing it into one of the NEWSET, MODSET, and DELSET based on the type of change. Depending on the type change, each vertex will fall into either a PDCSET or an NDCSET. If any vertex participating in a path belongs to PDCSET, then that path is changed and the corresponding test cases are considered affected.

Obsolete test cases are those that do not have the same input signature that matches the conditions in the design. Input signature is defined as attribute values and types associated with a test case. This implies that any test case that has a vertex in PDCSET is considered as obsolete. Let $\Delta_{DC}(TC)$ be a binary function that evaluates to one if a test case TC is affected by change DC and zero otherwise. The set of obsolete test cases can be formalized as follows:

$$obsolete = TC : \Delta_{DC}(TC) = 1 \wedge DC \in PDCSET$$

To be on the safe side, any modification in the test case signature will classify the test case as obsolete. In other words, the obsolete test cases are actually overestimated and in the later stages new test cases should be generated.

Retestable test cases are those that are changed, but the path that they cover is not changed. This means that none of the vertices participating along a test case path are members of PDCSET. In addition, at least one path vertex should be a member of either NEWSET, MODSET, or DELSET. Reusable test cases are neither obsolete nor retestable. These sets can be formalized in the following manner:

$$retestable = \{TC : \Delta_{DC}(TC) = 1 \wedge DC \in NDCSET \wedge (DC \in NEWSET \vee DC \in MODSET \vee DC \in DELSET)\}$$
$$reusable = \{TC \notin obsolete \vee TC \notin retestable\}$$

The need for new test cases arises from the fact that not every change is traversed by the existing test cases. Let GEN represent the set of vertices that require new path generation. GEN is defined in terms of *NEWSET*, *MODSET*, and *PDCSET* as follows:

$$GEN = ((PDCSET \cap NEWSET) \cup (PDCSET \cap MODSET))$$

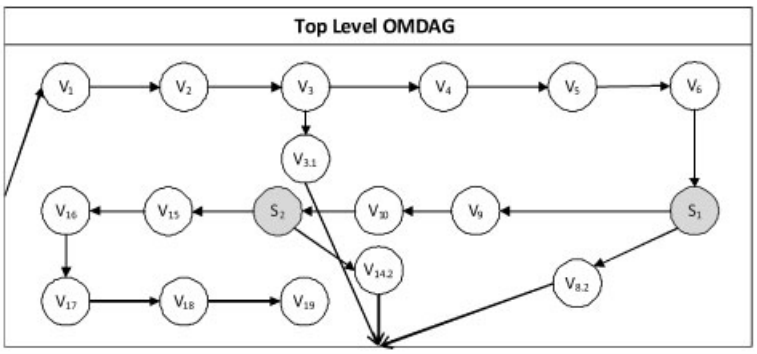
All conditions on paths that lead to vertices in GEN are placed in $CGEN$ and this set of conditions will be subject to non-binary variable domain analysis adapted from [27].

In this approach, test selection (classification) is based on change identification in a directed graph (OMDAG), which is a simpler form of a CFG. Since Rothermel and Harrold [25] showed that using a CFG would make the test selection algorithm safe, this approach is safe, in the following sense: it is safe for testing the design itself but not necessarily for code changes. In the end Pilskalns *et al.* argue that if the number of changes is small, this approach is proved to be efficient[¶].

[¶] This means that the number of test cases selected and generated would be less than the total number of test cases in the initial test suite.

DOI: 10.1002/smr

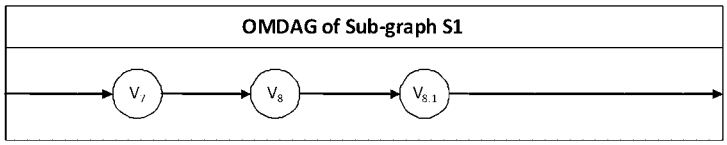
Now, we illustrate this technique on our example design. Figure 13 shows the top level OMDAG for the original sequence diagram of Figure 5. Figures 14 and 15 show the OMDAGs for sub-graphs S_1 and S_2, respectively. As mentioned earlier, the test cases are defined in terms of values for input attributes. Table VII shows the test cases generated after applying the technique in [10] to the UML design. For traceability purposes, the messages traversed by each test case is listed by a sequence



Node Label	[Condition]	<Object,	Message,	Arguments,	Lifeline,	Object's class>
V1		<mainTerminal,	display(),	{<Window, IdleScr>},	exists,	Terminal>
V2		<usr1,	acceptCard(),	{null},	exists,	User>
V3		<cardReader1,	validateCard(),	{null},	exists,	Card Reader>
V3.1	[cardValid == False]	<cardReader1,	ejectCard(),	{null},	exists,	Card Reader>
V4	[cardValid == True]	<cardReader1,	display(),	{<Window, LoginScr>},	exists,	Card Reader>
V5		<mainTerminal,	authenticateUser(),	{null},	exists,	Terminal>
V6		<cntrl,	readCard(),	{null},	exists,	Controller>
S1	[Invalid PIN Entries < 3]					
V8.2	[Invalid PIN Entries == 3]	<cntrl,	ejectCard(),	{null},	exists,	Controller>
V9		<cntrl,	display(),	{<Window, MainMenu>}	exists,	Controller>
V10		<usr1,	withdrawCash(),	{null},	exists,	User>
S2	[overBalanceEntries <=2]					
V14.2	[overBalanceEntries > 2]	<cntrl,	ejectCard(),	{null},	exists,	Controller>
V15		<cntrl,	updateBalance(),	{<int, cardNum>, <real, amount>},	exists,	Controller>
V16		<cntrl,	printStatement(),	{null},	exists,	Controller>
V17		<cntrl,	ejectCard(),	{null},	exists,	Controller>
V18		<cntrl,	dispenseCash(),	{<real, amount>},	exists,	Controller>
V19		<cntrl,	displayResult(),	{<int, ret>},	exists,	Controller>

Figure 13. OMDAG for the sequence diagram of Figure 5.

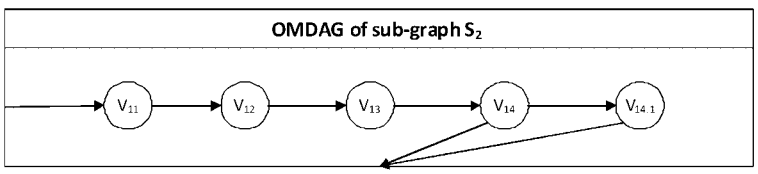
DOI: 10.1002/smr



For Subgraph S1

V7		<cntrl,	getPIN(),	{null},	exists,	Controller>
V8		<cntrl,	getPIN(),	{int, cardNum},	exits,	Controller>
V8.1	[PIN != Entered PIN]	<cntrl,	display(),	{<Window, ErrorScr>},	exists,	Controller>

Figure 14. OMDAG of sub-graph S_1 in Figure 13.



For Sub-graph S2

V11		<mainTerminal,	display(),	{<Window, WithdrawMenu>},	exists,	Terminal>
V12		<usr1,	getInput(),	{null},	exits,	User>
V13		<mainTerminal,	withdrawCash(),	{<int, cardNum>, <real, amount>},	exists,	Terminal>
V14		<cntrl,	getBalance(),	{<int, cardNum>},	exists,	Controller>
V14.1	[amount > currentBalance]	<cntrl,	display(),	{<Window, ErrorScr>},	exists,	Controller>

Figure 15. OMDAG of sub-graph S_2 in Figure 13.

of message numbers in the last column of Table VII. It is important to mention that we did not include all test cases due to space constraints. For similar reasons, we did not mention the value of the attributes that are not influencing the path traversed by the test case.

We demonstrate this technique on the modified design of the ATM introduced earlier. The sequence diagram of the Withdraw Cash transaction for the modified design is shown in Figure 10. In comparison to the original sequence diagram of Figure 5 the modifications include:

- The message labeled with 8.1 (change in the value of argument) is modified.
- The 'ejectCard()' message labeled with 8.2 in the original sequence diagram is replaced with the message 'writeSecurityLog()' with the same label.
- The messages 8.3 and 18.1 are added.

The changes in the class diagram include the addition of 'retainCard()' and 'writeSecurityLog()' methods to Card Reader class and Controller class, respectively. Figures 16 and 17 show the

DOI: 10.1002/smr

Table VII. Test cases for the original Withdraw Cash sequence diagram.

TC #	Test case: values for the input attributes	Path in the OMDAG
1	cardValid = False	(1), (2), (3), (3.1)
2	cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 0	(1), (2), (3), (4), (5), (6), (S_1), (9), (10), (S_2), (15), (16), (17), (18), (19)
3	cardValid = True, Entered PIN_1 = Wrong PIN, Entered PIN_2 = Right PIN, Invalid PIN Entries = 1, overBalanceEntries = 0	(1), (2), (3), (4), (5), (6), (S_1), (S_1), (9), (10), (S_2), (15), (16), (17), (18), (19)
4	cardValid = True, Entered PIN_1 = Wrong PIN, Entered PIN_2 = Wrong PIN, Entered PIN_3 = Wrong PIN, Invalid PIN Entries = 3	(1), (2), (3), (4), (5), (6), (S_1), (S_1), (S_1), (8.2)
5	cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 1	(1), (2), (3), (4), (5), (6), (S_1), (9), (10), (S_2), (S_2), (15), (16), (17), (18), (19)
6	cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 4	(1), (2), (3), (4), (5), (6), (S_1), (9), (10), (S_2), (S_2), (S_2), (S_2), (14.2)

OMDAGs for the new version of the Withdraw Cash transaction sequence diagram and its subgraph S_1.

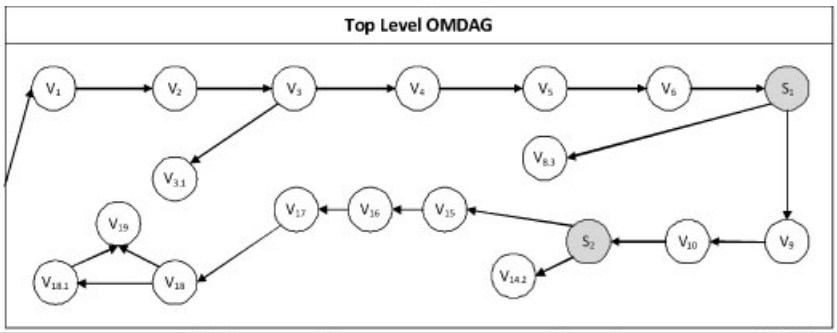
The existing test cases are classified based on the differences in the OMDAGs of the old and the new designs. The message 8.1 is placed in the MODSET and the new methods are included in the NEWSET. In addition, the 'ejectCard()' message labeled with 8.2 in the old sequence diagram is placed in the DELSET. Based on the mapping in Figures 11 and 12, with the exception of the change in the message labeled with 8.1, all other changes are classified as PDCs. Hence, test cases that traverse any vertices affected by these changes are either retestable or obsolete and the others are reusable.

Based on the classification of changes and the OMDAG path traversed by each test case, the existing test suite is classified as follows:

- Test cases 1, 2, 5, 6 are reusable, because none of the vertices along the associated test execution paths are affected by DCs.
- Test case 3 is retestable because the only affected vertex in its OMDAG path is vertex 8.1, which although it is in MODSET, is not considered a PDC, i.e., it is a member of NDCSET.
- Test case 4 is obsolete, because it traverses vertex 8.2 in the old sequence diagram and this vertex is deleted (replaced) in the new sequence diagram.

Finally, new test cases need to be generated for the new parts of the design to test the new functionalities (e.g., retain card after three invalid entries). In the new sequence diagram, the message labeled with 18.1 is executed based on the value of 'ret', which is based on the status of the Dispenser unit. Hence, we include an implicit input attribute called 'Dispenser Status', which has a direct effect on the value of ret. In other words, if Dispenser Status is faulty, then ret will be unsuccessful. The newly generated test cases should include this new input attribute. Table VIII lists the new test cases.

DOI: 10.1002/smr



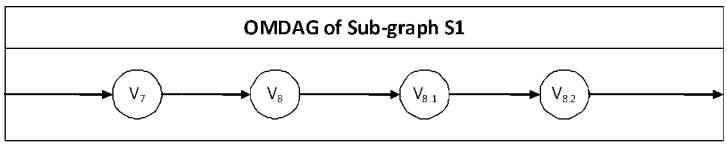
Node Label		<Object,	Message,	Arguments,	Lifeline,	Object's class>
V1		<mainTerminal,	display(),	{<Window, IdleScr>},	exists,	Terminal>
V2		<usr1,	acceptCard(),	{null},	exists,	User>
V3		<cardReader1,	validateCard(),	{null},	exists,	Card Reader>
V3.1	[cardValid == False]	<cardReader1,	ejectCard(),	{null},	exists,	Card Reader>
V4	[cardValid == True]	<cardReader1,	display(),	{<Window, LoginScr>},	exists,	Card Reader>
V5		<mainTerminal,	authenticateUser(),	{null},	exists,	Terminal>
V6		<cntrl,	readCard(),	{null},	exists,	Controller>
S1	[Invalid PIN Entries <3]					
V8.3	[Invalid PIN Entries== 3]	<cntrl,	retainCard(),	{null},	exists,	Controller>
V9		<cntrl,	display(),	{<Window, MainMenu>}	exists,	Controller>
V10		<usr1,	withdrawCash(),	{null},	exists,	User>
S2	[overBalanceEntries <=3]					
V14.2	[overBalanceEntries > 3]	<cntrl,	ejectCard(),	{null},	exists,	Controller>
V15		<cntrl,	updateBalance(),	{<int, cardNum>,<real, amount>},	exists,	Controller>
V16		<cntrl,	printStatement(),	{null},	exists,	Controller>
V17		<cntrl,	ejectCard(),	{null},	exists,	Controller>
V18		<cntrl,	dispenseCash(),	{<real, amount>},	exists,	Controller>
V18.1	[ret==unsuccessful]	<cntrl,	updateBalance(),	{<int, cardNum>, <real, -amount>},	exists,	Controller>
V19		<cntrl,	displayResult(),	{<int, ret>},	exists,	Controller>

Figure 16. OMDAG for the sequence diagram of Figure 10.

6. EVALUATION OF THE REGRESSION TESTING TECHNIQUES

In this section, we evaluate the regression testing techniques explained in the previous section using the framework introduced by Rothermel and Harrold [26]. Table IX summarizes our evaluation of the existing UML regression testing techniques. The first column lists each of the techniques. The first four techniques select test cases that are testing the code, i.e., *With UML*, as opposed to the last two that select test cases that are testing the design itself i.e., *For UML*.

DOI: 10.1002/smr



For Subgraph S1

V7		<cntrl,	getPIN(),	{null},	exists,	Controller>
V8		<cntrl,	getPIN(),	{int, cardNum},	exits,	Controller>
V8.1	[PIN != Entered PIN]	<cntrl,	display(),	{<Window, LoginScr>},	exists,	Controller>
V8.2		<cntrl,	WriteSecurityLog(),	{null},	exists,	Controller>

Figure 17. OMDAG of sub-graph S_1 in Figure 16.

Table VIII. New test cases for the new Withdraw Cash sequence diagram.

TC #	Test case: values for the input attributes	Path in the OMDAG
$4'$	cardValid = True, Entered PIN_1 = Wrong PIN, Entered PIN_2 = Wrong PIN, Entered PIN_3 = Wrong PIN, Invalid PIN Entries = 3	(1), (2), (3), (4), (5), (6), (S_1), (S_1), (S_1), (8.3)
7	cardValid = True, Entered PIN = Right PIN, overBalanceEntries = 0, Dispenser Status == Faulty	(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (9), (10), (S_1), (15), (16), (17), (18), (18.1), (19)

The second column classifies each regression testing technique based on the regression testing domain shown in Figure 1. The third column classifies the UML diagram types that are being used by each technique to select the test cases. In case a testing technique is supported by an automatic tool, the tool name is listed in the fourth column. The fourth column addresses the safety aspect of each technique. Finally the six evaluation measures that are introduced in [26] comprise the last five columns in Table IX.

Because of the lack of sufficient experimental studies for these techniques, we use an ordinal level of measurement (low, medium, high). Although *Inclusiveness*, *Precision*, and *Efficiency* can be measured using ratio scales, due to the lack of sufficient experimental studies, a ranking scale is more appropriate.

In [15] a case study is provided based on three components of IBM's WebSphere Commerce product. There is very little detailed information available about the components. Based on the number of defects and number of test cases, we consider this as a small to medium scale study. Safety is not a real concern in this approach. Instead, the cost of regression testing and risk of compromising the main functionality are the focus. Hence the approach is not safe. The authors claim (and we agree) that their approach is automatable using current tools (no specific tools are mentioned). The fact that [15] uses only the activity diagram yields low generality.

DOI: 10.1002/smr

Table IX. Evaluation of regression testing technique.

Technique	Regression testing domain	Type(s) of UML diagram	Tools	Safety	Evaluation measures			
					Inclusiveness	Efficiency	Precision	Generality
[15]	Cost and priority minimalistic	Activity diagram	None	Not safe	N/A	High	Low	Low
[31]	Coverage	Activity diagram, Class diagram	None	Safe	High	High	Low	Low
[32]	Cost minimalistic	Statecharts	Statechart base DeJavu	Safe	High	High	High	Low
[30]	Coverage	Use case diagram, Sequence diagram, Class diagram	RTSTool	Not safe	High	High	High	High
[10,33]	Coverage	Sequence diagram, Class diagram		Safe	High	Medium	High	Low

DOI: 10.1002/smr

Orso *et al*. [32] is a cost minimalistic approach and hence it is efficient in terms of cost. Although the goal is to select the fewest test cases, the approach is safe since it selects all modification exercising test cases. Its precision is considered high, based on the case study done on a small XML Parser. As with [15], the generality of [32] is limited due to the fact that it only uses state diagrams. On the other hand this approach is tool supported.

Martins and Viera's [31] approach is also a coverage-based approach. This approach uses activity and class diagrams, which restrict its generality. The authors provide a case study using a small size C++ library. The results indicate that their approach is effective and achieves a good level of precision and inclusiveness. Although the authors do not provide any tools for their approach, they claim that it is efficient.

Briand *et al.*'s [30] approach is based on achieving the appropriate test coverage when selecting test cases. This approach employs use cases, sequence, and class diagrams, which makes it the most generalizable approach available to date. Many of its stages are also tool supported. They represent results from three case studies varying from small to large scale, which indicate that their approach is efficient, and achieves high precision and inclusiveness. This approach is not safe, although the authors have empirical data from their case studies stating that in most cases this is not a major problem. Additionally, this approach requires a mapping between UML and code, a mapping that stays consistent through changes to UML and code artifacts.

The Pilskalns *et al.* [33] approach differs substantially from other regression testing techniques, since it generates test cases that are only executed upon UML designs. The purpose of the approach is to reveal inconsistencies in a UML design by transforming the design into a graph and testing the graph. The benefit of this approach is that when the DCs, the corresponding graph reflects the changes. Thus, the approach can guarantee 100% inclusiveness, resulting in a safe approach. However, since this approach can be applied to designs only, we cannot say it will generalize to include code testing. The approach is only efficient if the ratio of DCs to test cases is less than the total runtime of executing all test cases. Thus the efficiency should be classified as medium or low.

In general, we consider [30] to be the strongest regression testing technique *With UML*. It is tool supported, and the most general among all the other techniques. We consider the current experimental studies in regard to all these testing techniques as very limited and inconclusive. Almost all techniques presented here could benefit from more empirical evaluation related to both effectiveness (fault exposure) and efficiency (scalability). Further, as of yet, there is no technique that combines all UML model types to enable testing across multiple views. Pilskalns *et al.* [33] is the first step in that direction. We plan research to further integrate UML model types in the future to develop a more comprehensive testing approach. In addition, many of the techniques presented are 'single issue', e.g., prioritization. It would be useful to have techniques that either can be combined or are integrated and cover all aspects of the regression domain illustrated in Figure 1.

7. CONCLUSIONS

This article provided a survey of regression testing techniques for UML design artifacts. While regression testing techniques with UML span activity diagrams, state charts, and multiple notations, regression testing techniques that were developed to test DCs are limited to class and sequence diagrams. These are the most common UML diagram types. However, additional techniques for

DOI: 10.1002/smr

other types of diagrams are also needed. This article only concerned itself with functional testing, yet many DCs impact other properties like performance, fault tolerance, or security. At this point, no regression testing approaches exist to evaluate the impact of DCs on these design properties and the designer is left without a systematic approach to selectively re-evaluate the effect of DCs. Additionally, techniques become most useful when automated tool support exists. The techniques are all amenable, in principle at least, to automation, but many have not been automated.

We conclude that while the beginnings of a body of knowledge exists for regression testing UML, we are far from providing a comprehensive set of solutions that are tool supported.

REFERENCES

1. OMG. The Unified Modeling Language Standard.
2. Briand L, Cui J, Labiche Y. Towards automated support for deriving test data from UML statecharts. *UML '03*: *Proceedings of the Sixth International Conference on UML*. Springer: London, U.K., 2003; 249–264.
3. Briand L, Labiche Y. A UML-based approach to system testing. *UML'01*: *Proceedings of the Fourth International Conference on UML*. Springer: London, U.K., 2001; 194–208.
4. Gnesi S, Latella D, Massink M. Formal test-case generation for UML statecharts. *ICECCS '04*: *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age* (*ICECCS '04*). IEEE Computer Society: Los Alamitos CA, U.S.A., 2004; 75–84.
5. Latella D, Massink M. A formal testing framework for UML statechart diagrams behaviours: From theory to automatic verification. *HASE '01*: *The Sixth IEEE International Symposium on High-assurance Systems Engineering*. IEEE Computer Society: Los Alamitos CA, U.S.A., 2001; 11–22.
6. Offutt A, Abdurazik A. Generating tests from UML specifications. *UML'99*: *Proceedings of the Second International Conference on UML*. Springer: London, U.K., 1999; 416–429.
7. Scheetz M, von Mayrhauser A, France R, Dahlman E, Howe A. Generating test cases from an OO model with an AI planning system. *ISSRE '99*: *Proceedings of the 10th International Symposium on Software Reliability Engineering*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1999; 250–259.
8. von Mayrhauser A, France R, Scheetz M, Dahlman E. Generating test-cases from an object-oriented model with an artificial-intelligence planning system. *IEEE Transactions on Reliability* 2000; **49**(1):26–36.
9. Gogolla M, Bohling J, Richters M. Validation of UML and OCL models by automatic snapshot generation. *UML '03*: *Proceedings of the Sixth International Conference on UML*. Springer: London, U.K., 2003; 265–279.
10. Pilskalns O, Andrews A, Knight A, Ghosh S, France R. UML design testing. *Journal of Information Science and Technology* 2007; **19**(8):192–212.
11. Pilskalns O, Andrews A, Ghosh S, France R. Rigorous testing by merging structural and behavioral UML representations. *UML '03*: *Proceedings of the Sixth International Conference on UML*. Springer: London, U.K., 2003; 234–248.
12. Trong T, Kawane N, Ghosh S, France R, Andrews A. A tool-supported approach to testing UML design models. *ICECCS '05*: *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society: Los Alamitos CA, U.S.A., 2005; 519–528.
13. Agrawal H, Horgan J, Krauser E, London S. Incremental regression testing. *ICSM '93: Proceedings of the Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1993; 348–357.
14. Chen Y, Probert R, Ural H. Model-based regression test suite generation using dependence analysis. *A-MOST '07*: *Proceedings of the Third International Workshop on Advances in Model-based Testing*. ACM: New York NY, U.S.A., 2007; 54–62. http://doi.acm.org/10.1145/1291535.1291541.
15. Chen Y, Probert R, Sims D. Specification-based regression test selection with risk analysis. *CASCON '02*: *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press: Canada, 2002; 1–14.
16. Elbaum S, Malishevsky AG, Rothermel G. Prioritizing test cases for regression testing. *SIGSOFT Software Engineering Notes* 2000; **25**(5):102–112.
17. Fischer K, Raji F, Chruscicki A. A methodology for retesting modified software. *Proceedings of the National Telecommunication Conference B-6-3*. IEEE Computer Society: Los Alamitos CA, 1981; B6.3.1–B6.3.6.
18. Gupta R, Harrold M, Soffa M. An approach to regression testing using slicing. *ICSM '92*: *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1992; 299–308.
19. Hartmann J, Robson D. Techniques for selective revalidation. *IEEE Software* 1990; **7**(1):31–38.
20. Kim J, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. *ICSE '02*: *Proceedings of the 24th International Conference on Software Engineering*. ACM: New York NY, U.S.A., 2002; 119–129.

DOI: 10.1002/smr

21. Korel B, Tahat L, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings of the International Conference on Software Maintenance*, 2002; 214–223.
22. Korel B, Tahat L, Harman M. Test prioritization using system models. *ICSM'05*: *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005; 559–568.
23. Leung H, White L. A study of integration testing and software regression at the integration level. *ICSM '90*: *Proceedings of the International Conference on Software Maintenance*, 1990; 290–301.
24. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transaction on Software Engineering* 2007; **33**(4):225–237.
25. Rothermel G, Harrold M. A safe, efficient algorithm for regression test selection. *ICSM'93*: *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society: Washington DC, U.S.A., 1993; 358–367.
26. Rothermel G, Harrold M. A framework for evaluating regression test selection techniques. *ICSE'94*: *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, U.S.A., 1994; 201–210.
27. von Mayrhauser A, Zhang N. Automated regression testing using DBT and Sleuth. *Software Maintenance*: *Research and Practice* 1999; **11**(4):93–116.
28. Wong W, Horgan J, London S, Bellcore HA. A study of effective regression testing in practice. *ISSRE '97*: *Proceedings of the Eighth International Symposium on Software Reliability Engineering* (*ISSRE '97*). IEEE Computer Society: Los Alamitos CA, U.S.A., 1997; 264–274.
29. Yau S, Kishimoto Z. A method for revalidating modified programs in the maintenance phase. *COMPSAC '87*: *Proceedings of the 11th Annual International Computer Software and Applications Conference*. IEEE Computer Society: Los Alamitos CA, U.S.A., 1987; 272–277.
30. Briand L, Labiche Y, Soccar G. Automating impact analysis and regression test selection based on UML designs. *ICSM'02*: *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, U.S.A., 2002; 252–261.
31. Martins E, Vieira V. Regression test selection for testable classes. *Dependable Computing—EDCC 2005*, vol. 3463, 2005; 453–470.
32. Orso A, Do H, Rothermel G, Harrold M, Rosenblum D. Using component metadata to regression test component-based software: Research articles. *Software Testing*, *Verification and Reliability* 2007; **17**(2):61–94.
33. Pilskalns O, Uyan G, Andrews A. Regression testing UML designs. *ICSM '06*: *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society: Los Alamitos CA, U.S.A., 2006; 254–264.
34. Rothermel G, Harrold M. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
35. Bates S, Horwitz S. Incremental program testing using program dependence graphs. *POPL'93*: *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*. ACM Press: New York NY, U.S.A., 1993; 384–396.
36. Harrold M, Soffa M. An incremental approach to unit testing during maintenance. *ICSM '88*: *Proceedings of the Conference on Software Maintenance.* IEEE Computer Society: Los Alamitos CA, U.S.A., 1988; 362–367.
37. Ostrand T, Weyuker E. Using data flow analysis for regression testing. *Proceedings of Sixth Annual Pacific Northwest Software Quality Conference*, 1988; 147–233.
38. Rothermel G, Harrold M, Dedhia J. Regression test selection for c++ software. *Software Testing*, *Verification and Reliability* 2000; **10**(2):77–109.
39. Sabnani K, Lapone A, Uyar M. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications* 1989; **37**(9):940–948.

AUTHORS' BIOGRAPHIES

Alireza Mahdian is currently a third year PhD student in the Department of Computer Science at the University of Denver. His current research interests include software testing, software design, software maintenance. He received his BS degree in Computer Engineering from the Sharif University of Technology, Iran.

DOI: 10.1002/smr

Anneliese Amschler Andrews before joining the University of Denver, Dr Andrews was the Huie Rogers Endowed Chair in Software Engineering and the Associate Director of the School of Electrical Engineering and Computer Science at Washington State University. Dr Andrews is the author of a text book and over 130 articles in the area of Software Engineering, particularly software testing and maintenance. Dr Andrews holds an MS and a PhD from Duke University and a Dipl.-Inf. from the Technical University of Karlsruhe. She served as Editor-in-Chief of the IEEE Transactions on Software Engineering. She has also served on several other editorial boards including the IEEE Transactions on Reliability, the Empirical Software Engineering Journal, the Software Quality Journal, the Journal of Information Science and Technology, and the Journal of Software Maintenance. She was the Director of the Colorado Advanced Software Institute from 1995 to 2002. CASI's mission was to support technology transfer research related to software through collaborations between industry and academia. Dr Andrews also served as elected Vice President and Member of the Board of Governors for the IEEE Computer Society.

Orest Jacob Pilskalns earned Bachelor Degrees in Mathematics and Physics in 1996 and a Master's Degree in Computer Science from the University of Montana in 1998. After his graduation Orest worked as a research engineer for Lockheed Martin and as a software engineer for GemStone Systems. Orest returned to academics and received his PhD in Computer Science from Washington State University in 2004. He is currently an assistant professor at Washington State University and also the founder of the software company, MapWith.Us. His primary research interests are in software modeling analysis, software security, and highly scalable distributed systems.

DOI: 10.1002/smr