Regression Testing with

Software Designs

Alireza Mahdian¹, Anneliese Amschler Andrews¹∗,† and Orest Jacob Pilskalns²

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

Abstract.

UML designs encompass a variety of different notations. They can be quite large. 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.

1. Introduction

The Unified Modeling Language (UML) [Gro] has become a defacto industry standard. Traditionally, evaluation of UML designs usually 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 [BCL03, BL01, GLM04, LM01, OA99, SvMF+99, vMFSD00], or test the design itself [GBR03, PAK+07, PAGF03, TKG+05]. Given that designs tend to change, effects of such changes are often difficult to assess, especially for large and complex designs that have 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 modifications have not introduced errors. At this stage, existing work on regression testing techniques for designs or with designs is limited. This paper provides a survey of techniques that either suggest regression testing

∗Correspondence to: University of Denver, Department of Computer Science, 2360, S. Gaylord St., Denver CO 80208, USA
†E-mail: andrews@cs.du.edu
techniques that use UML artifacts, or suggest ways in which to regression test design modifications themselves. Table I classifies existing regression testing techniques into three categories:

1. Non-UML based regression testing techniques for designs
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 which can be adapted to or are relevant to testing designs, including non-UML based design reevaluation approaches. Section 3 describes an example design and 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, state charts, and multiple diagrams and their interactions. Section 5 describes a technique that defines a regression testing technique to evaluate UML design changes on some structural and behavioral notations. Section 6 draws conclusions.

## 2. Principles of Regression Testing

The process of testing software modifications is called regression testing. It involves retesting all or part of a software system after it is modified, depending on different regression testing strategies. Retest all and selective regression testing strategies are the two basic approaches to regression testing. The set of tests run during a regression test is the regression test suite [vMZ99].

A retest all strategy tests the system all over again, assuming that changes could have affected and introduced errors anywhere in the code. A selective regression testing strategy assumes that not all parts of the software could have been affected by modifications [vMZ99] and only a subset of the original test suite needs to be selected for regression testing. The main objective in selective regression testing is to 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 features of software that are not covered by existing tests. Rothermerl and Harrold [RH96] provided a definition of selective regression testing as consisting of the following steps:

1. Identify changes.

---

**Table I. Classification of Existing Work on Software Testing**

<table>
<thead>
<tr>
<th>Design Methods</th>
<th>non-UML Based</th>
<th>With UML</th>
<th>For UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression Testing</td>
<td>[AHKL93][FRC81][GHS92]</td>
<td>[BLS02][CPS02]</td>
<td>[PUA06]</td>
</tr>
<tr>
<td></td>
<td>[HR90][LW90][RH93][RH94]</td>
<td>[MV05][ODR+07]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[vMZ99][WHLB97][YK87]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 broken any existing functions. 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 [LW90] classified the 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 the modified code with respect to an unmodified specification. This means that they specify the correct input and output relation but they test changed code. Finally, obsolete test cases are those that are no longer valid.

This classification is based on three types of modifications during software maintenance: corrective maintenance, adaptive maintenance, and perfective maintenance. The objective of corrective maintenance is to keep the software working and involves correcting software, performance and implementation failures. In adaptive maintenance the system is adapted to changed data requirements, or processing environments. Perfective maintenance improves system performance and efficiency.

Adaptive maintenance and perfective maintenance are the result of modification in the specification. They introduce new modules to the system. In corrective maintenance the specification is not likely to be changed and no new modules are likely to be introduced. Leung and White [LW89] introduce the notions of progressive and corrective regression testing. Progressive regression testing deals with adaptive maintenance and perfective maintenance and in general whenever the specification is modified, whereas corrective regression testing deals with corrective maintenance.

Most of the time new test cases need to be added to the test suite in order to test a modified program or a modified specification. Thus, adding two new classes of test cases to reusable, retestable, and obsolete classification. In corrective regression testing we are dealing with reusable, retestable and new structural (white box) test cases whereas in progressive regression testing we are dealing with all five classes. Selective regression testing can be used for the purpose of corrective or progressive regression testing.

All regression testing strategies fall on a spectrum which on one end is the all-reuse based selective regression testing and on the other is the regeneration-based regression testing. The number of each test cases from each class is determined based on where a regression testing technique falls on the spectrum. All reuse-based selective regression testing techniques select retestable test cases from the existing test suite. A regeneration-based retest all approach produces the test cases from scratch and is equivalent to development testing [vMZ99].

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 testing to retest all regeneration-based regression testing.
Table II. Regression testing in various situations [vMZ99]

<table>
<thead>
<tr>
<th>Approach</th>
<th>Situation descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confidence in software quality</td>
</tr>
<tr>
<td>Full new test cycle</td>
<td>Low</td>
</tr>
<tr>
<td>Minimalistic regression test</td>
<td>High</td>
</tr>
<tr>
<td>Expanded scope of regression test</td>
<td>Moderate</td>
</tr>
<tr>
<td>Full reuse of existing test suite</td>
<td>Low</td>
</tr>
</tbody>
</table>

With the exception of retest all, regeneration-based regression testing techniques, all regression testing techniques employ some kind of selective testing strategy. Selective testing strategies can be classified as Minimization, Coverage, and Safe approaches [RH94]. This classification is based on how it is determined which parts of the software have been affected by changes and to what extend and how they have to be retested.

In minimization approaches [CPS02, FRC81, HR90, ODR+07, SK] the goal of regression testing is to reestablish satisfaction of some structural coverage criterion, by identifying a minimal set of tests that must be rerun to meet that criterion. Coverage approaches [BH93, BLS02, HS88, LW90, PUA06, OW88, RH93, vMZ99, YK87] rely on coverage criteria, but do not require minimization. Instead, they assume that a second but equally important goal of regression testing is to rerun tests that could produce different output, and they use coverage criteria as a guide in selecting such tests. Safe approaches place less emphasis on coverage criteria, and aim instead to select every test that will cause the modified program to produce different output than the original program [RH94].

Rothermel and Harrold [RH94] 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 a few of the existing selective retest algorithms. The framework analyzes each test selection method based on five aspects: Inclusiveness, Precision, Efficiency, Generality, and Accountability. The first three measures are quantitative. Generality and accountability are qualitative.

Before we define each measure we adapt some notations from [RH94]. 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 in [RH94]:

**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 \( (\frac{m}{n}) \times 100 \).
### Table III. Regression testing approaches [vMZ99]

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Full new test cycle</td>
<td>• redesign entire test plan</td>
</tr>
<tr>
<td></td>
<td>• rebuild the whole test suite</td>
</tr>
<tr>
<td>(2) Minimum regression test</td>
<td>• reuse test plan as much as possible</td>
</tr>
<tr>
<td></td>
<td>• rerun minimum number of retestable test cases</td>
</tr>
<tr>
<td></td>
<td>• generate minimum number of test cases for changes</td>
</tr>
<tr>
<td>(3) Expanded scope of regression testing</td>
<td>• reuse part of test plan</td>
</tr>
<tr>
<td></td>
<td>• rerun all retestable test cases</td>
</tr>
<tr>
<td></td>
<td>• generate new test cases on the full scope of changes</td>
</tr>
<tr>
<td>(4) Full reuse of existing test suite</td>
<td>• reuse the test plan and test suite</td>
</tr>
</tbody>
</table>

Safety can be defined in terms of inclusiveness as follows: *if a regression test strategy $S$ results in 100% inclusiveness then it is considered as safe.* Precision measures the ability of a method to avoid choosing tests that will not cause the modified program to produce different output. Using the notations introduced earlier, precision can be quantified using the following definition in [RH94]:
Definition 2. Suppose $T$ contains $n$ non-modification-revealing tests, and $S$ selects $m$ of these tests. The precision of $S$ relative to $P$, $P'$, and $T$ is the percentage calculated by the expression $\left(\frac{m}{n}\right) \times 100$.

Efficiency measures the computational cost and automatability, and thus practicality, of a selective retest approach. In general a selection strategy is efficient if the cost of running $T'$ is less than the cost of running $T - T'$.

Generality measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications. It is considered a qualitative measure. 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 [RH94].

Although this framework is based largely on qualitative comparisons, it provides a way to evaluate and compare existing selective retest approaches and therefore can be used as a guide to select the appropriate approach for a particular application.

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.

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
performance during unit testing, but not for system testing as they lack scalability. This is due to the fact that they use some traceability techniques (e.g. traceability table) to relate test cases to code statements. As the program becomes larger and larger constructing and maintaining traceability tables become more and more difficult. Another problem with code-based regression testing is the required knowledge of the code to select the test cases. In many cases this is a time consuming process and requires access to the code which is not always possible (e.g. a system composed of 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 [vMZ99]. 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 what would be the new 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 [BLS02].

The concept of design-based regression testing is new and there are few techniques that are design-based. Von Mayrhofer and Zhang [vMZ99] 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. It is more limited than the UML notation.

According to Von Mayrhofer and Zhang [vMZ99], 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 following sections, we introduce these more recent techniques. For better illustration, each technique is demonstrated on our general example design which is introduced in the next section.
3. Example design

As we introduce each regression testing technique we illustrate each technique on our example Automatic Teller Machine (ATM) design. For simplicity we only assume three functionalities for the ATM as demonstrated in Figure 2: Users can withdraw cash, get a print of their statements, or deposit cash.

![Usecase diagram of the example ATM.](image)

The design deliberately contains several defects and we mention each problem as we apply each testing technique to each type of UML diagram. 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 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.
As mentioned earlier, the techniques that we are going to introduce in the next sections use one or multiple UML diagrams to derive regression test cases. These techniques use UML use case diagrams, class diagrams, sequence diagrams, activity diagrams, and statecharts. Thus we include one of each of these diagrams in our example. ATM can be in one of five different states. Figure 4 represents these states and their interactions with each other through the statechart diagram.
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.

In addition, 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 insure there is enough balance before any withdraw transaction.
- The sequence diagram of the withdraw cash transaction also includes a fault which 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 the 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 the 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 [CPS02] 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 the targeted tests. Hence safety tests focus on key functions and exception handling to avoid this situation.

Chen et. al [CPS02] present a similar approach to the code-based approach of Rothermel et.al [RHD00] to select the targeted tests. In [RHD00] 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 [CPS02] use a UML activity diagram instead of the CFG to select the test cases.

Figure 5. Sequence diagram for the Withdraw cash transaction.
They identify two groups of changes. The first group consists of code changes that do not affect the system behavior. Hence, these changes are not detectable from the activity diagram. The code 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 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 which could result in losing market share. The other is the maintenance cost. Cost is categorized into a one to five scale.
2. Derive a severity index for each test case which 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 et. al [MV05] propose an approach to select test cases for regression testing a class. This approach is similar to [CPS02] in that both approaches select test cases based on control flow analysis.

Figure 6. Activity diagram for the Controller class.
and employ the activity diagram as the main artifact. The main difference is the fact 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 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 also labeled in the form of \([\text{condition}]\) or \([\text{else}]\) 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 length \( n \) is represented as \( p = [e_1, ..., e_n] \). 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 due to different test data or configurations.

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 includes the following features:

- Assertions that check pre/post conditions and invariants at run-time.
- 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 design change. Examples of contract changes include addition or removal of interface methods, alteration of method signature or changes in the pre/post conditions or invariants. An implementation change comprises all changes to a class that are not visible to its clients. In other words, they don’t alter the external behavior of the class. As for [CPS02] 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 post/pre conditions. The assertion feature of the BIT is a helpful instrument 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 the 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 detects all edges that are affected by

---

\(^{†}\)loops are not addressed in this work

Copyright © 2007 John Wiley & Sons, Ltd.

changes. The edges are classified into two sets: $D_e$ which represents deleted edges, and $R_e$ which represents retestable edges. Deleted edges are those that are either not present in the new version of the activity diagram or point to a vertex with a changed label. Retestable edges are those that 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 [MV05] and [CPS02] 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.

![Image](image.png)

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

Figure 8 shows the Behavioral Control Flow Graph of the old and new activity diagrams for the Controller class. Table IV represents the test cases corresponding to the original activity diagram. Each test case is composed of a sequence of activities that start from the initial state and ends in the final state of the activity diagram. The traceability information is provided in the third column of Table IV.

---

‡Refer to the example section

---

Copyright © 2007 John Wiley & Sons, Ltd.

by denoting the edges that are traversed by each test case. In addition, the 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 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. Also, because a cash withdrawal transaction is considered 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 [CPS02] and [MV05], the 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.
### Table IV. Test cases for the old Activity diagram of the Controller class

<table>
<thead>
<tr>
<th>TC #</th>
<th>Test Case</th>
<th>Traversed Edges</th>
<th>Severity Index</th>
<th>Cost</th>
<th>Risk Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>authenticateUser, withdrawCash, getBalance</td>
<td>(1), (2), (7), (5)</td>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>authenticateUser, getBalance</td>
<td>(1), (3), (5)</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>authenticateUser, depositCash</td>
<td>(1), (4), (6)</td>
<td>5</td>
<td>4</td>
<td>20</td>
</tr>
</tbody>
</table>

### 4.2. Using Statecharts

Orso et. al [ODR+07] present a similar approach to those of [CPS02, MV05] for regression test selection, but they employ statechart diagrams instead. This approach has two basic steps:

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

The first step uses an adapted version of an algorithm proposed in [RH93] to find the changed sections. The algorithm performs a pairwise synchronous walk of the two versions of the statechart diagram 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 a deleted transition from the old version and added transition to the new version of the statechart. In the next step, all transitions that lead to a changed state are marked as "dangerous transitions". In addition, changed transitions are marked as "obsolete transitions". In the process of marking the transitions "obsolete" has precedence over "dangerous", i.e. if a transition can be considered as both dangerous and obsolete it will be considered as 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, those test cases that have at least one obsolete transition in their path are considered obsolete test cases. From the remaining test cases, those that have at least one dangerous transition in their path are considered as retestable. The rest are reusable test cases.

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 practical. For this, Orso et. al [ODR+07] suggest the use of the Incremental Composition and Reduction (ICR) method proposed by Sabnani et. al [SLU89] to construct a global statechart of the system. The ICR method addresses the state explosion problem by avoiding all the unreachable states at the composition step and removing all replicated states from the global statechart diagram. Given the statechart diagrams for components are provided by the vendor, perhaps in the form of "metadata" as suggested in [ODR+07]. This approach can be employed to select regression test cases for component based software.
Table V. Test cases for the old statechart of the ATM design

<table>
<thead>
<tr>
<th>TC #</th>
<th>Test Case</th>
<th>Traversed Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert card, card read error.</td>
<td>(1), (2), (9)</td>
</tr>
<tr>
<td>2</td>
<td>insert card, card read successfully, invalid PIN entered.</td>
<td>(1), (3), (4), (9)</td>
</tr>
<tr>
<td>3</td>
<td>insert card, card read successfully, successful authentication, canceled.</td>
<td>(1), (3), (5), (6), (9)</td>
</tr>
<tr>
<td>4</td>
<td>insert card, card read successfully, successful authentication, transaction chosen, customer finished.</td>
<td>(1), (3), (5), (7), (8), (9)</td>
</tr>
</tbody>
</table>

Table V represents the test cases for the statechart of Figure 4. Each test case is defined as a sequence of events that results in traversing the statechart from 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.

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

The statechart of the upgraded ATM design is shown in Figure 9. As for the original statechart, the transitions are labeled with numbers 1 through 13. The differences introduced in the new statechart include the addition of "Writing log", and "Writing Security Log" states and four transitions labeled with 10 through 13. The "Performing Transaction" state is modified so that Unsuccessful transactions are written in the log file. Similarly, "Reading PIN" state is modified to support writing to 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 a changed transition (removed 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 [ODR’07] 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 by (7) is a dangerous transition. Similarly, "Reading PIN" state is a modified state. Thus, the transition labeled with (3) is considered a dangerous transition. In addition, "Card Read Error" and "Invalid PIN Entered" transitions labeled with (2′) and (4′) respectively, in the new diagram are changed transitions and thus, they are considered obsolete. Based on [ODR’07], test cases 1 and 2 are obsolete and 3 and 4 are retestable.

4.3. Using Multiple Diagrams

A more recent approach [BLS02] is to use architectural/design information to guide the test selection strategy. In this approach the impact of the change is first assessed on the design itself and depending on its magnitude a change management group will decide 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 to code. Another advantage of design level change analysis is that regression testing tools can be programming language independent. There is also a drawback associated with design level change analysis because the design needs to be complete and up to date[BLS02].

Briand et al. [BLS02] 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, namely, 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. Using test case classifications in [LW89] each test case is then categorized into mutually exclusive sets of obsolete, reusable, or retestable test cases.

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 [BLS02]:

---

1. Added/Deleted attribute: if $R_{AC}^A$ and $R_{AC}^D$ are the sets of added and deleted attributes (identified as pairs (attribute, class)) respectively then:
$$R_{AC}^A = R_{AC}^2 - R_{AC}^1, R_{AC}^D = R_{AC}^1 - R_{AC}^2 \text{ where the superscripts } 1 \text{ and } 2 \text{ denote the version numbers}$$

2. Changed attribute: is an attribute that exists in both versions but with different scope, type, or visibility

3. Added/Deleted method: a method is identified uniquely by its signature which consists of method name, parameters’ order, and type, return type, visibility and scope. If method’s signature is changed it is also considered as added/deleted method change. If $R_{MC}^A$ and $R_{MC}^D$ are the sets of added and deleted methods (identified as pairs (method, class)) respectively then:
$$R_{MC}^A = R_{MC}^2 - R_{MC}^1, R_{MC}^D = R_{MC}^1 - R_{MC}^2$$

4. Changed method: is represented as a member of $R_{MC}^C$. A method is considered as changed if it falls into one of the following situations:
   - It has a changed pre/post condition. Given $P$ the set of post conditions and $R_{PMC}$ the relation that defined postconditions of methods in classes, we have:
     $$\forall (m, c) \in R_{MC} \exists p_1, p_2 \in P \ (p_1, m, c) \in R_{PMC} \land (p_2, m, c) \in R_{PMC} \land p_1 \neq p_2 \Rightarrow (m, c) \in R_{MC}$$
   - It accesses a changed attribute. Given $R_{ACM}$ the relation that identifies attributes accessed by methods in classes, we have:
     $$\forall (m, c) \in R_{MC} \exists a \in A \ a \in A^c \land (a, c, m) \in R_{ACM} \Rightarrow (m, c) \in R_{MC}$$
   - It navigates a relationship that has been changed. Let $R_{PMCR}$ capture relationships that are navigated in the postconditions of methods in classes, we have:
     $$\forall (m, c) \in R_{MC} \exists p \in P, r \in R \ (p, m, c, r) \in R_{PMCR} \land (c, r) \in R_{CR} \Rightarrow (m, c) \in R_{MC}$$

5. Added/Deleted relationship: A relationship is identified by its name and if the name is changed it is considered as deleted and added relationship. Using similar notations we can define the set of added/deleted relationships as follows:
$$R_{CR}^A = R_{CR}^2 - R_{CR}^1, R_{CR}^D = R_{CR}^1 - R_{CR}^2$$

6. Changed relationship: It exists in both versions but with different type (i.e. generalization, association, dependency), multiplicity, or navigability. Given a set $C_{ac}$ the set of association classes ($C_{ac} \subseteq C$) and $C^c$ the set of changed classes we have:
$$\forall r \in R \exists c \in C_{ac} \land (c, r) \in R_{CR} \Rightarrow r \in R_{CR}$$

7. Added/Deleted classes: assuming that the class names are unique, changed class names are also considered as added/deleted classes:
$$C^A = C^2 - C^1, C^D = C^1 - C^2$$

8. Changed class: It is a class that exists in both versions of the class diagram but with an added/changed/deleted attributes, method, or relationship. Given the previous definitions we have:
$$C^C = \{c \in C | \exists a \in A, m \in M, r \in R \ (a, c) \in R_{AC}^A \cup R_{AC}^D \cup R_{AC}^C$$
The other 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 [BLS02]. 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). $S_i$ denotes 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 [BLS02] these changes are:

1. Added/Deleted use cases: A use case is identified by its name and a changed name would be recognized as deleted and added change. This class of change can be formalized as:

   $U^a = U^2 \setminus U^1$, $U^d = U^1 \setminus U^2$

2. Changed use cases: The use case exists in both versions but has a changed sequence diagram. A sequence diagram is changed if one of the following holds:

   1. Added/Deleted method: A method is considered as added/deleted if it appears in only one of the versions of the same sequence diagram.
   2. Changed method: A changed method in a sequence diagram invokes a different sequence of methods.

The above changed classes can be summarized using mathematical formalization. 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 to a use case. The impact of added/deleted/changed methods on changed sequences and use cases can be formalized 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 can be classified into obsolete, retestable, and reusable. Let $R_{TS}$ and $R_{TSB}$ denote the relations that traces a test case to a sequence in $S$ and $SB$ respectively. A test case is considered obsolete if it consists of an invalid execution sequence of boundary class methods. This can be caused by 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 $T_o$ can be formalized as:
A retestable test case is a test case that remains valid in terms of a sequence of boundary method executions. Although the methods directly involving in the test case or those that are indirectly triggered by the test case may have changed. The set of retestable test cases $T_{rt}$ can be formalized as:

$$\forall t \in T \ R_{TSB}^1 \in SB^d \Rightarrow t \in T_o.$$  

All the other existing test cases would be classified as reusable. In [BLS02] 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 [BLS02] they should be included in the $R_{AC}$ set.

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. Furthermore, the "ejectCard()" method labeled with 8.2 in the original sequence diagram of Figure 5 is removed from the new sequence diagram. Hence it should be 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 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 [PUA06] 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 [BLS02]
2. Adapting a safe and efficient code based technique [RH93] to UML designs for selecting UML tests.
3. Generating new UML tests using an approach similar to [vMZ99].

This approach is different from [BLS02] in that here they confront the problem of identifying changes that affect UML test cases as opposed to code test cases in [BLS02]. Unlike the other techniques, [PUA06] 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 [RH93] 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:
Table VI. Traceability between Test cases and their path in the original sequence diagram

<table>
<thead>
<tr>
<th>TC #</th>
<th>Message Sequence in the Original Sequence Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1), (2), (3), (3.1)</td>
</tr>
<tr>
<td>2</td>
<td>(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19)</td>
</tr>
<tr>
<td>3</td>
<td>(1), (2), (3), (4), (5), (6), (7), (8), (8.1), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19)</td>
</tr>
<tr>
<td>4</td>
<td>(1), (2), (3), (4), (5), (6), (7), (8), (8.1), (7), (8), (8.1), (7), (8), (8.1), (7), (8), (8.1), (8.2)</td>
</tr>
<tr>
<td>5</td>
<td>(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)</td>
</tr>
<tr>
<td>6</td>
<td>(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), (14.1), (14.2)</td>
</tr>
</tbody>
</table>

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 $D'$.
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 use the OMDAG which was introduced in their earlier work [PAK⁺07]. 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 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 [PAK+07] 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.

Constraint Class Tuples (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. Constraint Class Tuples 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:

\[
\text{CCT}(c) = \langle \{\text{ParentCCT}\} \rangle \backslash \backslash \langle \text{Attribute}\rangle \rangle \backslash \backslash \langle \text{Operation}\rangle \rangle \rangle \langle \text{invariant} \rangle
\]

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

\[
\text{Attribute} = \langle \text{attributename}, \text{attributetype}, \text{visibility}, \text{invariant}, \langle \text{CCT}\rangle \rangle
\]

The Operation tuple is defined as follows:

\[
\text{Operation} = \langle \text{name}, \text{returntype}, \text{visibility}, \text{pre\_condition}, \text{post\_condition}, \{\text{Parameters}\} \rangle
\]

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 [PUA06] 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 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 1 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 design change (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 PDC) 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 design change 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 design change (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 path design change (PDC) or a non-path design change (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 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 path design change 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 [PAK+07]. Mapping design changes 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 a NDCSET. If

![Sequence Diagram](image)

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

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

$$\text{obsolete} = \{TC : \delta_{DC}(TC) = 1 \land DC \in PDCSET\}$$

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 latter 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. Also, 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:

$\text{retestable} = \{TC : \delta_{DC}(TC) = 1 \land DC \in NDCSET \land (DC \in NEWSET \lor DC \in MODSET \lor DC \in DELSET)\}$

$\text{reusable} = \{TC : TC \notin \text{obsolete} \land TC \notin \text{retestable}\}$

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 $[vMZ99]$.  

In this approach, test selection (classification) is based on change identification in a directed graph (OMDAG) which is a simpler form of a control flow graph. Since Rothermel and Harrold $[RH93]$ showed that using a control flow graph would make the test selection algorithm safe, this approach is safe, too. In the end Pilskalns et. al argue that if the number of changes is small this approach is proved to be efficient$^5$.

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 $[PAK+07]$ to the UML design. For traceability purposes, the messages traversed by each test case is listed by a sequence 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.

$^5$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.
Figure 13. OMDAG for the sequence diagram of Figure 5.
We demonstrate this technique on the modified design of the ATM introduced in 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.
Table VII. Test cases for the original Withdraw Cash sequence diagram

<table>
<thead>
<tr>
<th>TC #</th>
<th>Test Case: Values for the Input Attributes</th>
<th>Path in the OMDAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cardValid = False</td>
<td>(1), (2), (3), (3.1)</td>
</tr>
<tr>
<td>2</td>
<td>cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 0</td>
<td>(1), (2), (3), (4), (5), (6), (S₁), (9), (10), (S₂), (15), (16), (17), (18), (19)</td>
</tr>
<tr>
<td>3</td>
<td>cardValid = True, Entered PIN₁ = Wrong PIN, Entered PIN₂ = Right PIN, Invalid PIN Entries = 1, overBalanceEntries = 0</td>
<td>(1), (2), (3), (4), (5), (6), (S₁), (S₁), (9), (10), (S₂), (15), (16), (17), (18), (19)</td>
</tr>
<tr>
<td>4</td>
<td>cardValid = True, Entered PIN₁ = Wrong PIN, Entered PIN₂ = Wrong PIN, Entered PIN₃ = Wrong PIN, Invalid PIN Entries = 3</td>
<td>(1), (2), (3), (4), (5), (6), (S₁), (S₁), (S₁), (S₁), (S₁), (S₂), (S₁)</td>
</tr>
<tr>
<td>5</td>
<td>cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 1</td>
<td>(1), (2), (3), (4), (5), (6), (S₁), (9), (10), (S₂), (S₂), (15), (16), (17), (18), (19)</td>
</tr>
<tr>
<td>6</td>
<td>cardValid = True, Entered PIN = Right PIN, Invalid PIN Entries = 0, overBalanceEntries = 4</td>
<td>(1), (2), (3), (4), (5), (6), (S₁), (9), (10), (S₂), (S₂), (S₂), (S₂), (14.2)</td>
</tr>
</tbody>
</table>

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 OMDAGs for the new version of the Withdraw Cash transaction sequence diagram and its sub-graph S₁.

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 Path Design Changes. 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 design changes.
Figure 16. OMDAG for the sequence diagram of Figure 10.
Figure 17. OMDAG of sub-graph $S_1$ in Figure 16.

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

<table>
<thead>
<tr>
<th>TC #</th>
<th>Test Case: Values for the Input Attributes</th>
<th>Path in the OMDAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4'$</td>
<td>cardValid = True, Entered PIN$_1$ = Wrong PIN, Entered PIN$_2$ = Wrong PIN, Entered PIN$_3$ = Wrong PIN, Invalid PIN Entries = 3</td>
<td>(1), (2), (3), (4), (5), (6), (1), (2), (3), (4), (5), (6), (S$_1$), (S$_1$), (8.3)</td>
</tr>
<tr>
<td>$7$</td>
<td>cardValid = True, Entered PIN = Right PIN, overBalanceEntries = 0, Dispenser Status = Faulty</td>
<td>(1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (9), (10), (S$_1$), (15), (16), (17), (18), (18.1), (19)</td>
</tr>
</tbody>
</table>

- Test case 3 is retestable because the only affected vertex in its OMDAG path is vertex 8.1 which although is in MODSET, is not considered a Path Design Change, 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.
6. Conclusions

This paper 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 design changes are limited to class and sequence diagrams. These are the most common UML diagram types. However, additional techniques for other types of diagrams are also needed. This paper only concerned itself with functional testing, yet many design changes impact other properties like performance, fault tolerance, or security. At this point, no regression testing approaches exist to evaluate the impact of design changes on these design properties and the designer is left without a systematic approach to selectively re-evaluate the effect of design changes. 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


