A Regression Test Selection and Prioritization Technique for TDE/UML

By

Siyuan He, B.Sc., M.A.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

May 2009

Copyright © 2009 by Siyuan He

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-52038-3
Our file Notre référence
ISBN: 978-0-494-52038-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Regression testing is an expensive but necessary activity to validate that existing functionalities of a system still work correctly and have not been negatively affected by the changes made to the system. Meanwhile, UML models are widely used during software development, including when building a test model for a system under test (for instance using the Siemens TDE/UML CASE tool). In this context, we propose a regression test selection and prioritization approach based on analyzing changes made to UML test models. We derive a mapping between model elements (in the test model) and test cases in an existing test suite and then detect changes made to the UML test model to classify test cases into three categories: obsolete test cases (test cases that cannot be re-executed on the new version of the system), retestable test cases (test cases that need to be re-executed on the new version of the system), and reusable test cases (test cases that do not need to be re-executed on the new version of the system). Moreover, the retestable test cases are further prioritized according to the detected changes. Two prioritization criteria are considered, and can be used in isolation or in combination: increasing early fault detection ability of prioritized test cases, reducing the risk of not prioritizing some test cases early in the prioritization. Finally, a case study is performed to evaluate the usefulness of our regression test selection and prioritization approach.

Acknowledgments

First, I would like to give my sincere thanks and appreciation to Dr. Briand and Dr. Labiche for all of their support and guidance throughout my research.

I would like to thank my colleagues in the SQUALL Lab for the kindness support and encouragement.

My warmest thanks to my family for their support and encouragement they have given me. Without them, I would not have been able to get this so far.

Table of Contents

1	Introduction.....	10
2	Related Work	14
2.1	Regression Test Selection	14
2.2	Regression Test Prioritization.....	16
3	Test case Generation Tool: TDE/UML.....	21
3.1	Modeling System Behavior.....	21
3.1.1	Use Case Diagram.....	21
3.1.2	Activity Diagram	23
3.1.3	UML Extensions	27
3.2	Test Case Generation	39
3.3	Metamodel	41
4	TRACEABILITY between Elements and Test Procedures.....	52
5	Changes on Model Elements	55
5.1	Changes in Use Case Diagrams	56
5.1.1	Changes to Use Cases	56
5.1.2	Changes to Use Case Relationships.....	57
5.2	Changes in Activity Diagrams	57
5.2.1	Changes to Nodes	57
5.2.2	Changes to Transitions.....	59
5.2.3	Changes to Notes in the Activity Diagram	60
5.2.4	Changes to Note Connections.....	61
5.3	Changes in Class Diagram.....	61
5.3.1	Changes to Associations	61
5.3.2	Changes to Choices.....	62
5.3.3	Changes to Inheritances	63
5.3.4	Changes to Attributes.....	63
5.3.5	Changes to Categories.....	63
5.3.6	Changes to Notes in the Class Diagram.....	64
6	Classification.....	66
6.1	Obsolete	67
6.1.1	Invalid Functionality	67
6.1.2	Invalid Input/Output Relation	68
6.1.3	Unsatisfied Constraints	74
6.2	Retestable	76
6.3	Reusable.....	80
6.4	Summary	80
7	Prioritization Technique in TDE/UML.....	86
7.1	Test Suites Prioritization.....	87
7.1.1	Requirement Prioritization.....	87
7.1.2	Use Case Prioritization with the AHP Method	88
7.2	Test Case Prioritization.....	91
7.2.1	Expected Risk of a Tested Path	94
7.2.2	Execution Time of a Test Case	101

7.2.3	Multi-Objective Test Case Prioritization	102
7.2.4	Single Objective Test Case Prioritization	121
7.3	Test Procedures Prioritization	121
7.4	Overall sequence for test procedures	122
8	Automation	127
8.1	Tool Overview	127
8.2	Genetic Algorithm Tool	130
9	Case Study	132
9.1	Test Generation	134
9.2	Test Selection	135
9.3	Test Prioritization	140
9.3.1	Single-Objective Approach Result	141
9.3.2	Multi-Objective Approach Result	145
9.3.3	Effectiveness	155
9.4	Conclusions of Case Study	165
10	Conclusion	168
11	References	171
Appendix A	Addendum to OCL Expressions	175
A.1	Description of Class Helper	175
A.2	Constraints on the Metamodel	179
A.3	Traceability links for Model Elements	183
A.4	OCL Expressions for Model Element Changes	186
A.4.1	Changes to use case relationships	186
A.4.2	Changes to Nodes	186
A.4.3	Changes to transitions	187
A.4.4	Changes to Notes in the Activity Diagram	187
A.4.5	Changes to Note Connections	188
A.4.6	Changes to Associations	188
A.4.7	Changes to Choices	188
A.4.8	Changes to Inheritances	189
A.4.9	Changes to Attributes	189
A.4.10	Changes to Categories	189
A.4.11	Changes to Notes in the class diagram	190
A.5	The rationale for fitness function f_3 in optimization scheme two.	190

List of Figures

Figure 1: Example of a Use Case Diagram [1]	22
Figure 2: Properties for a use case	23
Figure 3: An example of activity diagram for a use case [1].....	24
Figure 4: Activity with stereotype <<Nested>> [1]	25
Figure 5: Properties of a node	26
Figure 6: Properties of a transition.....	26
Figure 7: Example of Category and Choice [1].....	28
Figure 8: Attribute of a category.....	29
Figure 9: Properties of a choice	29
Figure 10 Example of an activity diagram in TDE/UML [1]	31
Figure 11: Constraints in a Note and Global Constraints [1].....	33
Figure 12: An activity with fork and join nodes	37
Figure 13: An example of TDE properties [1].....	38
Figure 14: Example of configuration options	40
Figure 15: Example of a test suite.....	41
Figure 16: metamodel for TDE/UML (part 1).....	43
Figure 17: metamodel for TDE/UML (part 2).....	48
Figure 18: An example of activity diagram	53
Figure 19: inserting/deleting decision nodes	70
Figure 20: Example of choices with probabilities	96
Figure 21: A crossover example	104
Figure 22: A mutation example	104
Figure 23: SPEA2 algorithm [35]	109
Figure 24: Example of the PMX operator.....	113
Figure 25: The algorithm to find the number of executed test cases for optimization scheme one.....	115
Figure 26: Values of f1 for test case orders one and two (Table 9).....	117
Figure 27: The algorithm to find the number of executed test cases for optimization scheme two	118
Figure 28: Values of f3 for test case orders one and two (Table 9).....	120
Figure 29: the result of test cases prioritization.....	125
Figure 30: Cumulative expected risk and test suite fraction.....	126
Figure 31: the algorithm to find the stop point for each test suite	126
Figure 32: Steps of Test Selection and Prioritization Tool.....	128
Figure 33: Packages of Test Selection and Prioritization Tool.....	129
Figure 34: Extensions of the ECJ framework	131
Figure 35: Use case diagram of the ATM system.....	132
Figure 36: result of single objective prioritization approach	143
Figure 37: The distribution of retestable test cases' execution time (ms)	146
Figure 38: Distribution of solutions for scheme one	149
Figure 39: curves of test sequence A and B.....	149
Figure 40: Distributions of solutions for scheme two.....	154

Figure 41: Distribution of APFDc scores for random test sequences with 65 executed test procedures	161
Figure 42 Justifying fitness function f_3	192

List of Tables

Table 1 : syntax of the Constraint [1]	32
Table 2: Test Suite for activity diagram in Figure 4.....	53
Table 3 : An example of traceability matrix	54
Table 4: Relations between change rules and classification rules	84
Table 5: Deletion/addition/change of model elements we account for.....	85
Table 6: Scale for pair-wise comparisons	90
Table 7: An example of pair-wise comparisons	91
Table 8: Normalized comparison matrix	91
Table 9: Example <i>nERisk</i> and time values of a test suite	116
Table 10: Test Generation result.....	135
Table 11: Detected changes in the first change	137
Table 12: Detected Changes in the second change.....	138
Table 13: Detected Changes in the third change	138
Table 14: Selection result.....	139
Table 15: Changed Elements	141
Table 16: User defined values and stop points	145
Table 17: Summary of results for optimization scheme one	147
Table 18: optimization scheme one VS random test sequences	151
Table 19: summary of results of scheme two	152
Table 20: Optimization scheme two VS random test sequences	155
Table 21: APFD _c scores for the single-objective approach and 100 random test sequences.....	161
Table 22: APFD _c scores for optimization scheme one and 12837 random test sequences	163
Table 23: APFD _c scores for optimization scheme two and 32398 random test sequences	164

1 INTRODUCTION

In the iterative and incremental development process, changes to a software system are made at each iteration. Test efforts should not only be spent on changed or added parts of software, but also on the un-changed parts since they could be affected by the changes. Some existing features of a system, which had correct functionalities previously, may no longer work as intended after changes have been made to the system. Regression testing is such test activity that tests the existing parts of a system to guarantee they still work correctly after a system has been changed.

In practice, regression testing requires a lot of efforts if testers re-run all test cases in the existing test suite. Various regression testing techniques have been proposed to reduce the cost of regression testing. The regression test selection technique is one of those techniques. It selects a subset of an existing test suite that is still valid based on the analysis of changes of a system, for example identifying differences between two versions of source code, and then only executes the selected test cases to reduce the cost of regression testing. Similarly, regression test minimization techniques reduce the cost of regression testing by selecting the minimum subset of an existing test suite such that it can still achieve the same coverage in terms of certain test criterion, such as statement coverage of source code, as the original test suite. Another regression testing technique is test case prioritization, which ranks test cases in the test suite according to certain criteria so that test cases with higher ranks are selected to execute earlier than those with lower ranks during regression testing. Therefore, running a prioritized test suite could achieve testing goal earlier than un-prioritized test suite.

Meanwhile, as the Unified Modeling Language (UML) becomes a de-facto standard in industry, UML-based testing techniques have been proposed. A UML model is used to describe the behavior of a system under test, and test cases are derived from the UML model. In recent years, Siemens Corporate Research has implemented a testing tool, named TDE/UML, to realize such UML model based testing technique. It can derive test cases from a UML model, which includes a use case diagram, activity diagrams and class diagrams [1]. The use case diagram provides a top-down perspective of the functionality of a system. Each use case is associated with an activity diagram, which specifies system behavior in the use case. Beyond the standard UML model, some annotations are added to the UML class diagrams to specify the test requirement (i.e. input data). From these annotated UML diagrams, TDE/UML generates test cases to validate system functionalities.

A TDE/UML model needs to evolve when users want to change their requirements, including modification of existing features, addition of new features, or correction of errors. Then, a research question is to identify whether an analysis of changes made to the TDE/UML model can facilitate regression testing.

On the one hand, as a system evolves, the size of its test suite can become gradually substantial. Meanwhile, some test cases might become invalid and should be removed from the test suite. Therefore, regression test selection is necessary. Here, we are focusing on how to guarantee that existing functionalities are not affected by changes to the system. We intend to define a model-based regression test selection technique specifically tailored to this context which classifies test cases into one of the three following categories [2]:

- **Obsolete:** A test case that can not be executed for the new version of a system since it has become invalid.
- **Retestable:** A test case is valid and needs to be rerun due to the changes on the system.
- **Reusable:** A test case is valid but does not need to be rerun.

On the other hand, beyond test selection, another issue is the prioritization of test cases. Regression testing is an expensive activity: it is costly to rerun all regression test cases. Under some time and resource constraints, testers would like to execute test cases that are important to them. For example, some test cases are important to testers because they are deemed more effective at revealing faults in a program. To address this problem, a prioritization technique is applied to the regression test suite to order test cases according to several criteria. Test cases with higher priorities are placed in the front of the test order and are executed earlier during regression testing than test cases with lower priorities. To facilitate the regression testing in TDE/UML, we therefore propose a prioritization technique to order the retestable test cases obtained from the regression test selection.

The approach presented in this thesis assumes there is traceability between model elements in the original model and test cases: i.e., we can determine which test cases are affected by changes to a TDE/UML model through traceability information. Another assumption is that TDE/UML models, including use case diagrams, activity diagrams and class diagrams, should be internally consistent. For instance, if an association in a class diagram is deleted, any guard condition where the association is used in an activity diagram should be updated.

The contributions of this thesis are the following:

- A comprehensive study of TDE/UML model changes and their impact on regression test selection;
- A model-based regression test selection approach for TDE/UML;
- A model-based test case prioritization approach for TDE/UML that accounts for different optimization criteria and is solved with a multi-objective genetic algorithm;
- The implementation of the change identification, test case selection and prioritization approaches as a plug-in to TDE/UML;
- An initial proof of concept case study.

The rest of this document is organized as follows: In the Section 2, the related work on regression testing is discussed. Then we discuss the modeling and testing aspects with TDE/UML in Section 3. The approach to identify the traceability between TDE/UML model elements and test cases is presented in Section 4. In Section 5, we discuss changes to the system model. The regression test selection technique is presented in Section 6. The prioritization technique is presented in Section 7. Section 8 describes our tool support for regression test selection and prioritization (extension of TDE/UML). Section 9 reports the result of a case study. Finally, Section 10 gives the conclusion and describes the future work.

2 RELATED WORK

As our prioritization technique combines both regression test selection and regression test prioritization, previous work in these two fields are discussed in the next two sub-Sections, respectively.

2.1 Regression Test Selection

In previous work, a number of the code-based and model-based regression test selection techniques have been developed.

In [3], Rothermel et al. developed a code-based regression test selection technique to select a subset of test cases from an existing test suite. They selected test cases that exercised the new or modified statements in the new version of the program or formerly executed the deleted statements. The result of empirical studies suggested that this approach was efficient in reducing the cost of regression testing.

As UML has become the de-facto standard in the industry, some regression test selection techniques have been developed based on changes to a UML model. In [4], Briand et al. identified the design changes between two versions of a UML 1.x model (with use case diagram, sequence diagram and class diagram). They also relied on a mapping between design changes and test cases. Then, using the mapping information, they performed an impact analysis of design changes to classify test cases of the existing test suite into one of three categories: obsolete, retestable and reusable. Only test cases in the retestable category were selected to rerun. The main difference between our approach and Briand et

al.'s approach is the usage of the UML model. UML models used in Briand et al.'s approach are analysis and design models that aim to bridge the gap between requirements and design. However, UML models in the context of TDE/UML are test models and they are used to derive test cases for the system under test.

In [5], Chen et al. presented an approach to perform regression test selection based on a UML activity diagram. They built the traceability between edges and test cases and identified the affected edges caused by modifications of an activity diagram. Then, using the traceability between edges and test cases, the test cases that execute the affected edges are selected and rerun during regression testing. This approach has some limitations. First, for the changes of an activity diagram, they only considered the addition and deletion of nodes and edges, but did not include changes to other model elements, such as guards on the edges. Second, they only used the activity diagram to specify a system's behavior while recent approaches combine the use of several types of UML diagrams.

In[6], Chen et al. proposed a regression test selection method based on the analysis of a Extended Finite State Machine (EFSM), which is used to capture the expected behavior of a state based system. This test selection method first identifies a set of modifications between two versions of an EFSM, including additions, deletions and changes of transitions. Then it analyzes the data and control dependences of modified transitions (i.e. deleted, added or changed transitions) and identifies interaction patterns for each modified transition. Finally, the test selection method excluded test cases that do not cover any interaction pattern from the existing test suite. The authors reported that their method could produce encouraged results on systems modelled by EFSMs.

Pilskanlns et al. developed a technique to test a UML design model (with a class diagram, sequence diagram and OCL statements) [7]. Based on this UML design testing technique, they went further to perform regression test selection when a UML model evolves [8]. They identified the design changes in the different UML views and then classified test cases of the existing test suite based on the design changes. Similar to [4], they classified test cases into three categories: obsolete, retestable and reusable. Unlike other regression test selection methods discussed previously [3-5], which focus on the test cases used to detect faults in the code of a program, Pilskanln et al.'s method focuses on test cases used to detect faults in a UML model.

2.2 Regression Test Prioritization

In previous work, researchers have developed various prioritization techniques for regression testing.

In [9], Rothermel et al. investigated six prioritization techniques for regression testing. Four of them were coverage based techniques that ranked the test cases based on the coverage of the structural entities of a program (i.e. statements or branches). The other two were fault based techniques that ordered the test cases based on their fault detection ability. In the fault based techniques, the fault-exposing-potential (FEP) must be estimated by mutation analysis. These six techniques fall into two categories. The first one consists in ordering test cases according to their coverage or FEP (e.g., the highest coverage first), whereas the second one consists in ordering test cases according to their cumulative coverage or FEP. Empirical studies showed that these prioritization techniques were effective at early detection of faults during regression testing. Empirical

studies also showed that accounting for cumulative coverage or FEP was better but is more expensive.

In [10], Srivastava and Thiagarajan presented a code-based test prioritization system, named Echelon. The Echelon system has been integrated into the Microsoft software development process. The set of impacted blocks were defined to identify the changes between two versions of a program at the binary code level. Then Echelon ordered the test cases based on the additional coverage of impacted blocks so that the ordered test cases can quickly achieve the maximum coverage of the impacted blocks. The studies showed that the Echelon system was capable of effectively detecting faults in the early phase in the Microsoft software development process.

In [11], Kim and Porter proposed a different prioritization technique that used the historical execution data for regression testing. This prioritization technique firstly performed safe regression test selection to select a subset, T' , from the existing test suite T . And then, the test cases in T' were ranked according to historical execution data, which can be test case execution history or fault detection history. These historical data were recorded in every testing session. One of the prioritization criteria was based on execution history. During test case prioritization, higher priorities were given to the test cases that had not been executed in the recent testing sessions. Another criterion was based on the fault detection history. Test cases were given a higher rank if they had detected more faults in recent test sessions. As this prioritization technique was designed for resource constrained environments, the case studies showed that this approach was cost-effective in such environments.

A genetic algorithm was recently applied to the test case prioritization problem [12]. The genetic algorithm is a rank-based fitness approach, to prioritize test cases for regression testing purpose. Individuals (i.e., test cases) are ranked according to their objective function values such that the individual with highest objective function value is ranked first. Then the fitness of each individual is given by its rank position in the population. The fitness function used in this approach was: $Fitness(Pos) = 2 \cdot (Pos - 1) / (Nind - 1)$, where the `pos` was the rank position in the population and `Nind` was the size of the population. An empirical study was conducted to compare the genetic algorithm with other prioritizing algorithms (presented in [9] and already discussed above), the 2-optimal algorithm and the Hill climbing algorithm, to evaluate their effectiveness in early achieving maximum coverage of code elements. The results showed that the genetic algorithm can not outperform the other five algorithms in all cases. But in most case, the genetic algorithm yielded fairly good results.

In [13], Kristen, et. al. also proposed a regression test prioritization approach that applied a genetic algorithm to prioritize test cases when time is a constraint. In their approach, the weighted sum approach was used to combine both the code coverage and execution time of each test case into a single objective function by adding them together. The code coverage part was multiplied by a weight to ensure that achieving maximum code coverage was always the primary objective. But the drawback of this approach was that the value of weight given to the code coverage can not be easily and precisely selected. The choice of different weight values could produce quite different results.

Recently, some researchers developed a prioritization technique based on changes of a system model. In [14], B. Korel et al. proposed a test case prioritization technique based

on the changes to an extended finite state machine (EFSM) model. Their approach first identified a set of modified transitions in two versions of the model, which could be added transitions or deleted transitions. Then two prioritization methods were proposed to improve the effectiveness in early fault detection of a test suite. In the first method, test cases that executed modified transitions were given higher priority while test cases that did not execute any modified transitions were given the lower priority. In the second method, a model dependence analysis, which captures both data dependence and control dependence between transitions, is performed for modified transitions to identify how those modified transitions interact with other transitions of the model. Then the higher priority test cases obtained from the first method are further prioritized based on the result of the proposed model control and data dependence analysis (see [14] for details). The result of an experiment showed that two prioritization methods can improve the effectiveness of early fault detection of a test suite as compared to the random prioritization technique. In [15], Korel et al. went further to develop several heuristic methods based on the modified transitions in the system model. These heuristics mainly used the information collected during the execution of an EFSM model to prioritize test cases, such as the number of modified transitions executed by a test, the frequency of execution of modified transitions or the number of times that a modified transition has been executed after running a portion of a test suite. The result of experiments indicated that these heuristics improved the effectiveness of early fault detection compared to random prioritization, except the one considering the frequency of execution of modified transitions.

All the prioritization techniques listed above are based on the analysis of either source code of a given program or a system model. Recently, in [16], Srikanth H. et. al. proposed an approach to prioritize test cases from requirements. They took four factors into consideration when prioritizing requirements. The first factor is the priority of requirements. Customers assigned this value to measure the importance of a requirement. The second factor is requirements volatility, which measures how frequently a requirement has been changed during the development cycle. The third factor is the implementation complexity of requirement. The developer is responsible to assign a value to each requirement, based on their judgments, describing the implementation complexity of the requirement. The last one is the fault proneness of requirements, which is measured by the number of test failures found in the previous code. Then they used the weight-sum approach to integrate these four factors to compute a Prioritization Factor Value (PFV) for each requirement. Based on a mapping between requirements and test cases, a Weighted Priority (WP) is derived for each test case based on the PFV values of requirements that the test case maps to and the number of requirement that the test case exercises. Then the test cases are ordered in the descending order of the WP values. The result of case studies showed that their approach improves the rate of fault detection over the random prioritization approach. And the priority of requirements was shown to be the most important factor that contributed to the improvement of fault detection ability.

3 TEST CASE GENERATION TOOL: TDE/UML

This Section first introduces some terminology associated to the use of TDE/UML. A *test suite* is a group of test cases. A *test case* is associated with a set of use cases and is a path in a graph which merges activity diagrams of the tested use cases. A *test step* is an activity along a path of a test case. A *test procedure* is an instance of a test case, i.e., a path in the corresponding activity diagrams plus values associated to the variables used in the activity diagrams. Each test procedure of a test case has the same sequence of test steps but different input data.

In Section 3.1, we describe how to use the use case diagram, the activity diagram and some UML extensions to model a system with TDE/UML. Then, the test case generation aspect of TDE/UML is introduced in Section 3.2. All the TDE/UML model elements, concepts and their relationships are formalized into a metamodel (Section 3.3). TDE/UML model elements are simply referred to as model elements in the rest of this document.

3.1 Modeling System Behavior

3.1.1 Use Case Diagram

Modeling with TDE/UML begins with an analysis of the textual use case specifications which describe the user (or environment) stimuli and system responses step by step. Normally these use case specifications are identified by their names and provide information on include or extend relationships between them. An included use case

describes a behavior of the system that can be shared by other use cases. When two use cases are related by an include relationship, the including use case (or base use case) contains the flow of events of the included use case. In the case of an extend relationship, the extending use case describes some additional behavior the extended (or base) use case may include under certain condition. Normally the extend relationship is used to model exceptional behavior¹. The identified use cases and their relationships are modeled in a use case diagram. A use case diagram can provide a top-down perspective of the system functionality, including the stereotype <<include>> to present the include relationship and stereotype <<extend>> to present extend relationship.

As an example, Figure 1 shows an example of a use case diagram. The top level use case “UseCase A” includes one use case and can be extended to three other use cases.

Figure 1: Example of a Use Case Diagram [1]

In addition, as shown in Figure 2 for use case UseCase A, TDE/UML allows the definition of some properties associated with a use case (in the “Properties” tab), such as

¹ Although include, extend, and generalization relationships between use cases are part of UML, we can only model the first two types of relationships in TDE/UML [17].

Abstract (whether the use case is abstract or not), Description (textual description of the use case), and stereotype. The usage of these properties is not clearly stated in [1] and has to be clarified with Siemens. Note that a use case has an `Id` property, which is set by TDE/UML to uniquely identify use cases.

Property	Value
Abstract	No
Color	Yellow
Description	
Id	1183057385586
Name	UseCase A
Package	Use Case Package
Stereotype	

Figure 2: Properties for a use case

3.1.2 Activity Diagram

In TDE/UML, each use case in the use case diagram is associated with an activity diagram to describe the interaction between the system and its actors. If a (base) use case has an include or extend relationship with another use case, the included or extending use case should appear as one activity (it has the same name as the included or extending use case) with stereotype “Use Case” in the activity diagram of the base use case. For instance in Figure 3, the use case “UseCase C” (extended use case of “UseCase A” in Figure 1) becomes an activity in the activity diagram of use case “UseCase A”.

Figure 3: An example of activity diagram for a use case [1]

Moreover, an activity in the activity diagram with stereotype “nested” can be refined by creating a sub-diagram to specify behavior in more details. For instance in Figure 4, activity “change sound and settings” is annotated with <<Nested>>. It will be associated with another activity diagram that further describes its behavior. Compared with activities annotated with <<use case>>, activities that are annotated with <<Nested>> are also refined by another activity diagrams but they don’t refer to any use case.

Figure 4: Activity with stereotype <<Nested>> [1]

The basic standard elements in activity diagram can be categorized into nodes (activity, decision, start, end, fork and join nodes), edges (i.e., transitions), and guard conditions on the transitions. In addition, there are two kinds of end nodes in TDE/UML: stop nodes and final stop nodes. Both of them represent the end points of activity diagrams. However, when TDE/UML generates test cases (see section 3.2), if a test path exercises a lower level activity diagram, which describes an activity with <<Nested>> stereotype or an included/extending use case, it will exit when it reaches at a final stop node (the test path ends). However, if a test path reaches a stop node, it will continue to exercise a higher level activity diagram (similarly to a return statement in a piece of source code). In most cases, final stop nodes are used to model the ends of exception scenarios.

Although the swimlane is also a standard element, it has no impact on the test cases generation in TDE/UML. So currently, we don't consider it in our methodology.

Similarly to use cases, we can also specify some properties for a node or a transition in the “Properties View” with TDE/UML. For instance in Figure 5, a node is associated with properties “Description”, “Do Action”, “Stereotype”, “Use Case”. As shown in Figure 6, we can specify property “name” for a transition. The “stereotype” of a Transition is set to “Transition” automatically by the tool. Again the usages of these properties are not clearly described in [1]. Note that the “Use Case” property of an activity is only used when a node refers to a use case (and therefore to an activity diagram). Activities and transitions are also given a unique identifier (Id) by TDE/UML.

Property	Value
Color	Yellow
Description	
Do Action	
Id	1187720114456
Name	Power Up
Package	Use Case Package
Stereotype	nested
Use Case	Power Up

Figure 5: Properties of a node

Property	Value
Guard Expression	rb = AM
Id	1183057385696
Name	
StereoType	Transition

Figure 6: Properties of a transition

3.1.3 UML Extensions

Beyond the basic UML elements discussed previously, TDE/UML adds some extensions to specify the input data used during test case generation.

3.1.3.1 Categories and Choices

The category-partition method [18] is used in TDE/UML to model data being used in activity diagrams. This information is then used during test case generation. In TDE/UML, categories and choices are defined in a UML class diagram. A category is represented by a class in a class diagram, and choices of a category are specified as attributes of the (category) class. For example, in Figure 7, a category `Colour` is defined, including four choices: `Red`, `Blue`, `Yellow`, and `Green`. Similarly to classes in any UML class diagram, and especially domain models [19], categories can have inheritance and association relationships. The class diagram notation offers the possibility to specify the refinement of categories (choices) by means of class generalization/specialization relationships. In Figure 7, as Category `OddShapes` is the subclass of Category `Shapes`, it has a total of seven choices (four of them are inherited from its base class `Shapes`). Association relationships allow the designer to decompose a class (i.e., category) into small pieces and specify related categories. In Figure 7, the association position is created between category `Shapes` and category `Location`. If a variable, says of type

Shapes is used in an activity diagram, variable `s.position` of type `Location` can also be used².

Figure 7: Example of Category and Choice [1]

Similarly to nodes and transitions in activity diagram, categories and choices are given unique integer number as identifiers. In addition, choices have unique names in a category and categories have unique names in a class diagram.

Like other classes in the class diagram, categories can also have attributes. For an attribute of a category, each choice of the category has a value of the attribute. The attributes of a category and their associated values can be used in constraints (Section 3.1.3.1). As shown in Figure 8, a category `Size` has an attribute `inches` and three choices (`big`, `medium` and `small`). Then each choice is associated with a value for the attribute `inches`.

² From the “TDE/UML user manual”, a class (i.e., a category) can have choices as well as attributes, but it is not clear whether these are two different kinds of characteristics of a class (category). The usage of association is not mentioned too.

Every category has a default attribute probability, as shown in Figure 9³. The value of attribute probability of a choice in a category captures how frequent is the choice in practice. During test case generation, TDE/UML will select choices as the values for input variables to satisfy certain choice coverage criteria (these choice coverage criteria are discussed in Section 3.1.3.4). The property “probability” is applied when “sampling” (see the description in Section 3.1.3.4) is chosen as the choice coverage criterion. In this case, a choice is selected by TDE/UML according to its “probability”.

Size	
inches	
big medium small	
Name	inches
big	200
medium	12
small	1

Figure 8: Attribute of a category

Size	
inches	
big medium small	
Property	Value
Choice	
Probability	60
Value	
General	
Color	Yellow
Description	
ID	1216822590919
Name	big
Package	Logical Package
Stereotype	

Figure 9: Properties of a choice

³ Note that the probability is represented in the “Properties View” in TDE/UML. In Figure 9, there are some other properties except probability defined for a choice, such as value, description and stereotype. But the usages of these properties are not clearly described in the user manual [1].

3.1.3.2 Variable Declarations

TDE/UML combines input data information with the UML notation by declaring variables in notes attached to activities. The type of an input variable is a category in the class diagram of categories and choices. Values of the input variable are the choices of the type (category) of the variable. In Figure 10, variable *n* of type *Name* is declared in a note attached to activity *Print Name #1* following a specific syntax: `define VariableName CategoryName`. As a result, if the *Print Name #1* activity is executed during a test, it will require an input named *n* of type *Name* and select one of the choices of category *Name* as the input value for variable *n*. Notes used for variable declarations should be stereotyped `<<TDE/UML>>` to indicate that they will be used by the TDE/UML test generator.

The scope of a variable is by default the activity diagram where it is defined. There is one exception though. As discussed in Section 3.1.2, an activity can be refined by another activity diagram (when a base use case has an extension or included use case). A variable defined in a base use case's activity diagram can be used (with the same name) in the activity diagram of the included/extension use case. This is done by using keyword `use` instead of `defined` in the activity diagram of the included/extension use case. In Figure 10, which shows the activity diagram of an included/extension use case, activity *Print Name #2* declares the use of an input *c* (of type *Colour*) by means of the `use` keyword, thereby specifying that variable *c* is defined in another activity diagram (the one of the base use case).

Figure 10 Example of an activity diagram in TDE/UML [1]

3.1.3.3 Constraints

Three types of constraints can be specified on the variables in TDE/UML: Guard conditions, Constraints on nodes (specified in a note), and Global constraints.

In TDE/UML, a guard condition can constrain a transition. Terms involved in a guard condition could be⁴: a variable, a choice or a constant string. Moreover, a special term that represents a variable with an attribute can also be used. For example, “s.inches” (*s* is a variable and *inches* is an attribute of a category) is a term that refers to the value of the attribute *inches* that is associated with variable *s*. A specific syntax has to be used: the

⁴ This terminology is the one adopted in the TDE/UML user manual and may not correspond to the one the reader is familiar with (for instance from Boolean logic theory). We however decided to keep the TDE/UML terminology.

only operators available are summarized in Table 1. These guard conditions, specify which behaviors are possible (or legal) in an activity diagram (i.e., use case), i.e., which paths in the activity diagram are feasible. For example, in Figure 10, the outgoing transition of activity `Print Name #2` has a guard `n=n1` which means only test case that takes `n1` as the value of variable `n` can pass through this path (`n` is a variable of type/category `Name` and `n1` is one of its choices).

More specifically, a guard expression that uses keyword `exists` or `forall` is evaluated by using all the variables typed by a given category (the category constrained in the `exists` or `forall` expression). An expression using keyword `exists` evaluates to true if any variable with a given category type make the expression true. For example, expression `exists c: Color | c = red` evaluates to true if there exists a variable of type `Color` that is red. A guard expression using keyword `forall` evaluates to true if all the variables of a given category type make the expression true. For example, expression `forall c: Color | c = red` evaluates to true if all the variables of type `Color` are red color.

Operator/keyword	Description
=	Equal
!=	Not equal
and, or, not	Logic operations
implies	Logic implies
()	Parentheses to show the precedence
exist	The expression followed by “exist” to be true only if any of the variables of a given category type make the expression true.
forall	The expression followed by “forall” must be true for all the variables of a given category type.

Table 1 : syntax of the Constraint [1]

Another kind of constraints is defined in notes following variable declarations. As shown in Figure 11 (a), the constraint for variable *c* of type *Colour* is specified after its declaration. The category *Colour* is defined in a class diagram and has four choices. But as specified in the constraint, variable *c* can only take either the *Red* or *Blue* values. It is possible to specify multiple constraints on separate lines in a note.

Figure 11: Constraints in a Note and Global Constraints [1]

A global constraint can apply to many variables in all the activity diagrams and is defined in a note to be placed in the class diagram (where categories and choices are defined). Since a global constraint can apply to any subset of variables in all activity diagrams, global constraints typically use keywords *exists* or *forall*. As shown in Figure 11 (b), a global constraint for variables of type *Shape* and variables of type *Colour* is declared. The constraint specifies that every variable of type *Shape* that has value *Triangle* is *Red*. To facilitate the analysis of the changes to global constraints, we assume there is only one global constraint specified in one note. As a result, each global constraint is identified by a unique integer number. Similar to notes used for variable declarations, the notes used to specify global constraints should be annotated by stereotype `<<TDE/UML>>`.

3.1.3.4 TDE Properties

In TDE/UML, different test generators are available. The test generator to be used can be specified in the TDE/UML model, as a TDE Property (named test-generator). Additionally, each test generator is associated with a list of TDE properties which specify the configuration information for test case generation. Currently, two kinds of test generators are available: TdeTestGenerator and BtgTestGenerator, though only TdeTestGenerator can be used in all example projects [1]. We therefore only discuss the TDE properties associated with TdeTestGenerator.

TDE properties are specified in notes attached to nodes in activity diagrams. Each property has a name and a list of possible values. The properties of TdeTestGenerator are:

- **coverage-algorithm:** This property is associated with a refined activity node. It specifies the graph coverage requirement of the activity diagram which describes the behavior for the nested use case during test case generation. As different paths in the activity diagram are used to generate test cases, the test case generator chooses paths based on the coverage criteria specified by this property. This property has five possible values:
 - **all-paths:** The test case generator chooses all the possible paths to create test cases. In case of loops in the activity diagram, the generator only traverses the loops once to prevent infinite looping.
 - **all-activities:** The generated test set, i.e., the set of generated test cases, should contain all the activities in the activity diagram (if possible).

- `all-transitions`: The generated test set, i.e., the set of generated test cases, should contain all the transitions in the activity diagram (if possible).
- `round-trip`: Similar to “all-paths”, the test generator will choose all possible paths except that it doesn’t include the loops in the activity diagram.
- `happy-path`: The test generator chooses only paths that contain one or more nodes that are marked with the `happy-path` property. The `happy-path` property is another TDE property and will be introduced later.
- `choice-coverage-algorithm`: This property is also associated with a nested activity. It specifies the data coverage requirement of the variables used in the activity diagram during test case generation. The test generator has to combine the choices of variables based on this property. There are six values for this property:
 - `sampling`: This applies to a variable. The test generator randomly selects a choice for a variable used in a path.
 - `choice-per-suite`: This applies to all the variables in the test suite. The test generator ensures that all choices of a variable are used in the test suite.
 - `choice-per-variable`: This applies to a variable. The test generator ensures that all choices of the variable are used in the test cases that use this variable.
 - `choice-per-path`: The test generator ensures that all choices for a variable are used in each generated path.
 - `exhaustive`: This applies to a collection of variables. The test generator chooses all possible combinations of choices for variables used in generated test paths.

- `pair-wise`: This applies to a collection of variables. The test generator chooses all combinations for every two variables used in generated test paths.
- `sync-coverage-algorithm`: This property is associated with a fork or join node in an activity diagram. Fork and join nodes are used when we try to model parallel execution of some activities. But a test case is a (linear) sequence of activities that does not allow the specification of concurrent executions. Based on this property, the test generator decides which parallel activities should be involved in a test case and the order between these parallel activities. We only discuss two of the four possible values of this property available in TDE/UML:
 - `one-sequence`: The test generator involves all the activities between a fork node and a join node and arranges them in an arbitrary order. Note that the order of activities in one path (between the fork and the join) remains the same (they indicate sequential executions) and the sequence is not interleaved with other fork to join paths. Using Figure 12 as an example, all the activities between the fork and the join nodes will be involved in a test path, in a random order, with the specific constraints that activity 5 always follows activity 2 and other activities are not interleaved between them.
 - `combinations`: The test generator chooses a random combination of activities between fork and join nodes to construct test path. The number of branches taken into the test path is decided by the `sync-combination-length` property (see below). This means TDE/UML might not take all branches between a

fork and join pair to generate a test path. This is not intuitive, as since does not correspond to the semantics of forks and joins.

Figure 12: An activity with fork and join nodes

- **sync-combination-length:** This property is used in combination with value combinations of property **sync-coverage-algorithm**. It determines the number of outgoing branches of the fork node that are involved in a test path. If it is set to a number above (or equal to) two, the number of outgoing path from the fork node is a random number between two and this number. For instance in Figure 12, if the **sync-coverage-algorithm** is set to **combinations** and the value of **sync-combination-length** is two, then activity 2, activity 5 and activity 3 can be selected in a test path or activity 3 or activity 4 are chosen in a test path, but activity 2, activity 3 and activity 4 will not be selected together.

- **happy-path:** If a node in the activity diagram is associated with this property and the value is `true`, then any path that traverses this node is considered as happy-path.
- **require-activity:** This property specifies that an activity should be contained before another activity in a path. If the specified required activity is not in a path, then the path can not be used during test case generation.

Figure 13: An example of TDE properties [1].

As an example, consider the activity diagram in Figure 13. Activity UseCase B, which references another activity diagram (as indicated by the `<<use case>>` stereotype), is associated with two TDE properties. These TDE properties specify the coverage algorithm and choice coverage algorithm used in the refined activity diagram during test case generation.

These TDE properties, when specified in an activity diagram, only affect this activity diagram. They can also be defined in the TDE configuration editor, in which case, they affect all the activity diagrams.

3.2 Test Case Generation

During test case generation, TDE/UML starts from an initial use case, determined by the test designer, to generate a test suite. An initial use case can be any use case in the use case diagram, depending on which system behavior the tester wants to test. A graph representing the activity diagram of the tested use case is built. In case of refined activities, i.e., activities stereotyped <<Use Case>> or <<Nested>>, the graphs of the refined activities (i.e., the different activity diagrams) are merged. We can define the *refinement level*⁵ as the distance in terms of activity diagram, between the base (tested) activity diagram and the refined activity diagram. For instance, if use case A includes use case B and B is extended by use case C, then the refinement level of B (wrt A) is 1 and the refinement level of C (wrt A) is 2. During the construction of the graph, the tester can control the number of refinement levels that are considered by the algorithm using the configuration option `refinement-limit`: only refinement levels up to the value of `refinement-limit` are considered during graph construction. As shown in Figure 14, this option is set to 1, which means the test generator only accounts for the use cases from the initial use case and its next refinement level use cases in the use case diagram. After the graph is constructed, the test generator chooses different paths in the graph to construct test cases according to the `coverage-algorithm` property which specifies the

⁵ Note that the term “refinement level” is not part of TDE/UML but we introduce it here for clarification purposes. The term “refinement-limit” is a configuration option for test generation in TDE/UML

coverage requirement of the graph. A *test case* is then associated with a sequence of *test steps*, i.e., a sequence of activities along a path in the graph. Note that in TDE/UML, the notion of test case does not include test input values (see below).

Test Generator My Configuration Options

Property ID	Property Name	Element	Value
coverage-algorithm	Coverage Algorithm	Default	round-trip
refinement-limit	Refinement Limit	Default	1
max-temporary-...	Max Temporary Paths	Default	32167
sync-coverage-a...	Sync Coverage Algorithm	Default	one-sequence
sync-combinatio...	Sync Combination Length	Default	2
choice-coverage...	Choice Coverage Algorithm	Default	choice-per-suite
ignore-errors	Ignore Errors	Default	no
require-activity	Required Activity	Default	

Figure 14: Example of configuration options

TDE/UML uses the category-partition method [18] to generate the test data for test case. The category-partition method divides the input data into different categories. Each category is further defined by all possible choices. Choices in a category can be data values, references to another category or combination of both. During modeling (see Section 3.1), input variables that have a type defined by a category are specified by the test designer. The values of input variables are the choices of categories. Not all of choices can be selected for input variable since variables are constrained in activity diagrams (guard conditions, variable constraints in a note, global constraints). Additionally, input value selection depends on the data coverage criterion specified by TDE property `choice-coverage-algorithm` (Section 3.1.3.4). In TDE/UML, we refer to a test case with input values as *test procedure*: different test procedures, with different input values, can be generated for on specific test case.

Figure 15 shows a specification of a test suite for use case `Radio` generated by TDE/UML. This test suite contains three test cases, namely `Radio_TP0`, `Radio_TP1`, `Radio_TP2`. Each test case has a different sequence of test steps and is identified by a unique (integer) identifier (last column: Figure 15). Each test case can be implemented by many test procedures, corresponding to different input values combinations: e.g., test case `Radio_TP0` has four test procedures.

3 Test Cases					
Test Name	Use Cases	Test Steps	Procedure	Variables	Properties
Radio_TP0	Radio	1) Set Band 2) Set AM Station 3) Verify that station is playing	0	rb = AM ams = NewChoice	uniqueid:113661
			1	rb = FM ams = FM	
			2	rb = FM ams = a	
			3	rb = FM ams = b	
			4	rb = FM ams = AM	
Radio_TP1	Radio	1) Set Band 2) Set FM Station 3) Verify that station is playing	0	rb = FM fms = NewChoice	uniqueid:113737
			1	rb = AM fms = NewChoice	
			2	rb = FM fms =	
			3	rb = FM fms = AM	
			4	rb = FM fms = FM	
Radio_TP2	Radio	1) Set Band 2) display invalid radioBand	0	rb = NewChoice	uniqueid:41320

Figure 15: Example of a test suite.

3.3 Metamodel

We derive a metamodel from TDE/UML Plug-in Development Manual [20] to present TDE/UML modeling and testing activities in Figure 16. In addition, an extension to

support TDE/UML regression testing functionality is added to the TDE/UML model in Figure 17. This metamodel and extension will be used in the next Sections to facilitate the definition of (1) traceability links between model elements and test procedures, (2) changes to TDE/UML models and (3) the impact of those changes on test procedures (see Sections 4, 5 and 6, respectively). Note that part of our understanding, further described below, is not captured directly in the metamodel but in constraints on the metamodel that are described later in this Section using the Object Constraint Language (OCL) [21].

Diagram and DiagramElement). As class UmlPackage is a subclass of DiagramElement, a UML package can be contained by another UML package and presented in a diagram.

A TDE/UML meta-model is constructed based on three abstract classes: Diagram, DiagramElement and DiagramRelationship. A diagram contains a set of diagram elements. A diagram element can be further described by another diagram (association rolenames childDiagram and parentElement between DiagramElement and Diagram): for instance, a use case is further described by a diagram that is an activity diagram, an activity can further describe an activity diagram (if the activity is stereotyped <<Nested>>). The relationships between two diagram elements are expressed by class DiagramRelationship. A diagram element has zero or more incoming and outgoing relationships (association rolenames incoming and outgoing between DiagramElement and DiagramRelationship). Each diagram relationship has a source diagram element and a target diagram element (association rolenames from and to between DiagramElement and DiagramRelationship).

Three kinds of diagrams are used in TDE/UML: use case diagrams (class UseCaseDiagram, subclass of Diagram), activity diagrams (class ActivityDiagram, subclass of Diagram) and class diagrams (class ClassDiagram, subclass of Diagram).

Use cases (class UseCase, subclass of DiagramElement) are presented in use case diagrams. The relationships between use cases could be includes and extends relationships (class IncludesRelationship and ExtendsRelationships, both of them are subclasses of DependencyRelationship, which is subclass of DiagramRelationship).

Various kinds of nodes can appear in an activity diagram. Each kind of nodes is modeled as a subclass of `DiagramElement`: decision nodes (class `Decision`), start nodes (class `ActivityStartState`), stop nodes (class `ActivityStopState`), fork/join nodes (class `SyncItem`) and activities (class `Activity`). Final Stop nodes are represented by class `ActivityFinalStopState`, which is a subclass of class `ActivityStopState`. Two nodes can be linked by a transition (class `ActivityEdge`, subclass of `DiagramRelationship`). Class `SyncItem` has an attribute `type`, of Enumeration type `ItemType`. It is used to denote whether a `SyncItem` is a fork or join node. If an activity is stereotyped by `<<Use Case>>`, then it is associated with a use case (association refer between `Activity` and `UseCase`).

Classes (class `UmlClass`, subclass of `DiagramElement`) are defined in a class diagram. Classes can be related with inheritance relationships (class `Inheritance`, subclass of `DiagramRelationship`) as well as association relationship (class `Association`, subclass of `DiagramRelationship`). Class `UmlClass` is inherited by class `Category`, which represents categories. Each category has a list of possible choices (class `Choice`). Every choice has a set of values (class `AttributeValue`) that are associated with the attributes of its category (association rolename related between class `AttributeValue` and class `Attribute`).

In addition, notes⁶ (class `Note`, subclass of `DiagramElement`) can be used in activity diagrams or class diagram. If a note stereotyped by `<<TDE/UML>>` appears in an activity

⁶ Notes here only refer to notes with stereotype `<<TDE/UML>>` because only these notes are used during the test generation.

diagram, it has to be attached to a node (class `NoteConnection`, subclass of `DiagramRelationship`).

When generating test suites, a use case would be chosen by testers as the starting use case (association rolename `startingUseCase` between class `TdeTestSuite` and class `UseCase`). A test suite (class `TdeTestSuite`) consists of a set of test cases (class `TdeTestCase`). A test case can be implemented by a set of test procedures (class `TdeTestProcedure`). Each test procedure contains a set of test steps (class `TdeTestStep`), each of which is associated with an activity in the activity diagram (association rolename `corresponding` between `TdeTestStep` and `Activity`). Every test step has zero or more variable and choice bindings (class `VariableChoice`). For each variable-choice binding, there is a corresponding variable (class `Variable` and association rolename `correspondVar` between `Variable` and `VariableChoice`) and a corresponding choice (association rolename `correspondChoice` between `Choice` and `Variable`). The variable involved in a variable-choice binding is defined in a note in an activity diagram⁷. Every variable has a type, which is a category defined in the class diagram (association rolename `type` between `Category` and `Variable`). Each procedure refers to the use cases it triggers (rolename `involves`): specifically the starting use case, as well as the use cases that are referred to by activities.

⁷ Based on the TDE/UML java documentation, there is no direct relationship between class `Note` and class `Variable`. As a result, we add an extension to link them in Figure 17.

The constraints specified in the model (i.e. guard condition on a transition, constraints in note⁸, global constraint⁸) are parsed as a set of classes that implement the interface `Constraint`, such as `class ConstraintAnd`, `class ConstraintOr`, `class ConstraintEquals`, `class ConstraintNotEquals`, `class ConstraintImplies`, `class ConstraintNot`, `class ConstraintForAll`, `class ConstraintExists` and `class ConstraintXor`. The instance of these constraints classes can contain each other except the instance of `ConstraintEquals` and `ConstraintNotEquals`. An instance of `ConstraintEquals` or `ConstraintNotEquals` could be further parsed to two expressions (Interface `Expression`). There are three kinds of expressions: variable expression (`class VariableExpression`), constant expression (`class ConstantExpression`) and choice expression (`class ChoiceExpression`). Each variable expression is associated with a variable (association rolename represent between class `VariableExpression` and class `Variable`). Each choice expression is associated with a choice (association rolename represent between class `ChoiceExpression` and class `Choice`).

⁸ These constraints are specified in notes stereotyped by <<TDE/UML>> in the model. But there is not direct connection between constraints and notes based on the TDE/UML java doc. We add an extension to link them in Figure 17.

Figure 17: metamodel for TDE/UML (part 2)

As shown in Figure 17, we add the extension to the TDE/UML meta-model to support our regression testing functionality. Notes specified in activity diagrams (class *NoteInActivityDiagram*, subclass of class *Note*) are used to (1) declare input variables

(class `VarDeclaration`, associated with class `Variable`), (2) specify constraints for those variables (class `NoteConstraint`, associated with Interface `Constraint`), and (3) specify TDE properties (class `TDEProperty`). Another kind of constraints is guard conditions on transitions (class `Guard`, associated with `ActivityEdge`), thus the association between `Guard` and Interface `Constraint`. Notes specified in class diagrams (class `NoteInClassDiagram`, subclass of `Note`) are used to make constraint on variables defined in all activity diagrams (class `GlobalConstraint`, associated with class `NoteInClassDiagram`), thus the association between `GlobalConstraint` and Interface `Constraint`.

To derive data flow information (see Section 7.2.1.2.2) from a TDE/UML model, class `Definition` and class `Use` are added. A `Definition` is associated with a `VarDeclaration` (association rolename related between class `VarDeclaration` and class `Definition`). A `Use` is associated with a `VariableExpression` (association rolename related between class `VariableExpression` and class `Use`). Class `Use` contains an attribute `isRetestable`, which is used to formally define impacted data flow entities in section 7.2.1.2.2.

Class `Change` is defined to monitor changes of a diagram element or a diagram relationship in two versions of a TDE/UML model (association between `Change` and `DiagramElement` and association between `Change` and `DiagramRelationship`). An instance of `Change` is responsible to store information about changes of every diagram element or diagram relationship, including: 1) attribute `version` (to identify elements in different version of a TDE/UML model); 2) attribute `change`, of type `ChangedType` (either `added`, `deleted`, `changed`, or `not changed`—`noChanged`); 3) attribute `isRestable`

(to be used in section 7.2.1.2.1 to define impacted control flow entities). Moreover, as different kinds of model elements have different information to store, class `Change` is extended by several subclasses to store information for different model elements, including class `ActivityEdgeChanged`, `NoteInActivityDChange`, `NodeChange`, `ActivityChange`. Class `ActivityEdgeChange` has two attributes `isSTChanged` and `isGChanged`, which are used to indicate whether the source/target node or guard condition of an activity edge (transition) is changed, respectively. Class `NoteInActivityDChange` contains three attributes: `isConChanged`, `isVNChanged` and `isVTChanged`. The attribute `isConChanged` denotes whether the `NoteConstraint` in a note is changed. The attribute `isVNChanged` denotes whether the set of names of variables are changed. The attribute `isVTChanged` denotes whether the types of variable are changed. Class `NodeChange` has two attributes `isIncomingChanged` and `isOutgoingChanged`, which are used to indicate whether the incoming or outgoing transitions of a node are changed, respectively. Class `ActivityChange` extends class `NodeChange`. It has an attribute `isNameChange`, which is used to indicate whether the name of an activity is changed.

Class `TraceabilityLink` is used to record traceability links between test procedures and model elements. Then the test procedures that have traceability links to an element with unique `id` are defined as the related test procedure of that element (association rolename related between `TestProcedure` and `DiagramElement` and association rolename related between `TestProcedure` and `DiagramRelationship`).

Class `Selection` with an attribute `type`, of type `SelectionType` (either could be `obsolete`, `retestable` or `reusable`), is added to store the result of test procedure selection.

Class `Path` and class `Helper` are added to facilitate formalizing our methodology. A path of a test procedure is associated with a sequence of nodes (association `rolename nodesInPath` between class `Path` and class `DiagramElement`) and a sequence of transitions (association `rolename edgesInPath` between class `Path` and class `ActivityEdge`). The formalization of methods in class `Helper` is provided in Appendix A (Section A.1).

Appendix A (Section A.2) also provides constraints (in OCL) on the metamodel (e.g., a use case diagram only contains use cases and their relationships).

4 TRACEABILITY BETWEEN ELEMENTS AND TEST PROCEDURES

Each test procedure is a specialization of a test case and is bound with specific test input values. Compared to test procedures, test cases are more abstract specifications that don't include input values. As a result we want traceability information for test procedures instead of test cases. In this Section, we will use the metamodel discussed in Section 3.3 to define the traceability links between TDE/UML model elements (e.g. nodes, transitions, notes, categories, choices and global constraints) and test procedures. More specifically, we establish traceability links between test procedures and model elements which have a unique identifier, i.e., subclasses of `IdElement` in Figure 17. Using traceability link, we can trace model elements to test procedures in a test suite and conversely. The goal of such traceability links is to support regression test selection and prioritization based on model changes. In this thesis, we represent the traceability links by a matrix called a traceability matrix.

Figure 18: An example of activity diagram

In Figure 18, we label each node to facilitate the following discussion. Each transition can be identified by a source node and a target node, e.g. transition (b, c) in the above diagram. Suppose that a test case generator derives three test procedures from Figure 18, as reported in Table 2.

Test procedures	Path
t1	a, b, c, d, f, g
t2	a, b, c, e, f, g
t3	a, b, c, h, i

Table 2: Test Suite for activity diagram in Figure 4

A traceability matrix can be built based on Table 2 to relate test procedures to nodes and transitions in the activity diagram, as illustrated in Table 3.

Node	Related Test procedures	Transition	Related Test procedures
a	t1, t2, t3	<a, b>	t1, t2, t3
b	t1, t2, t3	<b, c>	t1, t2, t3
c	t1, t2, t3	<c, d>	t1
d	t1	<c, e>	t2
e	t2	<d, f>	t1
f	t1, t2	<e, f>	t2
g	t1, t2	<f, g>	t1, t2
h	t3	<c, h>	t3
i	t3	<h, i>	t3

Table 3 : An example of traceability matrix

We define the traceability links for model elements (recall that operations of class `Helper` are defined in Appendix A.1). Note that only the definition of traceability links for use cases and test procedures are provided as example. The definition of traceability links for other model elements can be found in Appendix A.4.

- Traceability between use cases and test procedures that involve them:

```

context TdeTestProcedure
inv: self.involves
      ->forall(u: UseCase|self.traceabilitylink
        ->exists(t :TraceabilityLink|
          t.tdeTestprocedure=self
          and t.diagramElement=u) )

```

5 CHANGES ON MODEL ELEMENTS

In this Section we discuss the design changes that should be considered for regression test selection and the algorithm to detect those changes in a TDE/UML model. Since use case diagrams, activity diagrams and class diagrams are used in TDE/UML, we consider changes on each of these diagrams in turn. We make the assumption that diagrams are consistent. For example, if a choice of a category is deleted, then any constraint that involves the deleted choice is updated.

The definitions in this Section use the OCL syntax and are based on the metamodel⁹ (see Section 3.3). The main motivation is to have clear and precise specifications of regression testing strategy in such a way that it can be easily reviewed and readily implemented. Each time, we assume we are comparing version 1 and version 2, i.e., we are comparing the attribute version of class Change equal to 1 (version 1) and 2 (in version 2), respectively. The identification of modification to model elements will therefore result in elements in version 1 being classified as deleted, changed or not changed, and elements in version 2 as being deleted or not. (Indeed, to classify elements in version 2 as being deleted or changed, we would need a third version to compare them with.)

Note that in this chapter, when an OCL expression is similar to the ones we present below for added/deleted/changed/not changed use cases, the OCL expression can be found in Appendix A, Section A.4.

⁹ Because TDE/UML is an ongoing project, some concepts in the metamodel (e.g. how to use the stereotype for nodes, transitions, categories, choices) are not yet completely specified. These concepts will be clarified when the project is finished and will be considered in our regression technique as future work.

5.1 Changes in Use Case Diagrams

5.1.1 Changes to Use Cases

In TDE/UML, each use case is uniquely identified by its id¹⁰ in a use case diagram. Given two versions of a use case diagram, we can detect the set of added, deleted, and changed use cases. So we can have the following definitions:

Change Rule 1—Deleted use case: If the id of a use case in version 1 does not appear in version 2, then this use case is identified as a deleted use case.

```
context UseCase
inv: if self.change.version = 1 then
    UseCase.allInstance->select(change.version=2
                                and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted)
endif
```

Change Rule 2—Added use case: If the id of a use case in version 2 does not appear in version 1, then that use case is identified as an added use case.

```
context UseCase
inv: if self.change.version = 2 then
    UseCase.allInstance->select(change.version=1
                                and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added)
endif
```

Change Rule 3—Changed use case: if a changed use case that has the same id appears in two versions of use case diagram, but shows different name.

```
context UseCase
inv: if self.change.version = 1 then
    UseCase.allInstance->select(change.version=2
                                and id=self.id
                                and name<>self.name)
    ->notEmpty()
    implies self.change.type=ChangeType.changed)
endif
```

Change Rule 4—Unchanged use case: An unchanged use case exists in both versions of the use case diagram with the same id and name.

```
context UseCase
inv: if self.change.version = 1 then
    UseCase.allInstance->select(change.version=2
                                and id=self.id and name=self.name)
    ->notEmpty()
```

¹⁰ Note that the id of a model element is maintained by TDE/UML and can not be modified by users.

```
        implies self.change.type=ChangeType.notChanged)
    endif
```

5.1.2 Changes to Use Case Relationships

In a use case diagram, we could specify relationships between use cases, which could be extends or include relationships. The changes that could be made to these relationships are deletion and addition.

Change Rule 5—Deleted use case relationship: If the `id` of a use case in version 1 does not appear in version 2, then that use case relationship is identified as a deleted.

Change Rule 6—Added use case relationship: If the `id` of a use case in version 2 does not appear in version 1, then that relationship is identified as an added relationship.

Change Rule 7—Changed use case relationship: A changed relationship has the same `id` in two versions of an activity diagram, but shows a different from/to end.

Change Rule 8—Unchanged use case relationship: An unchanged use case relationship exists in both versions of the use case diagram with the same `id` and same to/from use cases.

5.2 Changes in Activity Diagrams

5.2.1 Changes to Nodes

5.2.1.1 Deleted/Added Node

Five kinds of nodes can be specified in an activity diagram: start node, stop node, activity, decision node, fork/join node. Each of them can be deleted from or added to an activity diagram. For the sake of simplicity, we just give the formalization of deleted and added node in the context of activity as an example. Formalizing the addition and deletion of the other types of nodes is similar and therefore omitted here.

Change Rule 9—Deleted Activity: A deleted activity appears in the first activity diagram version, but not in the second version.

Change Rule 10—Added Activity: An added activity appears in the second activity diagram version, but not in the first version.

5.2.1.2 Changed Node

Except deletion and addition, testers could make some other changes on start nodes, stop nodes, activities, decision nodes and fork/join nodes.

Change Rule 11—Changed Start nodes: if a start node has the same id in both versions of the activity diagram, but has different outgoing transitions.

```
context ActivityStartNode
inv: if self.change.version = 1 then
    (self.outgoing.id <> ActivityStartNode.allInstance
    ->select(change.version=2 and id=self.id).outgoing.id
    implies self.change.type=ChangeType.changed
endif
```

Change Rule 12—Changed Stop nodes: if a stop node has the same id in both versions of the activity diagram, but has different incoming transitions.

```
context ActivityStopNode
inv: if self.change.version = 1 then
    (self.incoming.id <> ActivityStopNode.allInstance
    ->select(change.version=2 and id=self.id).incoming.id
    implies self.change.type=ChangeType.changed
endif
```

Change Rule 13—Changed Activity: A changed activity has the same id in both versions of the activity diagram, but shows different name or different incoming/outgoing transitions.

```
context Activity
inv: if self.change.version = 1 then
    Activity.allInstance->select(
        change.version=2 and id=self.id and name<>self.name)
    ->notEmpty()
    implies (self.change.type=ChangeType.changed
        and self.change.isNameChanged=true)
    and
    self.outgoing.id <> Activity.allInstance
    ->select(change.version=2 and id=self.id).outgoing.id
    implies (self.change.type=ChangeType.changed
        and self.change.isOutgoingChanged=true)
    and
    self.incoming.id <> Activity.allInstance
    ->select(change.version=2 and id=self.id).incoming.id
```

```

        implies (self.change.type=ChangeType.changed
                and self.change.isIncomingChanged=true)
    endif

```

Change Rule 14—Changed Decision: A changed decision has the same id in both versions of the activity diagram, but has different sets of incoming/outgoing branches.

For example, a branch can be deleted from/added to a decision node.

```

context Decision
inv:  if self.change.version = 1 then
      self.outgoing.id <> Decision.allInstance
      ->select(change.version=2 and id=self.id).outgoing.id
      implies (self.change.type=ChangeType.changed
              and self.change.isOutgoingChanged=true)
      and
      self.incoming.id <> Decision.allInstance
      ->select(change.version=2 and id=self.id).incoming.id
      implies (self.change.type=ChangeType.changed
              and self.change.isIncomingChanged=true)
    endif

```

Change Rule 15—Changed fork/join: A changed fork/join has the same id in both versions of the activity diagram, but has different sets of branches.

5.2.2 Changes to Transitions

Change Rule 16—Deleted transition: A deleted transition appears in the first activity diagram version, but not in the second version.

Change Rule 17—Added transition: An added transition appears in the second version activity diagram, but not in the first version.

Change Rule 18—Changed transition: A changed transition has the same id in two versions of an activity diagram, but shows a different source node, target node, guard condition. Because a guard condition is a textual constraint (attribute text defined in Constraint), simple string comparisons can be used to detect guard changes. Changes to other kinds of constraints (e.g., constraints in a note and global constraints) can be also detected by direct string comparisons. In order to facilitate the formalization of classification of test procedures, we use attributes `isSTChanged`, `isGChanged` to identify whether the source/target node, guard condition or name is changed.

```

context ActivityEdge
inv:  if self.change.version = 1 then
      ActivityEdge.allInstance->select(

```

```

        version=2 and id=self.id
        and (from<>self.from or to<>self.to)->notEmpty()
    implies (self.change.type=ChangeType.changed
            and self.change.isSTChanged=true)
    or
    ActivityEdge.allInstance->select(
        version=1 and id=self.id
        and guard<>self.guard)->notEmpty()
    implies (self.change.type=ChangeType.changed
            and self.isGChanged=true)
endif

```

5.2.3 Changes to Notes in the Activity Diagram

Notes in the activity diagram can contain variable declarations, note constraints and TDE Properties. Because the TDE properties specify the configuration options of the test generator, we don't consider the changes on the TDE properties. Thus we only analyze changes on the notes that contain variable declaration and note constraints. Because constraints in a note only use variables that have been defined within the note, we assume that there is at least one variable declaration in a note (when the note is not a TDEProperty).

Change Rule 19—Deleted NoteInActivityDiagram: A deleted note appears in the first activity diagram version, but not in the second version.

Change Rule 20—Added NoteInActivityDiagram: An added note appears in the second activity diagram version, but not in the first version.

Change Rule 21—Changed NoteInActivityDiagram: A changed note has the same id in two versions of an activity diagram, but shows different variable declarations or constraint. If a variable declaration is added or deleted, or a declared variable name is changed, we set attribute isVNChanged to true. If the type of a variable is changed, we set attribute isVTChanged to true. If the text of the constraint is changed, we set attribute isConChanged to true.

```

context NoteInActivityDiagram
inv: if self.change.version = 1 then
    NoteInActivityDiagram.allInstance->select(
        change.version=2 and id=self.id and con.text<>self.con.text)
    ->notEmpty()

```



```

implies (self.change=ChangeType.changed
          and self.isConChanged=true)
and
NoteInActivityDiagram.allInstance->select(
  version=1 and id=self.id and isVNChanged(self))
->notEmpty()
implies (self.change=ChangeType.changed
          and self.isVNChanged=true)
and
NoteInActivityDiagram.allInstance->select(
  version=1 and id=self.id and isVTChanged(self))
->notEmpty()
implies (self.change=ChangeType.changed
          and self.isVTChanged=true)
endif

```

5.2.4 Changes to Note Connections

Change Rule 22—Deleted note connection: A deleted note connection appears in the first activity diagram version, but not in the second version.

Change Rule 23—Added note connection: An added note connection appears in the second activity diagram version, but not in the first version

Change Rule 24—Changed note connection: A changed note has the same id in two versions of an activity diagram, but shows different “from” diagram element or “to” diagram element.

```

context NoteConnection
inv: if self.change.version = 1 then
  NoteConnection.allInstance->select(
    change.version=2 and id=self.id
    and (from<>self.from or to<>self.to))->notEmpty()
  implies self.change.type=ChangeType.changed
endif

```

5.3 Changes in Class Diagram

5.3.1 Changes to Associations

Change Rule 25—Deleted association: A deleted association appears in the first class diagram version, but not in the second version.

Change Rule 26—Added association: An added association appears in the second version class diagram, but not in the first version.

Change Rule 27—Changed association: A changed association has the same id in two versions of a class diagram, but with one or more differences regarding the following elements: name, stereotype, from end, to end, from multiplicity, to multiplicity, from role, to role, aggregation or navigability.

```

context Association
inv: if self.change.version = 1 then
    Association.allInstance->select(version=2 and id=self.id
    and (name<>a.name or stereotype<>a.stereotype
        or source<>a.source or target<>a.target
        or fromMultiplicity<>a.fromMultiplicity
        or toMultiplicity<>toMultitplicity
        or fromRole<>a.fromRole or toRole<>a2.toRole
        or isAggregation<>a.isAggregation
        or isBiDirectional<>a.isBiDirectional))
    ->notEmpty()
    implies self.change=ChangeType.changed
endif

```

5.3.2 Changes to Choices

Change Rule 28—Deleted choice: A deleted choice appears in the first class diagram version, but not in the second version or it appears in both versions with a different name. Because each choice has unique name in a category, so if the name of a choice is changed, we should also consider it as deleted choice.

Change Rule 29—Added choice¹¹: An added choice appears in the second class diagram version, but not in the first version.

Change Rule 30—Changed choice: A changed choice has the same id and name in two versions of a class diagram, but has different values of attributes.

```

context Choice
inv: if self.change.version = 1 then
    Choice.allInstance->select(change.version=2 and id=self.id
    and name=c.name
    and isAttValueChanged(c,c.attributeValue))->notEmpty()
    implies self.change.type=ChangeType.added)
endif

```

¹¹ Added choices and added categories are used in the the new version of a system model and have no impact on the classification of the existing test procedures.

5.3.3 Changes to Inheritances

Change Rule 31—Deleted Inheritance: A deleted inheritance appears in the first class diagram version, but not in the second version.

Change Rule 32—Added Inheritance: An added inheritance appears in the second class diagram version, but not in the first version

Change Rule 33—Changed Inheritance: A changed Inheritance has the same id in two versions of a class diagram, but shows different from end or to end.

```
context Inheritance
inv: if self.change.version = 1 then
    Inheritance.allInstance->select(
        change.version=2 and id=self.id
        and (from<>self.from or to<>self.to))->notEmpty()
    implies self.change.type=ChangeType.changed
endif
```

5.3.4 Changes¹² to Attributes

Change Rule 34—Deleted attribute: A deleted attribute appears in the first class diagram version, but not in the second version or it appears in both versions with a different name. Because each attribute has a unique name in a category, so if the name of an attribute is changed, we should also consider it as deleted attribute.

Change Rule 35—Added attributes: An added attribute appears in the second class diagram version, but not in the first version.

5.3.5 Changes to Categories

Change Rule 36—Deleted Category: A deleted category appears in the first class diagram version, but not in the second version, or it appears in both versions with a different name (since a category name is unique in a class diagram).

Change Rule 37—Added Category: An added category appears in the second class diagram version, but not in the first version.

¹² Note that there is no notion of changed attribute since the only change that can be made is a change of name and names uniquely identify attributes (so changing a name results in a deletion and an addition).

Change Rule 38—Changed Category: A changed category has the same id and name in two versions of a class diagram, but has a different generalization relationship (added, deleted, changed), different or changed association relationship (added, deleted, changed) or different attributes (added, deleted). Additionally, if a base class (Category class) is changed, its subclasses are considered changed too. (Operations `getInheritances()` and `getAssociations()`, and `isBaseCategoryChanged()` used in the OCL expression below are recursive. For instance, `getInheritances()` returns the ancestors of a category.)

```

context Category
inv: if self.change.version = 1 then
    Helper.getInheritances(self)
    ->exist (change.type=ChangeType.deleted
            or change.type=ChangeType.changed)
    implies self.change.type=ChangeType.changed
    and
    Helper.getInheritances(
        Category.allInstances->select (change.version=2
            and id=self.id and name=self.name))
    ->exist (change.type=ChangeType.added)
    implies self.change.type=changeType.changed
    and
    Helper.getAssociations(self)
    ->exist (change.type=ChangeType.deleted
            or change.type=ChangeType.changed)
    implies self.change.type=ChangeType.changed
    and
    Helper.getAssociations(
        Category.allInstances->select (change.version=2
            and id=self.id and name= self.name))
    ->exist (change.type=ChangeType.added)
    implies self.change.type=changeType.changed
    and
    Category.allInstances->select (change.version=2
        and id=self.id and name=self.name).attribute
    ->exist (change.type=ChangeType.added)
    implies self.change.type=changeType.changed
    and
    self.attribute->exist (change.type=ChangeType.deleted
        or change.type=ChangeType.changed)
    implies self.change.type=ChangeType.changed
    and
    isBaseCategoryChanged(self) = true
    implies self.change.type=ChangeType.changed
endif

```

5.3.6 Changes to Notes in the Class Diagram

Change Rule 39—Deleted NoteInClassDiagram: A deleted note in a class diagram, i.e., a deleted global constraint appears in the first class diagram version, but not in the second version.

Change Rule 40—Added `NoteInClassDiagram`: An added global constraint appears in the second class diagram version, but not in the first version.

Change Rule 41—Changed `NoteInClassDiagram`: A changed global constraint has the same `id` in two versions of a class diagram but shows different `text`.

6 CLASSIFICATION

As mentioned in Section 3.2, there are three artifacts generated for a system modeled by TDE/UML: test suites, test cases, test procedures. Each test suite is generated to test an initial use case (i.e., a use case triggered by an actor) that is chosen by the user. Each test suite contains a set of test cases that exercise different sequences of activities in activity diagrams. A test case consists of a set of test procedures, which differ in the values of input variables. Thus, only test procedures are bound with specific test inputs and executed to test the system under test. As a result, we try to classify test procedures instead of test cases or test suites. After identifying the changes to a model, we can use the traceability link between model elements (i.e. nodes, edges, notes, categories, etc) and test procedures to classify the test procedures. A test procedure can be classified in one of three categories: obsolete (section 6.1), retestable (section 6.2) and reusable (section 6.3). We will use the metamodel (see Section 3.3) and OCL expressions to formalize the classification of test procedures. Last we summarize how the change rules of section 5 are used in our classifications (section 6.4).

Before describing the criteria to classify test procedures, we need to make an assumption about the gap that may exist between changes of the TDE/UML model of a system under test and changes of its implementation. A TDE/UML model is a test model of a system under test, which is derived from user requirements. The implementation of the system under test also conforms to those user requirements. Thus any changes to the user requirements could cause changes of the TDE/UML model and the implementation. So it's reasonable to assume that any change to the TDE/UML model of a system implies

changes to the implementation of the system. Therefore, classifying an obsolete test procedure, as identified from a TDE/UML model is also obsolete for the corresponding implementation. Similarly, a reusable (resp. retestable) test procedure, as identified from a TDE/UML model is also reusable (resp. retestable) for the corresponding implementation. Note though that if changes are made to the implementation and those changes do not correspond to changes in the TDE/UML model (e.g., an algorithm has been changed), then our classification is not accurate (e.g., a reusable test procedure should in fact be retestable). This is an issue that is inherent to any model based regression test selection technique.

6.1 Obsolete

Test procedures specify the inputs and expected outputs of a system under test to test functionalities of a system. Thus, a test procedure is obsolete: 1) if it tests an invalid functionality, 2) if it specifies an invalid input/output relation. Moreover TDE/UML use constraints to limited the number of generated test procedures. Therefore if a test procedure does not satisfy constraints in the modified model, then it is obsolete. The next sub-sections will discuss these three reasons, respectively.

6.1.1 Invalid Functionality

Classification Rule 1—As a use case presents a functionality of a system, if the use case is deleted, then the functionality becomes invalid. Furthermore, the name of a use case is an identifier for the use case and captures the functionality that users want to achieve with this use case. If the name of the use case is changed, that means users want to achieve a different functionality than the original one. In other words, the original

functionality of a name-changed use case becomes invalid. Therefore the deletion of a use case or the change of a use case name causes the test procedures that verify the functionality of the use case to become obsolete. (We assume here that a change of name is not performed to fix a typographic mistake but rather suggests a change of functionality.)

```

context TdeTestProcedure
inv: self.element->select(
    (change.type=ChangeType.deleted
    or change.type=ChangeType.changed)
    and oclIsTypeOf(UseCase) )->notEmpty()
implies self.selection.type=selectionType.obsolete

```

6.1.2 Invalid Input/Output Relation

As shown in Figure 15, a test procedure specifies input/output relations for the system under test by: 1) A sequence of test steps, each of which could be either a user action or system response. These test steps are activities in an activity diagram; 2) A set of variables with values. These variables are defined in the notes attached to nodes in the activity diagram. The values assigned to the input variables are choices of a category, which are defined in a class diagram. Therefore, a test procedure specifies an invalid input/output relation if it contains an invalid sequence of test steps or uses an invalid set of input variables. In the next sub-sections, we discuss the changes that would cause an invalid sequence of test steps and an invalid set of input variables, respectively.

6.1.2.1 Invalid Sequence of Test Steps

Based on our understanding of TDE/UML, we identify the following changes of a TDE/UML model that would cause an invalid sequence of test steps:

Classification Rule 2—Change of the name of an activity. The name of an activity describes the content of a user action or an expected system output. Changing the name of an activity implies that the activity does not present the same system input or expected output as before. Thus, a test step associated with a name-changed activity becomes invalid, thus leading to an invalid sequence of test steps.

```

context TdeTestProcedure
inv:  self.element->select (change.type=ChangeType.changed
                                and oclIsTypeOf(Activity))
    ->select (isNameChanged=true) ->notEmpty()
    implies self.selection.type=selectionType.obsolete

```

Classification Rule 3—Adding/deleting a branch to/from a fork/join: An added branch to a fork/join or deleted from a fork/join changes the parallel execution of activities. Adding/deleting a branch to/from a fork/join means an outgoing transition is added/deleted to/from the fork and an incoming transition is added/deleted to/from a join. Hence, the test procedures that traverse the fork/join contain the invalid sequences of test steps.

```

context TdeTestProcedure
inv:  self.element->select (change.type=ChangeType.changed
                                and oclIsTypeOf(SyncItem))
    ->select (type=fork
            and (isOutgoingChanged=true
                or isIncomingChanged=true) )
    ->notEmpty()
    implies self.selection.type=selectionType.obsolete

```

Classification Rule 4—A node has a different next node: This clearly results in invalid test procedures. There are two exceptions: 1) the addition of one or more decision nodes between two unchanged nodes, as illustrated in Figure 19 (a); 2) the deletion of a decision node, as illustrated in Figure 19 (b). Indeed, deleting one or more decision nodes between two existing nodes simply remove a condition to be satisfied by the path: the path has one less condition to satisfy and is therefore valid. In the case of the addition of one or more decision nodes between two existing nodes, the path (i.e., the sequences of activities, and

therefore the sequence of inputs/outputs) is not changed, and the path is valid provided that the added conditions are satisfied by the path. Checking that conditions still hold is further discussed in section 6.1.3. Therefore, such modification leads to a valid procedure if for every pair of consecutive nodes in version 1 we have either (i) no change, or (ii) the second node in the pair is a deleted decision, or (iii) version 2 shows that there are only decision nodes between the nodes of the pair. This is what the OCL expression below specifies: lines 5-8, 10-11, and 13-17 specify those three situations, respectively. And it is only if those conditions do not apply for every pair of consecutive nodes in the procedure (lines 1-2) that the procedure is obsolete (line 19).

Figure 19: inserting/deleting decision nodes

```

context TdeTestProcedure
inv: not
    self.element->forAll(e1,e2 | getNextNode(e1)->includes(e2)
        and
        (
            getNextNode(DiagramElement.allInstances->select(
                change.version=2 and id=e1.id))
            ->includes(DiagramElement.allInstances->select(
                change.version=2 and id=e2.id))
        or
        (e2.change.type=ChangeType.deleted
            and e2.ocliIsTypeOf(Decision)=true )
        or
        Helper.isOnlyContainDecisions(
            DiagramElement.allInstances
                ->select(change.version=2 and id=e1.id),
            DiagramElement.allInstances
                ->select(change.version=2 and id=e2.id))
        )
    implies self.selection.type=selectionType.obsolete)

```

6.1.2.2 Invalid Step of Input Variables

There are five scenarios that make the inputs variables of a test procedure become invalid:

Classification Rule 5—Addition or deletion of input variables for a test procedure. Because the specification of a test procedure contains a list of input variables, if a variable is added to or deleted from a test procedure, this would cause the original set of variables to become invalid. Addition or deletion of inputs variables of a test procedure results from the following changes: 1) adding or deleting notes that contain variable declarations; 2) adding or deleting variable declarations in a note or changing declared names. Some of the obsolete test procedures are identified using the following definition.

```
context TdeTestProcedure
inv self.element->select(change.type=ChangeType.deleted
and
oclIsTypeOf(NoteInActivityDiagram))
->exist(n.vars.size()<>0)
implies self.selection.type=selectionType.obsolete
and
self.element->select(change.type=ChangeType.changed
and
oclIsTypeOf(NoteInActivityDiagram))
->exist(isVNChanged=true)
implies self.selection.type=selectionType.obsolete
and
self.element->select(oclIsTypeOf(Activity)
or oclIsTypeOf(Decision)
or oclIsTypeOf(ActivityStartState)
or oclIsTypeOf(SyncItem)
or oclIsTypeOf(ActivityStopState))
->exist(e|
DiagramElement.allInstances->select(
id=e.id and change.version=2
and
e.incoming->select(oclIsTypeOf(NoteConnection))
->exist(nc:NoteConnection|
nc.from.change.type=ChangeType.added
and nc.from.vars.size()<>0)
)
)
implies self.selection.type=selectionType.obsolete
```

Classification Rule 6—Changes on the type of a variable: Given a variable of a test procedure, the type of the variable is a category in a class diagram. The value of the

variable is a choice of the category that corresponds to the type of the variable in the variable declaration. If the type of a variable is changed, it would cause the value of the variable, and therefore the path, to be invalid except in the following three situations: (1) The new type of the variable is the sub-category of the original type; (2) The new type of the variable is the base category of the original type, provided that the value of the variable is a choice defined in the base category; (3) The new type of the variable has the same base category as the original type, provided that the value of the variable is a choice defined in that base category. Some of the obsolete test procedures are defined as follows¹³:

```

context TdeTestProcedure
inv self.element->select(change.type=ChangeType.changed
                        and oclIsTypeOf(NoteInActivityDiagram))
->select(isVTChanged=true)
->exist(note|
    Helper.getTypeChangedVarDeclaration(note)
->exist( vd: VarDeclaration|
    not Helper.isSubCategory(vd,
        Helper.getRelatedVarDeclarationInV2(vd))
    and
    not ( Helper.isBaseCategory(vd,
        Helper.getRelatedVarDeclarationInV2(vd))
        and
        Helper.isChoiceDefinedInBaseCategory(vd))
    and
    not ( Helper.hasSameBaseCategory(vd,
        Helper.getRelatedVarDeclarationInV2(vd))
        and
        Helper.isChoiceDefinedInBaseCategory(vd))
    )
)
implies self.selection.type=selectionType.obsolete

```

¹³ In the OCL expression below, operation `getTypeChangedVarDeclaration()` returns the set of changed variable declarations in a note, operation `isSubCategory()` returns true if the first argument is a subclass of the second argument, operation `getRelatedVarDeclarationInV2()` returns the variable declaration in version 2 that is the changed version of a variable declaration in version 1, operation `isBaseCategory()` returns true if the first argument is a parent category of the second argument, operation `isChoiceDefinedInBaseCategory()` returns true if a variable declaration assigns a choice from an ancestor category, operation `hasSameBaseCategory()` returns true if the two arguments have a common ancestor category.

Classification Rule 7—Changing the location where variables are defined: Input variables are defined in notes, which are attached to nodes in the activity diagram. If a note is linked to another node, then the variables defined in the note become inputs for another node, thus leading to test procedures that use these variables to be obsolete. Some of the obsolete test procedures are defined as follows:

```
context TdeTestProcedure
inv self.element->select(change.type=ChangeType.changed
and oclIsTypeOf(NoteConnection))
    ->exist(from.vars.size<>0)
implies self.selection.type=selectionType.obsolete
```

Classification Rule 8—Deletion of a choice. Deleting a choice or deleting a category (if a category is deleted, all choices of this category are also deleted) results in (at least) an invalid choice being used in a procedure. This can cause some test procedures to be obsolete.

```
Context TdeTestProcedure
inv self.element->select(change=ChangeType.deleted
    and (oclIsTypeOf(Choice) or oclIsTypeOf(Category)))
    ->notEmpty()
implies self.selection.type=selectionType.obsolete
```

Classification Rule 9—Deletion of an inheritance: As described in Section 3.1.3.1, if there is an inheritance relationship between two categories, an input variable of the subcategory can be assigned with a choice defined in the base category. Therefore, if an inheritance relationship is deleted, then the variables that are given a choice of the base category become invalid. Thus deletion of an inheritance could lead to its related test procedures to be obsolete. (Note that identifying a deleted inheritance relationship is sufficient since we have traceability links between test procedures and the categories they use, including the parent categories.)

```
Context TdeTestProcedure
inv self.element->select(change=ChangeType.deleted
    and oclIsTypeOf(Inheritance)) ->notEmpty()
implies self.selection.type=selectionType.obsolete
```

Classification Rule 10—Changes of the source end or target end of an inheritance: Similarly to the deletion of an inheritance, if such a change occurs then the variables that are assigned with a choice of the base category become invalid. Thus the obsolete test procedures that are caused by changes of the source end or target end of an inheritance are formalized as follows:

```
Context TdeTestProcedure
inv self.element->select (change=ChangeType.changed
                        and oclIsTypeOf(Inheritance))->notEmpty()
implies self.selection.type=selectionType.obsolete
```

6.1.3 Unsatisfied Constraints

After identifying test procedures that specify the invalid input/output relations, we need to check whether the remaining test procedures satisfy changed constraints and new constraints (i.e. added global constraints or guards on added transitions). If the test procedures do not satisfy those constraints, then they also need to be classified as obsolete.

Classification Rule 11—Changed constraints: If one of the constraints involved in a test procedure (e.g. guard conditions, constraints in a note and global constraints) is detected to be changed, then this test procedure must be verified to see whether it satisfies the new constraint. If not, the test procedure should be classified as obsolete¹⁴.

```
context TdeTestProcedure
inv self.element->select (change.type=ChangeType.changed
                        and oclIsTypeOf(ActivityEdge))
                        ->select (isGChanged=true)->exists (e|
                        not Helper.isTestProcedureSatisfied(self,
                        e.guard.con))
implies self.selection.type=SelectionType.obsolete
inv self.element->select (change.type=ChangeType.changed
                        and oclIsTypeOf(NoteInActivityDiagram))->exist (n|
                        not Helper.isTestProcedureSatisfied(self, n.nc.con))
```

¹⁴ Because TDE/UML can generate test cases that satisfy specified constraints (e.g. guards, global constraints), it is reasonable to expect this can be automated.

```

implies self.selection.type=selectionType.obsolete
inv self.element->select (change=ChangeType.changed
    and oclIsTypeOf(NoteInClassDiagram) )->exist (n|
    not (Helper.isTestProcedureSatisfied(self, n.gc.con)
implies self.selection.type=SelectionType.obsolete

```

Classification Rule 12—Added Global Constraint: If the categories in the added global constraint are included in the set of categories which are the types of the variables used in a test procedure, we have to check whether the test procedure satisfies this new constraint. If not, the test procedure is classified as obsolete.

```

context TdeTestProcedure
inv let notes:Set (NoteInClassDiagram)
    = NoteInClassDiagram.allInstance-
    >select (type.version=2
    and change=ChangeType.added)
in
    self.element->select (oclIsTypeOf(Category))
    ->exist (c|
        notes->exist (n|
            getCategoriesOfNoteInClassDiagram(n) ->includes (c)
            and
            not Helper.isTestProcedureSatisfied(self, n.gc.con)
        )
    )
implies self.selection.type=selectionType.obsolete)

```

Classification Rule 13—Guard condition on added transition: The addition of a guard can lead to (still) valid sequences of steps (i.e., test procedures) in two scenarios. First, when testers insert one or more decision nodes between consecutive nodes on a path tested by a test, provided that the added condition is satisfied. This was illustrated in Figure 19. Second, when testers delete a decision node on the path, the sequence of test steps on the path is still preserved, though a new transition is added to link the previous node and next node of the deleted decision node.

```

context TdeTestProcedure
inv getAddedTransitionOnthePath(self)->forAll (e: ActivityEdge |
    not Helper.isTestProcedureSatisfied(self, e)
implies self.selection.type=selectionType.obsolete

```

Classification Rule 14—Besides the situations discussed above, changed choices (Section 5.3.2) may lead to test procedures not being valid anymore because of unsatisfied

(changed or added) constraints If a choice is changed, we need to check whether its related test procedures satisfy the constraints that involve variables that take the changed choice as value. If not, those test procedures are classified as obsolete.

```

context TdeTestProcedure
inv self.element->select(change=ChangeType.changed
                        and oclIsTypeOf(Choice))->exists(c:Choice|
                        getConstraintsInvolvesVarsofChoice(c)
                        ->exists(con:Constraint|
                        not Helper.isTestProcedureSatisfied(self, con)))
implies self.selection.type=selectionType.obsolete

```

6.2 Retestable

After identifying obsolete test procedures, the remaining test procedures test valid functionalities, specify valid input/output relations and satisfy changed constraints and new constraints (i.e., new global constraints and guard conditions on added transition). Then we have to select a subset of those remaining test procedures to rerun. The purpose of rerunning the selected subset of remaining test procedures is to (hopefully) be able to detect faults in the system under test that are due to changes. As proposed in [22], the non-obsolete modification-traversing test cases, which are a superset of fault-detecting test cases, are selected to rerun. We adapt this idea to select the retestable test procedures for regression testing in TDE/UML.

Based on the assumption that any change of a TDE/UML model indicates that there is a corresponding modification in the implementation, if a non-obsolete test procedure is modification-traversing with respect to two versions of a TDE/UML model, then we expect it also traverses the modification parts of the implementation and therefore becomes retestable. This is however an assumption as some code changes may not be reflected as model changes (e.g., changing an algorithm implementation into a more efficient one).

Thus a retestable test procedure is a test procedure that still tests a valid functionality, specifies valid input/output relations and satisfies changed constraints and new constraints, but traverses model changes as described below. Note that to simplify the OCL expressions, we systematically first make sure that the test procedure under consideration is not obsolete.

Classification Rule 15—Deletion of decision nodes: If there is a decision node deleted from the path tested by a test procedure, the test procedure is still valid since a condition that used to be satisfied (the guard of the deleted decision) no longer exists. As such, the test procedure is non-obsolete modification-traversing, and thus becomes retestable.

```

Context TdeTestProcedure
Inv (not self.selection.type=selectionType.obsolete
      and
      self.element->exist(change.type=ChangeType.deleted
                          and oclIsTypeOf(Decision)))
      implies self.selection.type=selectionType.retestable

```

Classification Rule 16—Addition of decision nodes: The addition of a decision node to the path tested by a test procedure, so as to model an alternative path, wouldn't affect the sequence of test steps of the test procedure. Note that, as discussed in Classification Rule 13, we ensure that test procedures satisfy the guard conditions on the added transitions of the added decision nodes. Thus non-obsolete test procedures that traverse added decision nodes are still valid and thus considered to be non-obsolete modification traversing, which means that they become retestable.

```

context TdeTestProcedure
inv (not self.selection.type=selectionType.obsolete
      and
      self.element->forAll(e1,e2 | getNextNode(e1)->includes(e2)
                          and
                          Helper.isOnlyContainDecisions(
                              DiagramElement.allInstances
                              ->select(change.version=2 and id=e1.id),
                              DiagramElement.allInstances
                              ->select(change.version=2 and id=e2.id))))
      implies self.selection.type=selectionType.retestable

```

Classification Rule 17—Decision nodes with added/deleted outgoing transitions: If an outgoing transition of a decision node is added/deleted, and the decision node is in a test path but the added/deleted transition of that decision node is not, then a test procedure that exercises the test path is not affected and is still valid. In that case, the test procedure is non-obsolete modification-traversing, thus becomes retestable. (The OCL expression below checks that none of the edges taken by the path, available through `getActivityEdgesForProcedure()` is an added/deleted outgoing transition of modified decision nodes of the path, available through `getAddedOrDeletedOutgoingEdges()`.)

```

context TdeTestProcedure
inv    ( not (self.selection.type=selectionType.obsolete)
          and
          ( self.element->select (change.type=ChangeType.Changed
                                and oclIsTypeOf(oclType=Decision))
            ->select (isOutgoingChanged=true)
            ->forAll( d| getActivityEdgesForProcedure(self)
                    ->excludesAll(getAddedOrDeletedOutgoingEdges(d)) ) )
          ) implies self.selection.type=selectionType.retestable

```

Classification Rule 18—Changed categories: If a category that is used as a type of a variable of a test procedure (defined in Section 5.3.5) is changed, whatever the modification (recall Change Rule 38), the test procedure is non-obsolete because the variable typed by the changed category is valid and is still able to be used as the input for the test procedure. Thus the test procedure is non-obsolete modification-traversing and becomes retestable.

```

context TdeTestProcedure
inv    ( not self.selection.type=selectionType.obsolete
          and
          self.element->select (change.type=ChangeType.changed
                                and oclIsTypeOf(Category)) ->notEmpty() )
          implies self.selection.type=selectionType.retestable

```

Classification Rule 19—Changed choices: If the values of attributes of a choice are changed, the choice is identified as a changed choice (defined in Section 5.3.5). Because a changed choice can still be used as the values of test procedures' variables, test

procedures that use the changed choice are still valid. Thus they are non-obsolete modification-traversing, and retestable.

```
Context TdeTestProcedure
Inv (not self.selection.type=selectionType.obsolete
      and
      self.element->select(change.type=ChangeType.changed
                           and oclIsTypeOf(Choice))->notEmpty())
implies self.selection.type=selectionType.retestable
```

Classification Rule 20—Change of the type of a variable: As described in Section 6.1.2.2, the changes to the type of a variable could lead to test procedures that use this variable to be obsolete with the exception of three scenarios. In other words, if the change to the type of a variable falls into these exceptions, then the value of the variable is still valid. The test procedure is non-obsolete modification-traversing, and therefore retestable.

```
context TdeTestProcedure
inv (not self.selection.type=selectionType.obsolete
      and
      self.element->select(change.type=ChangeType.changed
                           and oclIsTypeOf(NoteInActivityDiagram))
      ->exist(isVTChanged=true))
implies self.selection.type=selectionType.retestable))
```

Classification Rule 21—Changed constraints: If one of the constraints involved in a test procedure (i.e. guard conditions, constraints in a note and global constraints) is detected to have been changed and this test procedure satisfies the new constraints, then the test procedure is still valid. Thus it would be considered to be non-obsolete modification-traversing and becomes retestable. Please note that detecting the changes and ensuring that the test procedure is not obsolete ensures that the condition is satisfied (Classification Rule 11).

```
context TdeTestProcedure
inv (not self.selection.type=selectionType.obsolete
      and
      self.rel->select(change.type=ChangeType.changed
                      and oclIsTypeOf(ActivityEdge))
      ->exist(isGChanged=true)
      or
      self.rel->select(change.type=ChangeType.changed
                      and oclIsTypeOf(NoteInActivityDiagram))
      ->exist(isConChanged=true))
```

```

or
self.element->select(change=ChangeType.changed
                    and oclIsTypeOf(NoteInClassDiagram))
                    ->notEmpty()
implies self.selection.type=selectionType.retestable)

```

Classification Rule 22—Added global constraints: If the categories in an added global constraint are included in the set of categories (which are the types of the variables used in a test procedure), and the test procedure satisfies the added global constraint, then the test procedure is also non-obsolete modification-traversing, thus becomes retestable. Again, by selecting a non-obsolete test procedure, if the procedure has an added global constraint, we know the constraint is satisfied (Classification Rule 12).

```

context TdeTestProcedure
inv let notes:Set(NoteInClassDiagram)
    = NoteInClassDiagram.allInstance-
    >select(type.version=2
          and change=ChangeType.added)
in
(not self.selection.type=selectionType.obsolete
 and self.element->select(oclIsTypeOf(Category))>exist(c|
 notes->exist(n| getCategoriesOfNoteInClassDiagram(n)
->includes(c))
)
implies self.selection.type=selectionType.retestable)

```

6.3 Reusable

Classification Rule 23—If no change is made to the model elements linked to a test procedure (i.e. nodes, transitions, categories, etc.), the test procedure is considered to be reusable.

```

context TdeTestProcedure
inv (not self.selection.type=selectionType.obsolete
 and
 not self.selection.type=selectionType.retestable)
implies self.selection.type=selectionType.reusable

```

6.4 Summary

Table 4 summarizes how changes that have been identified in Section 5 are used for test procedure classification. The table shows that some changes are not currently used. This includes added use cases, added/deleted/changed dependency relationships in use case

diagrams, deleted/added note connections, added/deleted start nodes, added stop nodes in activity diagrams, added categories, added choices, and deleted global constraints in class diagrams. The reasons are provided below:

- 1) Added use cases: Because an added use case requires a new test suite to test the functionalities that it represents, it doesn't affect the original test suites. Thus, we don't analyze the impact of addition of use cases. There is one exception though: when an added use case is an included used case or an extension use case. This would also lead to the addition of activity nodes, which we accounted for.
- 2) Added/deleted/Changed dependency relationships in a use case diagram: Any of these changes are reflected by changes in activity diagram(s), such as added/deleted/changed activity nodes, which we already account for. Therefore, we don't need to analyze the impact of changes to dependency relationships in a use case diagram.
- 3) Added/deleted note connections: The addition or deletion of a note connection is always triggered by the addition/deletion of a note in an activity diagram, which we already considered.
- 4) Added/deleted start nodes: Every activity diagram only needs a start node. A start node is added or deleted when users want to create or delete an activity diagram. Therefore, the addition or deletion of a start node happens with other changes in activity diagrams, such as added/deleted activities, which we already accounted for.

- 5) Added stop nodes: A new stop node is either added to a new path or an existing path in activity diagrams. If it is added to a new path, there is no impact on existing test procedures. If it is added to an existing path, it has impacts, which we already considered.
- 6) Added Categories: An added category in version 2 can only affect a test procedure of version 1 if it is used in a changed constraint, which we accounted for.
- 7) Added Choices: An added choice in version 2 can only affect a test procedure of version 1 if it is used in a changed constraint, which we accounted for.
- 8) Deleted GlobalConstraints: If a global constraint is deleted, that means we put less constraint on the variables involved in a test procedure. So the deletion of global constraints has no impact on existing test procedures.

Modification¹⁵	Obsolete	Retestable
Deleted use case (Change Rule 1)	Classification Rule 1	
Added use case (Change Rule 2)		
Changed use case (Change Rule 3)	Classification Rule 1	
Added use case relationship (Change Rule 6)		
Deleted use case relationship (Change Rule 5)		
Changed use case relationship (Change Rule 7)		
Deleted activity (Change Rule 9)	Classification Rule 4	
Deleted decision (Change Rule 9) ¹⁶		Classification Rule 15
Deleted fork/join (Change Rule 9) ¹⁶	Classification Rule 4	
Deleted start node (Change Rule 9)		
Deleted stop node (Change Rule 9)	Classification Rule 4	
Added activity (Change Rule 10)	Classification Rule 4	
Added decision (Change Rule 10) ¹⁶		Classification Rule 16
Added fork/join (Change Rule 10) ¹⁶	Classification Rule 4	
Added start node (Change Rule 10)		
Added stop node (Change Rule 10)		
Changed activity (Change Rule 13)	Classification Rule 2	
Changed decision (Change Rule 14)		Classification Rule 17
Changed fork/join (Change Rule 15)	Classification Rule 3	
Changed start node (Change Rule 11)	Classification Rule 4	Indirectly accounted by Classification Rule 15, Classification Rule 16
Changed stop node (Change Rule 12)	Classification Rule 4	Indirectly accounted by Classification Rule 15, Classification Rule 16
Deleted transition (Change Rule 16)	Indirectly accounted by Classification Rule 4	Indirectly accounted by Classification Rule 15, Classification Rule 16
Added transition (Change Rule 17)	Classification Rule 13, Indirectly accounted Classification Rule 4	Indirectly accounted by Classification Rule 15, Classification Rule 16
Changed transition (Change Rule 18)	Classification Rule 11, Indirectly accounted Classification Rule 4	Classification Rule 21, Indirectly accounted by Classification Rule 15, Classification Rule 16
Deleted Note (Change Rule 19)	Classification Rule 5	
Added Note (Change Rule 20)	Classification Rule 5	
Changed Note (Change Rule 21)	Classification Rule 5, Classification Rule 6, Classification Rule 11	Classification Rule 20, Classification Rule 21
Deleted Note Connection (Change Rule 22)	Indirectly accounted by Classification Rule 5	
Added Note Connection (Change Rule 23)	Indirectly accounted by Classification Rule 5	
Changed Note Connection (Change Rule 24)	Classification Rule 7	

¹⁵ Change Rule 4 and Change Rule 8 do not appear since they are examples of rules describing no change.

¹⁶ The rule would be similar to Change Rule 9 (deletion) or Change Rule 10 (addition) but has been omitted in section 5.

Deleted Association (Change Rule 25)		Indirectly accounted by Classification Rule 18
Added Association (Change Rule 26)		Indirectly accounted by Classification Rule 18
Changed Association (Change Rule 27)		Indirectly accounted by Classification Rule 18
Deleted Inheritance (Change Rule 31)	Classification Rule 9	Indirectly accounted by Classification Rule 18
Added Inheritance (Change Rule 32)		Indirectly accounted by Classification Rule 18
Changed Inheritance (Change Rule 33)	Classification Rule 10	Indirectly accounted by Classification Rule 18
Deleted choice (Change Rule 28)	Classification Rule 8	
Added choice (Change Rule 29)		
Changed choice (Change Rule 30)	Classification Rule 14	Classification Rule 19
Deleted Attribute (Change Rule 34)		Indirectly accounted by Classification Rule 18
Added Attribute (Change Rule 35)		Indirectly accounted by Classification Rule 18
Deleted category (Change Rule 36)	Classification Rule 8	
Added category (Change Rule 37)		
Changed category (Change Rule 38)		Classification Rule 18
Deleted GlobalConstraint (Change Rule 39)		
Added GlobalConstraint (Change Rule 40)	Classification Rule 12	Classification Rule 22
Changed GlobalConstraint (Change Rule 41)	Classification Rule 11	Classification Rule 21

Table 4: Relations between change rules and classification rules

Table 5 summarizes the model element modifications (addition, deletion, change) we account for, indicating which change rule is involved, and how those modifications are used for classification. For each model element, the first row indicates the rules for detecting deletions, additions and changes of that model element, and the second row indicates how each modification results in some classification (classification rule). A N/A indicates that the modification of the model element is not used for classification purposes.

	Deleted	Added	Changed
Use cases	Change Rule 1	Change Rule 2	Change Rule 3
	Classification Rule 1	N/A	Classification Rule 1
Use case relationships	Change Rule 5	Change Rule 6	Change Rule 7
	N/A	N/A	N/A
Start node	Change Rule 9	Change Rule 10	Change Rule 11
	N/A	N/A	Classification Rule 4, indirectly accounted by Classification Rule 15, Classification Rule 16
Stop node	Change Rule 9	Change Rule 10	Change Rule 12
	Classification Rule 4	N/A	Classification Rule 4, indirectly accounted by Classification Rule 15, Classification Rule 16
Activities	Change Rule 9	Change Rule 10	Change Rule 13
	Classification Rule 4	Classification Rule 4	Classification Rule 2
Decisions	Change Rule 9	Change Rule 10	Change Rule 14
	Classification Rule 15	Classification Rule 16	Classification Rule 17
Forks	Change Rule 9	Change Rule 10	Change Rule 15
	Classification Rule 4	Classification Rule 4	Classification Rule 3
Joins	Change Rule 9	Change Rule 10	Change Rule 15
	Classification Rule 4	Classification Rule 4	Classification Rule 3
Transitions	Change Rule 16	Change Rule 17	Change Rule 18
	Indirectly accounted by Classification Rule 4, Classification Rule 15, Classification Rule 16	Classification Rule 13 Indirectly accounted by Classification Rule 4, Classification Rule 15, Classification Rule 16	Classification Rule 11, Classification Rule 21, indirectly accounted Classification Rule 4, Classification Rule 15, Classification Rule 16
Notes in activity diagrams	Change Rule 19	Change Rule 20	Change Rule 21
	Classification Rule 5	Classification Rule 5	Classification Rule 5, Classification Rule 6, Classification Rule 11, Classification Rule 20, Classification Rule 21
Note connections	Change Rule 22	Change Rule 22	Change Rule 23
	N/A	N/A	Classification Rule 7
Associations	Change Rule 25	Change Rule 26	Change Rule 27
	Indirectly accounted by Classification Rule 18	Indirectly accounted by Classification Rule 18	Indirectly accounted by Classification Rule 18
Choices	Change Rule 28	Change Rule 29	Change Rule 30
	Classification Rule 8	N/A	Classification Rule 14, Classification Rule 19
Inheritances	Change Rule 31	Change Rule 32	Change Rule 33
	Classification Rule 9, Indirectly accounted by Classification Rule 18	Indirectly accounted by Classification Rule 18	Classification Rule 10, Indirectly accounted by Classification Rule 18
Attributes	Change Rule 34	Change Rule 35	N/A (see footnote 12)
	Indirectly accounted by Classification Rule 18	Indirectly accounted by Classification Rule 18	
Categories	Change Rule 36	Change Rule 37	Change Rule 38
	Classification Rule 8	N/A	Classification Rule 18
Notes in class diagrams	Change Rule 39	Change Rule 40	Change Rule 41
	N/A	Classification Rule 12, Classification Rule 22	Classification Rule 11, Classification Rule 21,

Table 5: Deletion/addition/change of model elements we account for

7 PRIORITIZATION TECHNIQUE IN TDE/UML

In this Section, we suggest a prioritization technique for regression test in the context of TDE/UML. During regression test selection, changes to a TDE/UML model are used to classify test procedures into three categories: obsolete, retestable and reusable. Based on the impact analysis, some modifications (e.g. changed notes, changed choices or changed categories) cause their *related test procedures* to be retestable. We use the term *retestable changed elements* to refer to the set of changed elements that cause their related test procedures to be retestable. We will focus on prioritizing these retestable test procedures.

Before discussing the prioritization strategy, let us review some concepts about test case generation in TDE/UML. An initial use case is selected to generate a test suite during test case generation. Each test suite contains a set of test cases that exercise different sequences of activities in activity diagrams. A test case consists of a set of test procedures which differ in the values of input variables. TDE/UML therefore generates three kinds of artifacts during test case generation: test suites, test cases and test procedures.

We propose a prioritization technique for a system under test modeled by TDE/UML that handles these three kinds of artifacts: test suites prioritization, test cases prioritization and test procedures prioritization. The prioritization technique begins with test suites prioritization (Section 7.1), which is followed by test case prioritization for each test suite (Section 7.2). Then test procedures for each test case are prioritized (Section 7.3). Finally, we obtain an overall sequence for test procedures (Section 7.4).

7.1 Test Suites Prioritization

As a test suite in TDE/UML tests the functionality of one initial use case of a system (plus the use cases triggered by this initial use case), we decide to prioritize test suites by prioritizing the corresponding initial use cases. As use cases present the functional requirements of a system, we apply the Analytical Hierarchy Process (AHP) method [23], which has been used in requirement prioritization, to use case prioritization in TDE/UML.

In the next sub section, we discuss AHP for requirement prioritization (Section 7.1.1) and give a detailed description of how we apply AHP to prioritize use cases (i.e., test suites) for TDE/UML (Section 7.1.2).

7.1.1 Requirement Prioritization

Although there is little previous work on prioritizing use cases, numerous approaches for requirement prioritization have been investigated. These requirement prioritization techniques [24-26] are used to plan the implementation of requirements when time and resources are limited. One of these requirements prioritization methods is the Analytical Hierarchy Process (AHP) method [23]. AHP is a decision making method, which involves comparing all pairs of alternative solutions to decide which one is better according to certain criteria (e.g., cost, value or risk), and to what extent. Karlsson adapted the AHP method to prioritize software requirements by building a pair-wise comparison matrix to calculate the relative value and cost of requirements so that the relative priorities of requirements can be determined [24]. Karlsson et al. compared the AHP method to other alternative methods for requirement prioritization [10] and showed

that the AHP method can provide more reliable results. However, the AHP method might have scalability problems when it is applied in large-scale industry projects since the number of pair-wise comparisons, that have to be (manually) determined by the user, can be substantial when the number of requirements grows.

7.1.2 Use Case Prioritization with the AHP Method

As use cases present the functional requirements of a system, we adapt the AHP method to use cases prioritization. The procedure of using the AHP method to rank use cases contains three steps: 1) Determining the use cases that need to be ranked (Section 7.1.2.1); 2) Performing the pair-wise comparisons for those use cases (Section 7.1.2.2); 3) Estimating the priorities for those use cases (Section 7.1.2.3).

7.1.2.1 Determining the Use Cases to Prioritize

Numerous use cases are involved in an industry project (e.g., the total number of use cases for an industrial system provided by Siemens is 54). Performing pair-wise comparisons on all these use cases would require a large amount of effort. The AHP method does not scale up very well when a large number of requirements are ranked. It is only suitable for a project with less than 20 requirements [27]. However, in the context of regression test selection and prioritization for TDE/UML, we only need to rank the use cases that have test suites with retestable test procedures. It is our conjecture that there is a small number of such use cases when performing regression test selection and prioritization. Therefore decision maker only need to perform a reasonable number of pair-wise comparisons, which are suitable for the application with the AHP method.

Furthermore, since regression testing is an iterative process, the decision maker may only need to provide a reduced number of pair-wise comparisons at each iteration. We can build a comparison pool which contains comparisons between pairs of use cases that have been evaluated in previous iterations. In a subsequent iteration, after identifying the use cases that need to be evaluated, we first select the pair-wise comparisons in the comparison pool, and then ask decision makers to assess the pairs of use cases that do not exist in the comparison pool. Then the new comparisons are put into the comparison pool for yet other iterations.

7.1.2.2 Performing Pair-Wise Comparison of Candidate Use Cases

Pair-wise comparison requires that decision makers assign a value to each pair of use cases (i, j) to indicate the relative importance of a use case i with respect to use case j according to a given scale. The importance of a use case can be measured in terms of different dimensions, depending on the domain experts' choice. Some potential criteria which have been used to evaluate the importance of requirements or use cases in the literature [28, 29] are:

- Business value. Different use cases might have different business values to the customer.
- Risk of fault-proneness. A use case which is more fault-prone is considered more important and requires more testing effort.

- Cost¹⁷ to implement. A use case that has been implemented with a high cost (or is expected to require a high cost) is considered to be more important than others.
- Frequency of use. The functional requirement that a use case describes could have an expected frequency of use. A use case frequently used should be considered as more important than use cases less frequently used.

For the scale scheme, Saaty [23] suggests the following nine-point scale (see Table 6) for the AHP method.

Numerical value	Explanation
1	Two use cases have equal importance.
3	Use case i has a slightly higher importance value than use case j.
5	Use case i has a strongly higher importance value than use case j.
7	Use case i has a very strongly higher importance value than use case j.
9	Use case i has an absolute higher importance value than use case j.
Reciprocals	If use case i has one of the numerical values when it compares with use case j, then use case j has the reciprocal value when compared with i.

Table 6: Scale for pair-wise comparisons

For example, assume that there are four candidate use cases: UC1, UC2, UC3 and UC4. They are placed into the rows and columns of a matrix, as shown in Table 7. An entry in row i and column j of the matrix indicates the relative importance of use case i with respects to use case j: for instance, UC2 has a slightly higher importance value than UC1 (value 3); conversely, the entry (UC1, UC2) has the reciprocal value 1/3. Note that the diagonal entries in the pair-wise matrix are all “1” since the importance of a use case is always 1 when it is compared to itself.

	UC1	UC2	UC3	UC4
UC1	1	1/3	5	7

¹⁷ The implementation cost could be estimated by the total amount of developer-hours to implement a requirement. A developer-hour is the amount of work performed by an average developer in one hour.

UC2	3	1	7	9
UC3	1/5	1/7	1	1/3
UC4	1/7	1/9	3	1

Table 7: An example of pair-wise comparisons

7.1.2.3 Estimating Priorities for Use Cases

Based on the pair-wise comparison matrix, we can estimate the priorities for use cases using AHP. First the sum of each column in the matrix is computed. Then we divide the value of each entry in the matrix by the column sum, and compute the sum for each row.

Table 8 shows this on the example of Table 7.

	UC1	UC2	UC3	UC4	Sums	Priority
UC1	0.230	0.208	0.3125	0.404	1.1545	0.29
UC2	0.690	0.629	0.4375	0.519	2.2755	0.57
UC3	0.046	0.088	0.0625	0.019	0.2155	0.05
UC4	0.033	0.069	0.1875	0.058	0.3475	0.09

Table 8: Normalized comparison matrix

Then we can derive the priority vector, which is the eigenvalues of the matrix, by dividing each row sum by the number of use cases. In this example, the priority vector is (0.29, 0.57, 0.05, 0.09). Based on this priority vector, the order of use cases in the example is: UC2, UC1, UC4 and UC3. Finally, the order of test suites is determined by the order of use cases.

7.2 Test Case Prioritization

After test suites have been prioritized, we prioritize test cases in each test suite. Note that only test cases that contain *retestable test procedures* need to be prioritized. In TDE/UML, each test case exercises a path of the graph representing activity diagrams for use cases. To facilitate the description, we use the term *tested path* to refer to a path exercised by a test case.

Although the AHP method is used in test suite prioritization, it is not applicable to prioritize test cases in each test suite. The reason is that the AHP method does not scale up well. It can be used to prioritize a system that has less than 20 requirements for example. However, the number of test cases within each test suites could be very large, likely much larger than 20. Thus, the AHP method can not be used in test case prioritization. Instead, we propose an automated test case prioritization approach, which relies on a probability value and risk value that we derive for each *tested path*. The probability value of a *tested path* refers to the likelihood of exercising this *tested path* when the system under test is running: this probability will be derived from probability values already available in a TDE/UML model (recall Section 3.1.3.1), as further discussed in Section 7.1.2.3. The risk value of a *tested path* refers to its fault-proneness. We consider two criteria to measure the risk of a *tested path*. First, we can determine the coverage of control flow entities (e.g. activities and edges) that have been impacted by changes as a measure of risk: the more covered elements the higher the risk. Second, we can also determine the coverage of data flow entities (definitions and uses of variables) that have been impacted by changes as a measure of risk: the more covered elements the higher the risk. Regardless of the selected criterion for measuring risk, multiplying probability and risk of a *tested path* gives us the expected risk of the *tested path*, which describes the probability that the *tested path* would fail during the execution of the system under test.

Because every *tested path* is used to generate a test case in TDE/UML, we consider the expected risk as a property for a test case. Since two different risk values are derived for a *tested path*, two alternative expected risks are computed for a test case.

In addition, the execution time could be derived for a test case. Each of the two expected risk values or the execution time can be used as a criterion to prioritize test cases. For example, considering one of the expected risk value alone, users might wish to order test cases in such a way that the test cases with higher expected risk values are in the front of the test order. Then, by running these high expected risk test cases in the early phase of the testing process, users would be able to reduce the risk of a system under test as quickly as possible.

Based on the expected risk and execution time derived for test cases, two alternative approaches are proposed to prioritize test cases. The first approach is a multi-objective test case prioritization, in which we take the execution time and the expected risk value into consideration. In practice, for test cases in each test suite, users might not only prioritize test cases according to a single criterion, but take multiple objectives into account, such as reducing the risk of a system as soon as possible and executing as many test cases as possible in a limited amount of time. However, in most cases, one objective might conflict with others, which leads to a multiple objective optimization problem. One of the efficient approaches to achieve multi-objective optimization is to apply the multi-objective genetic algorithm [30], which is capable of generating a set of alternative solutions that balance different conflicting objectives. And then users can choose a solution from those alternative solutions, depending on the tradeoffs between objectives that they want to make.

The second approach is a single objective test case prioritization since users might just want to get an optimal test order in terms of one specific objective. Therefore, in this approach, test cases are prioritized according to one expected risk value. As there are

three alternative expected risk values, some experimental studies are needed to decide which expected risk value is the most efficient according to certain criteria, such as the time cost or the ability of fault detection.

In the next sub-Sections, we explain how to derive the expected risks of a tested path, i.e., the expected risk of a test case (Section 7.2.1) and execution time of a test case (Section 7.2.2). In Section 7.2.3, we describe test case prioritization as a multi-objective optimization problem and show how to apply a multi-objective genetic algorithm to solve it. In Section 7.2.4, we describe an alternative single objective test case prioritization approach.

7.2.1 Expected Risk of a Tested Path

The expected risk of a path is computed by multiplying the probability and the risk of the tested path. The risk of a test path refers to its fault-proneness and the probability of a test path refers to the probability that the tested path would be exercised during the execution of the system under test. Combining the risk and the probability, we derive the expected risk for a tested path, which indicates the likelihood that the tested path may fail during the execution of a system. Prioritizing will therefore attempt to rank first tested path (i.e., test cases) with high expected risk. The formal definition of expected risk for a tested path (or a test case) is given as: $eRisk = p * risk$, where p refers to the probability of the *tested path* and *risk* refers to the Risk of the tested path.

In the next sub-Sections, we explain how to derive the probability of a tested path (Section 7.2.1.1) and the risk of a tested path (Section 7.2.1.2) respectively.

7.2.1.1 Probability of a Tested Path

Activity diagrams can contain guard conditions associated with outgoing transitions of decision nodes. Those guards constrain variable values using specific category choices. In TDE/UML, choices are associated with probabilities of occurrence, a piece of information that is used during test case/test procedure generation (Section 3.2). We use the choices in a guard condition and their probabilities to compute (see below) the probability that the guard condition evaluates to true, which we refer to as the *true probability* of the guard condition. Then multiplying the probabilities of all outgoing transitions of decision nodes along a tested path gives us the execution probability for the tested path.

Since the syntax of guard conditions in TDE/UML is simple (recall Section 3.1.3.3), a guard can only take a limited number of forms, as discussed below:

- A guard condition is a simple expression that is only composed of two terms. These two terms, using the TDE/UML terminology (Section 3.1.3.3), can be a variable, and a choice, and they are separated by = or != operator. If the operator is =, then the *true probability* is the probability that the variable could be bound with the choice, i.e. the probability of the choice. For example, in Figure 20, both categories `Color` and `Shape` have three choices with probabilities. If a guard expression is `c = red`, where the `c` is a variable typed by the category `Color`, then the *true probability* of the guard condition is 60%. If the operator is !=, then the *true probability* of the guard condition is given by the probability that the variable could not be bound with the choice. For instance, if a guard expression is `c != red`, the probability of the guard condition is 40%.

Color	Attributes	Source	Choices	Ch	Shape	Attributes	Source	Choices	Ch
probability	Name		probability		probability	Name		probability	
black	red		60		circle	circle		50	
red	black		30		triangle	square		20	
white	white		10		square	triangle		30	

Figure 20: Example of choices with probabilities

- A guard condition is a compound expression composed of two sub-expressions separated by `or` or `and`. If the operator is `and`, the *true probability* of the compound expression is given by multiplying the *true probabilities* of the two sub-expressions. For instance, if a guard expression is `c = red and s = circle`, where `c` and `s` are variables typed by the categories `Color` and `Shape` respectively, then the *true probability* of the guard condition is 30% ($60\% * 50\%$). In the case of the `or` operator, we convert the `or` expression to an `and` expression according to the following property of Boolean logic: $A \text{ or } B = \text{not} ((\text{not } A) \text{ and } (\text{not } B))$. Therefore, the *true probability* of the `or` expression can be computed by: $1 - (1 - P_a) * (1 - P_b)$, where P_a is the *true probability* of sub-expression `A` and P_b is the *true probability* of sub-expression `B`. For example, if a guard expression is `c = red or s = square`, then the *true probability* of the guard condition is 68% ($1 - 40\% * 80\%$).
- A guard condition is a compound expression composed of two sub-expressions separated by `implies`. Since `A implies B` is equivalent to `not(A) or B`, the *true probability* of the expression `A implies B` can be computed by: $1 - P_a * (1 - P_b)$, where P_a is the *true probability* of sub-expression `A` and P_b is the *true probability* of sub-expression `B`. For instance, if a guard expression is `c = red implies s =`

triangle, then the *true probability* of this expression is $58\% (1 - 60\% * (1 - 30\%))$.

- A guard expression is an `exist` expression. The `exist` expression is true if any of the variables typed by the given category makes the expression true. For example, expression `exist c:Color | c =red` is true if there exists (at least) one variable typed by category `Color` that is given the value `red`. Thus, an `exist` expression can be converted to a set of `or` sub-expressions to compute its *true probability*. For instance, if there are three variables `c1`, `c2` and `c3` typed by category `color`, then the *true probability* of expression `exist c:Color | c =red` equals the *true probability* of expression `c1=red or c2=red or c3=red`.
- A guard expression is a `forall` expression. The `forall` expression is true if all the variables typed by the given category make the expression true. For example, expression `forall c:Color | c =red` is true if all the variables typed by category `Color` are given the value `red`. Thus, similar to the `exist` expression, a `forall` expression is converted to a set of `and` sub-expressions to compute its *true probability*. For instance, if there are three variables `c1`, `c2` and `c3` typed by category `color`, then the *true probability* of expression `forall c:Color | c =red` equals the *true probability* of expression `c1=red and c2=red and c3=red`.
- As guard expressions can have more complex forms, in which case we parse them into the sub-expressions and apply the principles discussed above to derive their *true probabilities*.

In addition, if no guards are specified on the outgoing transitions for a decision, we assign equal probabilities (i.e., 1 divided by the number of outgoing transitions) to each

outgoing transition. This is in accordance with the way test cases are created with TDE/UML.

7.2.1.2 Risk of a Tested Path

The risk of a *tested path* refers to its fault-proneness. The prioritization technique described in [10] is based on the conjecture that new defects are more likely introduced by the recent changes to the program. This assumption denotes that the changed part of program is considered to be more fault-proneness. We make a similar conjecture for TDE/UML: the new defects are more likely introduced by the recent changes to a TDE/UML model. We propose two alternative ways to evaluate the risk of a *tested path* by accounting for the model elements that have been changed or impacted by changes (see definitions below). This risk value is computed based on the analysis of model elements in the second version of a model. The next two subSections explain three ways to compute the risk of a tested path based on this notion of elements changed impacted by changes.

7.2.1.2.1 Coverage of Retestable Control Flow Entities

The model elements on a tested path are control flow entities, including nodes (e.g. activities, decision nodes, fork/join nodes and etc.) and transitions in an activity diagram. We use the term *retestable control flow entities* to refer to the set of control flow entities that have been changed or affected by other changes (i.e. changed categories). Thus the first criterion is based on the coverage of *retestable control flow entities*. A path that covers a large number of retestable control flow entities leads to a high risk if it is not prioritized first. Because a *tested path* in the modified model might contains added

control flow entities (i.e. added decisions or added transitions), thus traceability links between those added control flow entities and the retestable test procedure should be established before we derive the coverage of the *retestable control flow entities*.

To formally define the impacted control flow entities, we add one more attribute `isRetestable` to class `Change` (see Figure 17). The formal definitions of retestable control flow entities are given the follows:

- 1) Retestable nodes: A node in an activity diagram may be linked to several notes and each note contains several constraints or variable declarations. Therefore, a retestable node is identified if it is linked to one or more changed notes (see Section 5.2.3), or its attached notes contain variable declarations that use one or more changed categories. Because there are five kinds of nodes allowed in an activity diagram, for the sake of simplicity, we formalize the retestable nodes in the context of activity as an example.

```
context Activity
inv: (getAttachedNote(self)-
>exist (change.type=ChangeType.changed)
or
getAttachedNote(self).varDeclaration.define.type
->exist (change=ChangedType.changed) )
implies self.change.isRestable=true)
```

- 2) Retestable decisions: Because an added decision or changed decision could also appear in a tested path, we also need to identify it as a kind of retestable control flow entity.

```
context Decision
inv: (self.change.type=Change.added
or self.change.type=Change.changed)
implies self.change.isRestable=true
```

- 3) Retestable transition: There are three cases that a transition is considered as retestable transition: 1) the transition is an added transition; 2) the transition has a

changed guard condition; 3) one of the categories involved in its guard condition is changed.

```
context ActivityEdge
inv: (self.change.type=Change.added
      or self.change.type=Change.changed)
      implies self.change.isRetestable=true)
      and
      GetInvolvedCategory(self) -> forall (c :Category |
      c.change.type = Change.changed)
      implies self.change.isRetestable = true
```

Thus, the risk value in terms of the coverage of retestable control flow entities for a tested path is defined as follows:

$$Risk_{cf} = \frac{\text{numberOf ReTestableControlFlowEntitiesOfTestedPath}}{\text{numberOf ReTestableControlFlowEntitiesOfTestSuite}}.$$

7.2.1.2.2 Coverage of Retestable Data Flow Entities

Here, we propagate the changes on model elements to data flow entities and then identify the data flow entities that have been impacted. First, the data flow entities have to be formally defined. Similar to data flow information retrieved from the source code, we define two kinds of data flow entities in a TDE/UML model: a variable definition and a variable use. A variable definition refers to a variable declaration where the variable is typed with a category. A variable use refers to a guard condition or a constraint in a note involving that variable. Since both guard conditions and constraints in notes are predicate expression, all uses refer to predicate uses (p-use) in a TDE/UML model. Therefore, two new classes are added to the TDE/UML metamodel (Section 3.3) to define these two kinds of data flow entities. A `Definition` is associated with a `VarDeclaration`. A `Use` is associated with a `VariableExpression` involved in a constraint in an activity diagram, which can be a guard of a transition or a constraint in a note.

Then the formal definition of impacted data flow entities are given as follows:

- **Retestable Definition:** If the variable declaration for a definition involves a changed category, then this definition is identified as a retestable definition.

```
context Definition
inv: self.varDeclaration.define.type.change=ChangeType.changed
      implies self.isretestable=true
```

- **Retestable Use:** A use of a variable linked to a changed category is a retestable use.

```
context Use
inv: self.variableExpression.represent.type.change.type
      =ChangeType.changed
      implies self.isretestable=true
```

Thus, two alternative ways to compute the risk value in terms of the coverage of retestable data flow entities for a tested path can be defined:

$$\text{Risk}_{\text{df-Defs}} = \frac{\text{numberOf ReTestableDefinitionsOfTestedPath}}{\text{numberOf ReTestableDefinitionsOfTestSuite}};$$

$$\text{Risk}_{\text{df-Uses}} = \frac{\text{numberOf ReTestableUsesOfTestedPath}}{\text{numberOf ReTestableUsesfinitionsOfTestSuite}}.$$

Note that since variables in a TDE/UML model are only defined once in the whole model, accounting for definition-use pairs for computing a risk (similarly to what is done for source code) will be identical to accounting for uses: there would be as many definition-use pairs as uses.

7.2.2 Execution Time of a Test Case

Assuming the execution time of test cases has been recorded before regression test selection and prioritization, the execution time of retestable test cases can be used for prioritization purposes. We can for instance prioritize test cases such that those with a smaller execution time are placed in the front of the test order and then executed earlier in

the test process. As a result, users would be able to execute more test cases within a limited amount of time.

7.2.3 Multi-Objective Test Case Prioritization

In practice, based on the three expected risk values and the execution time of test cases, users might want to take more than one objective into consideration when prioritizing the test cases of a test suite, such as maximizing the expected risk and minimizing the execution time. However, in most cases, it is not always possible to find one best solution that can satisfy all objectives as objectives might conflict with each other. This kind of problem is referred to as a multi-objective optimization problem (MOP), as opposed to a single objective optimization problem, where only one objective is taken into account. The multi-objective genetic algorithms (MOGA) [31] have been proven to be well suited for solving multi-objective optimization problems, since they are capable of providing a set of alternative solutions that balance different (possibly conflicting) objectives. Then users can choose a solution from those alternative solutions, depending on the tradeoffs between objectives that they want to make. Hence, in our prioritization technique, we decide to apply the MOGAs to solve the multi-objective test case prioritization optimization problem.

In the next sub sections, we present an overview of the MOGAs (Section 7.2.3.1) and explain the tailored MOGA that is suited to our test case prioritization problem (Section 7.2.3.2).

7.2.3.1 Multi-Objective Genetic Algorithms Overview

In this section, we provide an overview of the multi-objective genetic algorithms. Before introducing the MOGAs, we give a brief introduction on the Genetic Algorithms in Section 7.2.3.1.1. Next, we describe the basic principles of the multi-objective problems and discuss the different multi-objective genetic algorithms that have been used to solve MOPs, in Section 7.2.3.1.2. Finally, the strength pareto approach (SPEA2), which is one of MOGAs and is applied to solve multi-objective test case prioritization of TDE/UML, is described in Section 7.2.3.1.3.

7.2.3.1.1 Genetic Algorithms

Genetic algorithms are adaptive search techniques typically used to solve optimization problems. Genetic algorithms were first introduced by John Holland [32] to explain the process of natural selection. In GA terminology, a possible solution of a given problem is encoded as a chromosome. Normally, it is represented by a string of values. Every element in the string refers to a gene that determines one or more characteristics of a chromosome. A set of chromosomes is called a population. GAs process a population of chromosomes to generate an optimal solution by repeatedly replacing one population with another. Each chromosome, in each generation, is given a fitness value, according to a fitness function. High-fitness chromosomes are identified as better solutions compared with low-fitness chromosomes. These high-fitness chromosomes are selected and recombined to produce a new population, by crossover and mutation. A typical crossover takes two chromosomes and randomly determines a crossover point. As a result, two selected chromosomes are divided into two portions: the left portion and the right portion.

Then, either the right or the left portion could be swapped to generate the new chromosomes. For example, in Figure 21, two children chromosomes, which are also called offspring, are generated by swapping the right portion of two selected chromosomes. Note that not all of the chromosomes of a population are selected for crossover. A crossover rate is specified by GA users to determine the probability of crossover that applies to selected chromosomes.

Figure 21: A crossover example

A typical mutation is only applied to a chromosome after crossover, and is done by altering one or more of the values of its genes. For example, Figure 22 shows that the fifth gene of an offspring is mutated. Also the probability of mutation depends on the mutation rate given by GA users.

Figure 22: A mutation example

Thereafter the old generation is replaced by the new one, and the evolutionary process continues until all the individuals converge to an optimal solution. During this evolutionary process, the good characteristics of high-fitness chromosomes are mixed and exchanged to generate better solutions as the population evolves.

7.2.3.1.2 *Multi-Objective Genetic Algorithm*

In the case of test case prioritization, we try to maximize the expected risk and minimize the execution time. Since the two objectives are possibly conflicting, we have to find a tradeoff between them. This kind of problem is referred to as a multi-objective optimization problem and can be formally defined as follows [31]:

Definition 1: The Multi-Objective optimization problem (MOP):

Given a solution space, X , and a set of objective functions for k objectives for an element x of X , $F(x) = \{f_1(x), \dots, f_k(x)\}$, a MOP solution is to find a vector of decision variable $x^ = \{x_1, \dots, x_n\}$ in the solution space to minimize $F(x^*) = \{f_1(x^*), \dots, f_k(x^*)\}$, subject to constraints. Without loss of generality, we consider all objectives are minimization objectives.*

For a multi-objective problem, every solution is a set of fitness values, one for each objective. The most intuitive approach, which is called the “weighted sum approach”, converts a multi-objective problem to a single objective problem by adding up every weighted objective function. In other words, all objectives are assigned a weight to indicate their importance to users and then summed up, in order to produce an overall fitness value for every solution. Then a single objective GA selects the individuals with the higher overall fitness values, to produce the offsprings, and finally yields a solution

according to a given weight combination. To generate multiple solutions, the GA has to be applied several times with different weight combinations. In this weighted sum approach, the solutions are sensitive to the values of weight combinations. However, it is not always easy to select a proper weight combination before hand because the choice is quite subjective and requires domain knowledge. Moreover, in some cases, it is meaningless to combine separate objectives together since they are completely different criteria.

To overcome the problem of the weighted sum approach, the Pareto-ranking approach that uses the concept of Pareto-optimality has been introduced [33]. Before discussing the Pareto-ranking approach, some concepts are defined as follows:

Definition 2: Pareto Dominance

A vector $u^ = \{u_1, \dots, u_n\}$ is said to dominate $v^* = \{v_1, \dots, v_k\}$ (denoted as $u^* \prec v^*$) if and only if, u^* is partially less than v^* , i.e., $\forall i \in \{1, \dots, k\}, f_i(u^*) \leq f_i(v^*) \wedge \exists j \in \{1, \dots, k\}, f_j(u^*) < f_j(v^*)$.*

Definition 3: Pareto Optimality

For a given multi-objective problem, a solution x is said to be Pareto Optimal if it is not dominated by other solutions in the solution space.

Definition 4: Pareto Optimal Set

The pareto optimal set P^ is the set of all the non-dominated solutions, which can be defined as: $P^* := \{x \in \Omega \mid \neg \exists x' \in \Omega : F(x') \prec F(x)\}$.*

Definition 5: Pareto Front

Given the Pareto optimal set P^ and objective function $F(x)$ for a multi-objective problem, the Pareto Front PF^* is defined as: $PF^* := \{ u^* = F(x) \mid x \in P^* \}$.*

In order to find solutions to multi-objective optimization problems, the Pareto-ranking approach explores a Pareto optimal set, in which elements are not dominated with respect to each other. Any Pareto optimal solution cannot always be considered to be better than others, because although it may be superior to other optimal solutions with respect to some objectives, it is considered to be inferior in other objectives. However, every Pareto optimal solution can be an acceptable solution (i.e., it is an acceptable trade-off). Choosing which solution is superior to others requires that the decision maker have a good knowledge of the problem domain.

There are many multi-objective genetic algorithms (MOGAs) based on the Pareto-ranking. Among the Pareto-ranking MOGAs, the NSGA2 and SPEA2 are considered to be among the most efficient approaches [34]. In terms of time complexity, both NSGA2 and SPEA2 need $O(MN^2)$ [35], where M is the number of objectives and N is the population size. However, SPEA2 provides a better way to estimate the density of solutions surrounding a particular solution in a population so that the search of optimal results is more effective, especially when the number of objectives increases (and in particular for a number of objectives greater than 4) [36]. As a result, we choose the SPEA2 algorithm as the implementation of the MOGA algorithm for our test case prioritization problem.

7.2.3.1.3 The SPEA2 Algorithm

An overview of the SPEA2 algorithm is provided in Figure 23. In step 2, each individual i in both the external set $\bar{P}_t$ and the population P_t is assigned a strength value $S(i)$ which presents the number of individuals that it dominates. Then, based on the strength value, the raw fitness $R(i)$ is calculated for each individual, as follows:

$$R(i) = \sum_{j \in P_t + \bar{P}_t, j \succ i} S(j), \text{ where } S(j) = |\{k \mid k \in P_t + \bar{P}_t \wedge j \succ k\}|$$

An individual is considered to be a non-dominated individual when the raw fitness value is zero. As a result of the fact that most individuals do not dominate each other, additional density information is needed to discriminate between individuals that have the same raw fitness values. SPEA2 uses the k th Nearest Neighbour technique to that aim. The distances of each individual in the object space to all individuals both in the external set $\bar{P}_t$ and the population P_t are calculated. Then distances are stored in increasing order in a list. The density estimation is computed according to the k th element in the list, denoted as σ_i^k , where k is $\sqrt{N + \bar{N}}$. And the Density value of individual i defined as:

$$D(i) = \frac{1}{\sigma_i^k + 2}. \text{ Finally, the fitness of individual } i, F(i), \text{ is computed by: } F(i) = R(i) + D(i).$$

During the environmental selection, shown as step 3 in Figure 23, the SPEA2 algorithm copies all the non-dominated individuals from the external set and the population to the external set of the next generation: $\bar{P}_{t+1} = \{i \mid i \in P_t \cup \bar{P}_t \wedge F(i) < 1\}$.

If the size of the external set $\bar{P}_{t+1}$ exceeds $\bar{N}$, then truncation occurs to remove extra individuals from $\bar{P}_{t+1}$ until the size of $\bar{P}_{t+1}$ is equal to $\bar{N}$. If the external set $\bar{P}_{t+1}$ is less

than $\bar{N}$, the individuals of P_t and $\bar{P}_t$ are sorted by their fitness values and then the first $\bar{N} - \bar{P}_{t+1}$ individuals are copied into $\bar{P}_{t+1}$.

Input:	N,	population size;
	$\bar{N}$,	archive size;
	T,	number of generations
Output:	A,	nondominated set;
Step 1: Initialization: Generate an initial population P_0 and create an empty archive population $\bar{P}_0 = \phi$. Set $t = 0$.		
Step 2: Fitness Assignment: Calculate fitness values of individuals in P_t and $\bar{P}_t$.		
Step 3: Environmental Selection: Copy all nondominated individuals in P_t and $\bar{P}_t$ to $\bar{P}_{t+1}$. If size of $\bar{P}_{t+1}$ exceeds $\bar{N}$ then reduce $\bar{P}_{t+1}$ by means of the truncation operator, otherwise if size of $\bar{P}_{t+1}$ is less than then fill $\bar{P}_{t+1}$ with dominated individuals in P_t and $\bar{P}_t$.		
Step 4: Termination: If $t \geq T$ then set A to the set of decision vectors represented by the nondominated individuals in $\bar{P}_{t+1}$. Stop.		
Step 5: Mating selection: Perform binary tournament selection with replacement on $\bar{P}_{t+1}$ in order to fill the mating pool.		
Step 6: Variation: Apply crossover and mutation operators to the mating pool and set P_t to the resulting population. Increment generation counter ($t=t+1$) and go to Step 2.		

Figure 23: SPEA2 algorithm [36]

7.2.3.2 Tailored MOGA

In this Section, we have tailored various parameters used in the SPEA2 algorithm to solve our multi-objective test case prioritization problem. The size of population and archive are discussed in Section 7.2.3.2.1. The representation of the chromosome is described in Section 7.2.3.2.2. Crossover and mutation operators are addressed in Sections 7.2.3.2.3 and 7.2.3.2.4, respectively. Finally, we present two optimization schemes (i.e., fitness functions) used in our prioritization approach in Section 7.2.3.2.5.

7.2.3.2.1 Population Size and Archive Size

The population size is a critical parameter in a GA. If using a small population size, GA will quickly converge to a local minimum, since it insufficiently samples the search space. However, a large population causes the GA to spend more time on finding an optimal solution. For the population size used in single objective GAs, Grefenstette et al. [37] propose a range from 30 to 80 individuals (chromosomes) in the population while Schaffer et al. [38] suggest a smaller population size, between 20 and 30. For MOGAs, Zitzler et. al. suggest that larger population sizes than those of single objective GAs be used and that population sizes be proportional to the number of objectives [36]. To be consistent with the above findings, we decide to use the 64 individuals in the population for each objective. Two objectives are defined for each optimization scheme in section 7.2.3.2.5 and therefore the population size used for our SPEA2 implementation is 128.

Choosing the value of the archive size is also important as the archive size determines the elitism (because of the truncation procedure) of the SPEA2 algorithm. An archive that is too small in size causes some non-dominated (potentially good) solutions to be discarded (during truncation) since there is not enough space to store them. Systematic studies on how the archive size would impact on the performance of MOGAs (and therefore on how to set that parameter) are not readily available. One exception is [36], where an archive size in the range $[1/4, 4]$ of the population size is suggested. We chose 64 (half of the population size) as the archive size for our SPEA2 algorithm.

7.2.3.2.2 *Chromosome*

In GAs, a chromosome represents a solution to the given problem. For our multi-objective optimization problem, we apply GAs for prioritizing the test cases of a test suite. Thus, an order of test cases of a test suite is encoded as a chromosome and a test case of that order is a gene of the chromosome. Because we consider the expected risk and execution time of a test case when prioritizing test cases, each gene has to hold these two values so that the genetic algorithm is able to evaluate the fitness functions.

The length of a chromosome, which is equivalent to the number of its genes, is equal to the number of retestable test cases of a test suite. Since the number of re-testable test cases is fixed within a test suite, the length of a chromosome is fixed too.

7.2.3.2.3 *Crossover Operator*

In a GA, a crossover operator is used to pass desirable genes from the current generation to the next generation. It combines two individuals from the current generation, (also called parent chromosomes), to create two new individuals for the next generation, called child chromosomes. Several kinds of crossover operators have been proposed for different purposes. The partially matched crossover (PMX) [33] is a crossover operator aiming to solve the permutation problem, which fits our multi-objective test case prioritization. Thus, we choose PMX operator as our crossover operator for the SEPA2 algorithm.

With PMX, the two parent chromosomes are aligned and two points are randomly selected at the same place on the two parent chromosomes. The fragment between these

two points is defined as a matching section. As shown in Figure 24, the matching sections in Parent 1 and Parent 2 are (4,3,7,6) and (1,2,5,3) respectively, assuming the two randomly selected points are after the second gene and after the 6th gene. Then the two parent chromosomes swap their mapping sections to produce two intermediate child chromosomes: child 1' and child 2' in Figure 24. This will (likely) result in duplicated genes in intermediate child chromosomes: e.g. gene 2 appears twice in child 1'. A mapping relationship (noted $\leftrightarrow$) is then identified to remove gene duplication, i.e. to identify how the duplicated genes before and after the matching section have to be changed. The mapping relationship between two genes, one from the child chromosome (gc) in the matching section and one from the parent chromosome (gp), is defined as follows: $gc \leftrightarrow gp$ if and only if gc and gp appear at the same place in the two chromosomes (i.e. at the same index in the sequences of genes). In Figure 24, the mappings are (4 $\leftrightarrow$ 1), (3 $\leftrightarrow$ 2), (7 $\leftrightarrow$ 5) and (6 $\leftrightarrow$ 3). Then, for each of two intermediate child chromosomes, if there is a conflict between a gene outside the matching section and a gene in the matching section, then the one outside the matching section is replaced by another gene, according to the mappings. For example, the first gene of child 1' would be '2' like the first gene of the parent 1. However, the second element of the mapping sections of child 1' is also '2'. Thus, according to mappings (3 $\leftrightarrow$ 2) and (6 $\leftrightarrow$ 3), we choose '6' as the first gene in child 1. Finally, two child chromosomes are created after removing all conflicts.

Figure 24: Example of the PMX operator

The crossover rate is also a critical factor that determines the performance of a GA. If the crossover rate is too high, the algorithm will not converge to optimal results. In contrast, if the crossover rate is too low, the search space will not be fully explored [39]. De Jong suggests a desirable crossover rate of 0.6 [40]. Based on De Jong's work, Grefenstette et al. recommend that the crossover rate should be in the range [0.45, 0.9] [37]. To be consistent with these findings, we choose 0.7 as the crossover rate for our SPEA2 algorithm.

7.2.3.2.4 Mutation Operator

The mutation operator is applied at the last step when creating a new generation. Each newly generated individual has a random chance to mutate. The purpose of mutation is to ensure that the GA adequately explores the search space. We choose the swap mutation operator [41] for our SPEA2 implementation. This operator randomly chooses two genes in a chromosome and interchanges their positions. For example, consider a chromosome a-b-c-d-e-f. If genes and position 1 and 3 are selected, then the resulting chromosome is c-b-a-d-e-f.

The mutation rate is also critical to the performance of the GA. Various mutation rates have been proposed in the literature. De Jong [40] suggests a mutation rate of 0.001. Grefenstette et al. [37] suggest a rate of 0.01. In [42], Smith et al. state that mutation rates that relate to the chromosome length and population size have a significantly better performance than those that do not. Moreover, Schaffer et. al. [38] suggest a rate of $\frac{1.75}{\lambda\sqrt{l}}$, where λ is the population size and l is the length of chromosome. To be consistent with the above findings, we choose the mutation rate suggested by Schaffer et al.: $\frac{1.75}{\lambda\sqrt{l}}$.

7.2.3.2.5 Optimization Schemes

Based on the three expected risk values discussed previously (section 7.2.1) and the execution time of test cases, we propose the following optimization schemes to satisfy two different users' testing requirements: 1) users have a limited budget; 2) users want to reach a certain expected risk threshold.

Before performing the multi-objective test case prioritization, users are required to selected a risk measure (i.e. $Risk_{cf}$, $Risk_{df-Defs}$, $Risk_{df-Uses}$) to contribute the expected risk $eRisk$ values of test cases. Then, for a test suite, the $eRisk$ value of each test case is normalized as: $nERisk = eRisk / sumOfERisk$, where $nERisk$ refers to the normalized expected risk value of a test case, and $sumOfERisk$ refers to the summation of the expected risk values of all the test cases within the test suite.

a) Optimization scheme one – limited budget

In this optimization scheme, users have a limited budget with which to execute each test suite. Given an order of test cases, a user would execute test cases from the start of the order until the time budget is met. This optimization therefore has two objectives: 1) reaching the highest cumulative *nERisk* value possible within the budget; 2) increasing the cumulative *nERisk* value at a fast rate.

Since the execution times of test cases are different, given different test cases orders, the number of test cases that have been executed in a limited amount of time could be different. Thus, prior to defining the fitness functions for the above two objectives, we have to determine the number of test cases in a test cases order that can be executed to within the time constraint: algorithm in Figure 25.

<p>Inputs: time, the time constraint; t, a test case order.</p> <p>Output: n, the number of test cases that can be executed in t within time constraint time.</p> <p>Algorithm:</p> <p>Step 1: n=1 and time'=0;</p> <p>Step 2: add the execution time of the nth test case to time'.</p> <p>Step 3: if time' is less than time (and we have not reached the end of t), then increase n by 1 (n=n+1) and go to step 2; otherwise, the number of test cases in t that can be executed within time constraint time is n.</p>
--

Figure 25: The algorithm to find the number of executed test cases for optimization scheme one.

According to the value of n, the number of test cases that can be executed within the time constraint, the fitness functions in this optimization scheme are given as follows:

- The Rate of increasing the cumulative *nERisk* value for a test order. The fitness function to measure this rate is defined as:

$$f1 = \sum_{i=1}^n \frac{1}{2} \times nERisk(i) \times ((T - \sum_{j=1}^{j=i-1} time(j)) + (T - \sum_{j=1}^{j=i} time(j)));$$

where T is the user specified time constraint, n is the number of test cases that can be executed within T , and $nERisk(i)$ and $Time(j)$ are the $nERisk$ value and the execution time of the j^{th} test case, respectively. Higher values of fI indicate a faster rate at which to reach higher the cumulative $eRisk$ value within the given time constraint.

The following example, with ten test cases labeled 0 through 9, illustrates the above fitness function. The $nERisk$ and execution time for each test case are shown in Table 9. Consider the following two test cases orders: order one (1-0-2-5-4-3-6-7-8-9) and order two (7-0-8-4-1-9-2-3-6-5). Figure 26 shows the cumulative risk value versus the total execution time for these two test case orders. The value of fI is represented by the area surrounded by the curve, the x-axis, the time constraint (vertical) line and the horizontal line that represents the cumulative $nERisk$ that the given test order achieves before the time constraint. As shown in Figure 26, assuming the time constraint is 350 time units, both two orders can only execute their first five test cases. The cumulative $nERisk$ for order one and order two with 350 time units are 0.33 and 0.27 respectively. It is clear that order one has a faster rate at which the cumulative $nERisk$ is increased.

Test Case	$nERisk$	time	Test Case	$nERisk$	time
Test Case 0	0.08	84	Test Case 5	0.03	97
Test Case 1	0.04	47	Test Case 6	0.29	20
Test Case 2	0.09	25	Test Case 7	0.03	76
Test Case 3	0.23	51	Test Case 8	0.03	8
Test Case 4	0.09	49	Test Case 9	0.09	97

Table 9: Example $nERisk$ and time values of a test suite

Figure 26: Values of $f1$ for test case orders one and two (Table 9)

- The Cumulative $nERisk$ Value for a test order. The fitness function, which is used to measure the cumulative eRisk value that a test case order reaches with in a given time constraint, is defined as:

$$f2 = \sum_{i=1}^n nERisk(i);$$

where n is the number of test cases that can be executed within the time constraint, and $nERisk(i)$ is the $nERisk$ value of the i^{th} test cases. A test case order with higher values of $f2$ indicates it reaches higher cumulative $nERisk$ value within the given time constraint.

In the example of Figure 26, test order 1 has a higher cumulative $nERisk$ value than test order 2.

- Optimization scheme two – specifying the threshold for cumulative expected risk

In this optimization scheme, users are required to specify a threshold for the cumulative *nERisk* value. They would execute test cases, starting with the first test cases in the solution and stopping when the specified threshold is reached or exceeded. Thus two objectives are proposed that users will want to achieve in terms of this optimization scheme: 1) reaching/exceeding the threshold of the cumulative *nERisk* value at a fast rate; 2) minimizing the overall execution time in order to reach the specified threshold.

Prior to computing the fitness value for different test case orders, we have to determine the number of test cases that can be executed within the threshold of the cumulative *nERisk*: algorithm in Figure 27.

<p>Inputs: S_nERisk, the threshold of the cumulative <i>nERisk</i> specified by users; t, a test case order.</p> <p>Output: n, the number of test cases that can be executed within S_nERisk;</p> <p>Algorithm:</p> <p>Step 1: n=1, S_nERisk'=0</p> <p>Step 2: add the <i>nERisk</i> of the n^{th} test case to S_nERisk'.</p> <p>Step 3: if S_nERisk' is strictly less than S_nERisk (and we have not reached the end of t), then increase n by 1 ($n=n+1$) and go to step 2; otherwise, the number of test cases that can be executed within S_nERisk for test case order t is n.</p>

Figure 27: The algorithm to find the number of executed test cases for optimization scheme two

According to the value of n, the fitness functions of this optimization scheme are given as follows:

- Rate of reaching the threshold of the cumulative *nERisk*. The fitness function, to measure the rate at which a test case order reaches/exceeds the threshold of the cumulative *nERisk*, is defined as:

$$f3 = \sum_{i=1}^n \frac{1}{2} \times \text{Time}(i) \times ((S_nERisk - \sum_{j=1}^{i-1} nERisk(j)) + (S_nERisk - \sum_{j=1}^{i-1} nERisk(j)))$$

$$+ \frac{1}{2} \times \left(S_nERisk - \sum_{i=1}^n nERisk(i) \right)^2 \times \frac{\text{time}(n)}{eNRisk(n)};$$

where n is the number of test cases that can be executed within the threshold, $nERisk(i)$ and $\text{Time}(i)$ are the $nERisk$ value and the execution time of the i^{th} test case, respectively, and S_nERisk is the user specified threshold of cumulative $nERisk$ value. Fitness function $f3$ represents the area bounded by the curve of cumulative expected risk, the y-axis and the (horizontal) line of the user specified threshold, as illustrated in Figure 28. Lower values of $f3$ indicate a faster rate at which to reach/exceed the threshold of the cumulative $nERisk$. In Figure 28, the surrounded area of test order two is 123.735 while the surrounded area of test order one is 104.38. It is clear that test order one increases its cumulative $nERisk$ faster when it reaches/exceeds the threshold.

We describe and illustrate how this fitness function indeed represents the expected area in Appendix A.5.

Figure 28: Values of f3 for test case orders one and two (Table 9)

- The overall execution time to reach the threshold. The overall execution time to reach/exceed the threshold is given as:

$$f4 = \sum_{i=1}^n time(i) ;$$

where n is the number of test cases that can be executed within the threshold, and $time(i)$ is the execution time of the i^{th} test case. A test case order with lower value of $f4$ means that it can reach/exceed the threshold with less execution time.

Although only two objectives are defined in each optimization scheme, the multi-objective prioritization approach could be easily adapted for an optimization scheme with more objective functions (three or more objective functions) or with different objective functions as the ones we have used.

7.2.4 Single Objective Test Case Prioritization

The single objective test case prioritization is proposed when execution time is not available. In this case, users might only want to prioritize test cases in terms of a single objective (risk), rather than according to multi-objectives. Thus, we only prioritize test cases according to a single expected risk value, and test cases are placed in the descending order of their expected risks. If multiple test cases have the same expected risk, we order them randomly.

7.3 Test Procedures Prioritization

Finally, we prioritize retestable test procedures in each test case. In TDE/UML, all of the test procedures for a test case have the same number of input variables and the same types for those input variables. But the test generator assigns different choices of categories to the values of the input variables. Each choice of a category has a probability attribute which specifies the probability that the choice will be chosen as the value of an input variable during test generation. Considering the example in Figure 20, a Category `Color` contains three choices: `black`, `red` and `white`. The choice `red` has a probability of 60, while the choice `white` only has a probability of 10. The probability attribute of a choice indicates its (expected) usage frequency. A choice with higher probability value is used more frequently and requires more testing effort. As a test procedure could have several input variables assigned with different choices of categories, the priority of the test procedure can be computed by multiplying all the probabilities of all the choices used in this test procedure. The formal definition of probability of a test procedure is given as:

$$\text{Probability} = \prod_{i=1}^n P_v(i),$$

where n is the number of variables used in the test procedure, and $P_v(i)$ refers to the probability of the choice that is assigned to the variable i of the test procedure. As mentioned in Section 7.2.1.1, the probability of a choice can be automatically retrieved from a TDE/UML model. Thus the process to derive the probability of a test procedure can be automated too. Then, the retestable test procedures of a test case can be ranked according to their probabilities. Test procedures with higher probabilities are placed in the front of the test order and selected to execute first. When multiple test procedures have the same probabilities, we order them randomly.

7.4 Overall sequence for test procedures

After prioritizing test suites, test cases and test procedures by applying techniques discussed in the above sections, an overall test procedure sequence is obtained: for each test suite in the ranking of test suites, we rank test cases of the test suite, and for each test suite we rank test procedures. Users will execute re-testable test procedures according to the overall test procedure sequence to perform the regression testing.

However, users might not have enough resources to execute all those test procedures and prefer to execute a subset of those ranked test suites, test cases and test procedures. In other words, if there are not enough resources to execute all the test procedures following the above mentioned method, there is a risk that test suites (and therefore functionalities) ranked last may not even be exercised/tested. Users may therefore, want to make sure that every changed functionality be exercised at least once. In this case, the problem becomes:

How to select a subset of test cases within a test suite and how to select a subset of test procedures within a test case? We argue below that in general the number of retestable test procedures for a test case is relatively small and that therefore it is not worth trying to perform a selection of test procedures per test cases. We therefore argue that the problem becomes: How to select a subset of test cases within a test suite?

Recall (section 3.2) that test procedures are generated by TDE/UML according to a criterion and the categories and choices (i.e., the application of Category-Partition) defined for input parameters of activity diagrams (i.e., input parameters of use cases). TDE/UML applies one of six choice-coverage-algorithms (i.e., criteria) to choose the combinations of choices as inputs to generate test procedures of a test case. Thus, the number of generated test procedures depends on the number of combination of choices to select according to the criterion. If the most complicated criterion, “exhaustive” is applied, TDE/UML chooses all possible combinations of choices for parameters of a test case. If the simplest criterion, “sampling” is applied, TDE/UML only chooses one choice for a parameter. According to our knowledge of applying Category-Partition in the context of TDE/UML, the number of parameters of a use case (and therefore test case) is small, and in general it is less than or around three. Each variable is typed by a category, which usually has less than or around five choices. Given these observations, if the “exhaustive” criterion is used, then TDE/UML generates less than or around 125 test procedures per test case. If instead the “sampling” criterion is used, then TDE/UML only generates one test procedure per test case. If any other criterion is applied, the number of generated test procedures of a test case is (in general) between one and 125. In most cases, users won’t apply the “exhaustive” criterion since it generates too many test

procedures. Furthermore, during regression testing, after test case selection, we prioritize retestable test procedures. We therefore expect that the number of retestable test procedures within a test case will be small in general. As a result, we believe that it is not worthwhile to select a subset of test procedures within a test case. Future work will confirm or infirm whether our intuition on the number of retestable test procedures for test cases.

On the other hand, we can expect that the number of test cases within a test suite will be much larger than the number of test procedures for a test case. Therefore, it is more meaningful to select a subset of test cases within a test suite. Since two test case prioritization approaches (multi-objective and single-objective) are proposed to prioritize test cases within a test suite, we have to find ways to select a subset of test cases in both cases.

Recall that when performing multi-objective test case prioritization, a stop point is computed for each of the optimization schemes (see Section 7.2.3.2.5). Then if we apply multi-objective prioritization, we already obtain a subset of test cases of a test suite.

In the case of single-objective test case prioritization, one drawback of the approach is that test cases belonging to different test suites are prioritized separately: Test suites are first prioritized according to user inputs (Section 7.1) and then test cases are prioritized within test suites according to their expected risk values. This assumes that if test suite A is deemed more important than test suite B by the user, then all the test cases in A are more important than all the test cases in B. When more than one test suite is prioritized, this may result in some test cases being prioritized before other test cases that have a

higher risk (but belong to a test suite prioritized later one). This is illustrated in Figure 29 where the i^{th} test case of the first test suite is prioritized before the 1^{st} test case of the second test suite, although the latter may have a much higher expected risk than the former.

Figure 29: the result of test cases prioritization

To alleviate this (potential) drawback, we divide the test cases of a test suite into two parts by setting a user-defined stop point. This stop point specifies a point where the benefit of adding more test cases in terms of additional (cumulative) risk is deemed insufficient given the additional cost required. Then, only the test cases that belong to the first part of each test suite are finally kept for regression testing. This stop point can be visualized by plotting the cumulative risk against the number of test cases. If the cumulative risk attained for each additional test case reaches a plateau or is at least less steep, we stop: the steepness of the curve is user defined. (The stop-point specification can be set by the user according to budget constraints.) This is illustrated in Figure 30, where the first part of the curve is steeper than the last part, suggesting that the first part is more cost-effective than the second part; adding x test cases only brings an increase of y in cumulative expected risk.

Figure 30: Cumulative expected risk and test suite fraction

Given user-defined values x and y , the stop point for each test suite can be determined by the algorithm of Figure 31. The algorithm keeps updating the increased cumulative expected risk x' for the next y percentage test cases, followed by test case i . If x' is less than x , which is specified by the user, then the algorithm stops at test case i . Otherwise, the algorithm continues searching for the stop point.

Inputs:	x , the percentage of a test suite, y , the (minimum) increased cumulative expected risk. t , a prioritized test suite,
Output:	the stop point;
Algorithm:	
Step 1:	start from the first test case in the prioritized test suite t , set $i=1$.
Step 2:	calculate the increased cumulative expected risk y' for the next x percentage test cases followed by the test case i ;
Step 3:	compare y' to y . if y' is more than y , then increase i by 1 ($i=i+1$) and go to Step 2; otherwise the stop point of the test suite is test case i .

Figure 31: the algorithm to find the stop point for each test suite

8 AUTOMATION

The purpose of this chapter is to describe the automation of our test selection and prioritization techniques for TDE/UML. First, in Section 8.1, we describe the overall procedures of our test selection and prioritization tool and the basic design architecture in terms of packages. Then, as a genetic algorithm package named ECJ is used to implement the SPEA2 algorithm during test prioritization, we describe how we adapted that genetic algorithm to our test approach in Section 8.2.

8.1 Tool Overview

Figure 32 gives a summary of the steps of our test selection and prioritization tool. At the first step, the original version and modified version of a TDE/UML model are loaded, as well as the existing test suites generated for the original version model. Then the traceability links between the test procedures of the existing test suites and the original model elements are established in the step 2, following the discussion of Section 4. Following the establishment of traceability links, in step 3, the changes between two versions of a TDE/UML model are detected based on a set of rule defined in Section 5. Then, by analyzing the impact of changes detected, in step 4, the test artifacts in the TDE/UML, including test suites, test cases and test procedures, are classified into three categories: Obsolete, Retestable and Reusable (Section 6). At step 5, test artifacts that are identified as retestable in the previous step are prioritized respectively using the strategies mentioned in Section 7.

Figure 32: Steps of Test Selection and Prioritization Tool

The design architecture of our test selection and prioritization tool, as shown in Figure 33, is simply divided into four packages, which are `Traceability Repository`, `Model Changes`, `Classification` and `Prioritization`, according to the different functionalities that they provide. Since our tool is a plug-in of the TDE/UML, we have to use the API provided by TDE/UML to obtain the TDE/UML diagrams' information. Thus, we also present the TDE/UML metamodel package in Figure 33 and establish the dependency relationships between the TDE/UML metamodel and `Traceability Repository`, `Model Changes` and `Prioritization`. The `Package Traceability Repository` is responsible for building and maintaining traceability links between test procedures and model elements of the original version of a TDE/UML model. The `package Model Changes` compares the differences between two versions of a TDE/UML model, including use case diagram changes, activity diagram changes and class diagram changes. The `package Classification` needs to interact with `Model changes` and `Traceability Repository` to categorize test artifacts. The `Package Prioritization` has to use the results of `Classification` and `Model Changes` and access information of

TDE/UML diagrams as well as prioritize the test artifacts of a system and then provide an overall test procedure sequence to users.

When performing test selection, our tool works by reading two versions of a TDE/UML model and a test suite from relevant XML files, and then it outputs test selection results into a text file. Then based on test selection results, the test prioritization part of our tool will read users' input data (i.e., pair-wise comparisons of use cases) from a text file and genetic algorithm parameters from a configuration file to perform prioritization. At the end, the result of test prioritization is outputted to a text file. The GUI part of our tool has not been implemented yet. However, a simple GUI could be easily implemented to locate models and test suite files, facilitating entering users' input data and genetic algorithm parameters.

Meanwhile, we also build a set of test cases to verify the implementation of classification rules defined in section 6. Each test case checks a classification rule based on a simple example.

Figure 33: Packages of Test Selection and Prioritization Tool

8.2 Genetic Algorithm Tool

As described in Section 7.2.3, the SPEA2 algorithm is applied when users want to achieve multi-objective during test case prioritization. The SPEA2 algorithm is implemented by extending an open source framework ECJ [43] that supports genetic algorithms. ECJ is an Evolutionary Computation and Genetic Programming System written in Java. The reason to choose this framework is that it is a highly flexible framework and provides a default implementation of the SEPA2 algorithm. To integrate the ECJ framework into our test case prioritization approach, we only need to make certain extensions of the framework to include a new genotype, individual fitness functions and an individual evaluator.

The class diagram of the extensions of the ECJ framework is shown in Figure 34. ECJ provides simple representations for individuals and genes: classes `GeneVectorIndividual` and `VectorGene`. Each `GeneVectorIndividual` consists of vectors of `VectorGenes`. Since each individual in our context is a sequence of test cases, we choose to extend `GeneVectorIndividual` as `TestCaseVectorIndividual` to represent individuals and extend `VectorGene` as `TestCaseGene` to represent each test case. To support the crossover and mutation operator suggested in Section 7.2.3.2.3, we have to override the default implementation provided in the class `GeneVectorIndividual`.

The `SPEA2MultiObjectiveFitness` of the ECJ framework implements basic multi-objective fitness mechanisms used in the SPEA2 genetic algorithm. Therefore, the

`SPEA2MultiObjectiveFitness` is extended by the Class `TdeMultiObjectiveFitness` to handle the SPEA2 fitness assignment in our case.

Finally, according to the way that the ECJ framework evaluates individuals in a population, we have to extend the class `Problem` and implement an interface `SimpleProblemForm` in the class `TestCasePrioritizationProblem` to handle the evaluation of individuals for our purpose.

The rest of the implementation of the genetic algorithm remains as it was in ECJ, and more information can be found on the ECJ project website.

Figure 34: Extensions of the ECJ framework

9 CASE STUDY

In this case study, we apply our regression testing method to examine its effectiveness. The case study is based on a system developed by a team of students. The documentation of this system includes the textual use case description, the design model and the source code (in Java). This case study system is an Automated Teller Machine (ATM) system and has been modified to different version of systems. To apply our regression testing methodology, we build a TDE/UML test model of the ATM system. The functionalities of the system are modeled by use case diagrams. Each use case diagram is further described by an activity diagram. Then, the categories and choices, which specify the test data of the system, are specified in the class diagram. The ATM test model contains 12 use cases, 12 activity diagrams, 68 activities, 8 categories and 26 choices.

Figure 35: Use case diagram of the ATM system

As shown in Figure 35, the main functions provided by this ATM system, as modeled by a use case diagram, including use case `ATMStartUp`, `ATMShutOff`, `DoTransaction`. These three use cases are used as initial use cases to generate test suites. The use cases `ATMStartUp` and `ATMShutOff` are used to specify the start up and shut down procedures of the ATM system. The use case `DoTransaction` describes how the ATM system performs transactions. It includes use cases `Card Insert`, `Get Pin`, `Perform Transactions` and `AskingDoAnother`. The inheritance relationship between use cases is not supported by TDE/UML and therefore the inheritance relationships between `Perform Transaction` and `Deposit`, `Withdrawal`, `Transfer`, `Inquiry` in the original system design model are changed into includes relationships. In addition, in the original use case diagram, use case `DoTransaction` is extended by use case `Cancel` to capture an exception scenario whereby the user presses the cancel button to cancel a transition. However, the user might press the cancel button at any point during a transaction. Thus, we need to model this exception scenario when we describe the activity diagrams of use case `Perform Transactions`, `Transfer`, `Withdrawal` and `Deposit`. In order to reuse the flow of events of use case `Cancel` in the activity diagrams of these four use cases, we made the following modification: use case `Cancel` extends these four use cases, rather than extending use case `DoTransaction`.

This chapter is organized as follows. Section 9.1 describes the result of the test generation. Section 9.2 describes the changes of the system requirements and the result of our test selection approach, based on those changes. Section 9.3 presents the result of our test prioritization approach.

9.1 Test Generation

Before performing test generation, we have to decide on the configuration information of the test generator (see Section 3.1.3.4). Specifically, we decide to choose the `all-path` algorithm as the `coverage-algorithm`. By applying the `all-path` algorithm, the test generator is able to choose all possible paths to create test cases, so that most functions or combinations of functions are tested. The `choice-per-path` algorithm is chosen as the `choice-coverage` algorithm since all choices of a variable can be used in every generated path within this algorithm. Also, the test generator creates a relatively small test suite if it uses `choice-per-path` compared to other `choice-coverage` algorithms, such as `exhaustive` and `pair-wise`. For the other configuration options, we use the default values.

Because the use case `ATMStartup`, `ATMShutoff` and `DoTransaction` are use cases that are triggered by users, we choose them as the initial use cases to generate test suites.

Based on the test model and the above settings, the test generator creates a huge test suite for use case `DoTransaction`, containing 728 test cases and 1689 test procedures. In order to reduce the size of the test suite, we add a constraint to the activity diagram of use case `DoTransaction`: the second transaction is triggered only if the first transaction is a withdrawal. This is reasonable because the usage of constraints in TDE/UML is to limit the number of test procedures generated for a system, and adding this constraint also conforms to how users would often operate an ATM system.

Table 10 shows the test generation result according to the test model with the added constraint, and to the test generator settings. Since the most important functionality of the ATM is described by the use case `DoTransaction`, the test suite generated for this use case contains 364 test cases and 756 test procedures in total. The other two test suites of use case `ATMStartup` and `ATMShutoff` only have one test case and one test procedure respectively due to their simple flow of events.

Tested/Main Use case	Total number of test cases	Total number of test procedures
ATMStartup	1	1
ATMShutOff	1	1
DoTransaction	364	756

Table 10: Test Generation result

9.2 Test Selection

After test generation, three changes are performed to the test model according to the changes of user requirements. These requirement changes are described as follows:

- First change: The first requirement change is about the number of times that a user could enter an incorrect PIN number. In the original requirement, there is no limit to that number. In the new version a user has only three attempts to enter a valid PIN before their card is retained by the system.
- Second change: The second requirement change imposes some extra restrictions on “savings” accounts. In the original system, savings and chequing accounts are identical in terms of which transactions can be performed on them. In the new version, a user cannot withdraw money directly from a savings account.

- Third change: The third requirement change has to do with the cash dispenser. The dispenser in the original system only holds twenty dollar bills. The new dispenser will be able to distribute both five and twenty dollar bills.

According to the requirement changes, we modify the relevant parts of the test model, which are described as follows:

- First change: The way of handling the invalid PIN number has been changed in the new version of the test model. The branch that handles the incorrect PIN number in the activity diagram of use case `Get Pin` has been deleted. Two new branches have been added to that activity diagram. One of them is responsible for dealing with the scenario that results when a user enters the incorrect PIN number less than three times, while the other one handles the situation that results when a user tries the invalid PIN more than three times. Table 11 shows the changes detected in this scenario, as well as the change rules and classification rules triggered by those changes. As shown in the third row and the ninth row in Table 11, a deleted activity and a changed decision are detected. Test procedures that exercise the deleted activity are obsolete (Classification Rule 4). Test procedures that exercise the changed decision node are retestable (Classification Rule 17). The other changes in Table 11 do not trigger any classification rules for two reasons: (i) The two deleted transitions are linked to a deleted activity. The impacts of these two deleted transition are indirectly accounted for by the impact of the deleted activity (Classification Rule 4). Therefore, they do not trigger any classification rule; (ii) For the other added elements (i.e. added activities, added

decisions, added transitions, added notes in activity diagrams and added note connections), because they are added in new paths, they do not have any impact on the existing test procedures (paths), thus do not trigger any classification rule.

Table 11: Detected changes in the first change

Changed Elements	Amount	Change Rules	Classification Rules
Deleted Transition	2	Change Rule 16	
Deleted Activity	1	Change Rule 9	Classification Rule 4
Added Activity	13	Change Rule 10	
Added Decision	2	Change Rule 10	
Added Start node	2	Change Rule 10	
Added Stop node	3	Change Rule 10	
Added Transition	18	Change Rule 17	
Changed Decision	1	Change Rule 14	Classification Rule 17
Added Notes in Activity diagrams	2	Change Rule 20	
Added Note Connections	2	Change Rule 22	

- Second change: A user takes money from their savings account. In the new version of the system, this withdrawal from the savings account is banned. This requirement change is translated into the change of a guard condition of a transition in the activity diagram of use case `PerformTransaction`. By changing this guard condition, the flows of events in the activity diagram of use case `Withdrawal` is only triggered when the account type is chequing. As shown in Table 12, only a changed transition is detected in this scenario. Test procedures that exercise the changed transition and do not satisfy the new guard condition are classified as obsolete (Classification Rule 11). Test procedures that exercise the changed transition and satisfy the new guard condition are classified as retestable (Classification Rule 21).

Table 12: Detected Changes in the second change

Changed Elements	Amount	Change Rules	Classification Rules
Changed Transition	1	Change Rule 18	Classification Rule 11, Classification Rule 21,

- Third change: A user successfully withdraws money from their chequing accounts. In order to dispense both twenty and five dollar bills in the new cash dispenser, an activity that represents the behavior of dispensing cash is deleted from the activity diagram of use case Withdrawal. The deleted activity is replaced by four new activities, which are used to calculate the number of twenty and five dollars bills needed, and to dispense twenty and five dollars bills, respectively. Table 13 shows the changes detected in this scenario, as well as the change rules and classification rules triggered by those changes. As shown in the third row in Table 13, a deleted activity is detected in this scenario. Thus test procedures that exercise the deleted activity are obsolete according to Classification Rule 4. Again, the deleted transition is linked to a deleted activity and its impact is indirectly accounted for by the deleted activity (Classification Rule 4). Thus it does not trigger any classification rule. For the other added activities and added transitions, although they appear on an existing path (i.e., in version one), we stop examining the detected changed elements of a test procedure once a classification rule that results in obsolete test procedures is triggered, thus those added activities and transitions do not trigger any classification rule.

Table 13: Detected Changes in the third change

Changed Elements	Amount	Change Rules	Classification Rules
Deleted Transition	1	Change Rule 16	
Added Transition	4	Change Rule 17	

Deleted Activity	1	Change Rule 9	Classification Rule 4
Added Activity	4	Change Rule 10	

The test selection result once the three changes have been performed to the model is summarized in Table 14. Among the three test suites for the system, only the test suite of use case DoTransaction is classified as retestable; the other two are reusable. 305 (83.7%) test cases in the test suite of use case DoTransaction are identified as obsolete and the remaining 59 (16.3 %) test cases in that test suite need to be retested: none is reusable. For the classification result at the test procedure level, 638 (84.4%) test procedures in the test suite of use case DoTransaction are classified as obsolete. The remaining 118 (15.6 %) test procedures of that test suite are retestable. The reason we obtained so many obsolete test procedures is that the changes in this case study have very huge impacts on test selection even though they are very simple changes.

For test suites of use case ATM Startup and ATM Shutoff, each of them contains one reusable test case/test procedure because no change was made to the activity diagrams of these two use cases.

Regression Test Selection		Amount	Obsolete	Retestable	Reusable
	TestSuite	3	0	1	2
	TestCases (DoTransaction)	364	305 (83.7%)	59 (16.3%)	0
	TestProcedures (DoTransaction)	756	638 (84.4%)	118 (15.6%)	0
	TestCase (ATM Startup)	1	0	0	1
	TestProcedure (ATM Startup)	1	0	0	1
	TestCase (ATM Shutoff)	1	0	0	1
	TestProcedure (ATM Shutoff)	1	0	0	1

Table 14: Selection result

9.3 Test Prioritization

After obtaining retestable test suites, test cases and test procedures, we apply our prioritization method to order these retestable artifacts. Because there is only one retestable test suite identified in our case study, we don't need to apply the AHP method (see Section 7.1) to prioritize test suites. For prioritization at the test case level, test cases of a test suite could be prioritized either by the single-objective approach (see Section 7.2.4) or using the multi-objective approach (see Section 7.2.3). The single-objective approach only considers the expected risk value of test cases and then prioritizes test cases in the descending order of their expected risk value. The multi-objective approach considers both the expected risk value and the (expected) execution time of test procedures and tries to find a balance between these two factors. In both cases, once test cases have been prioritized, test procedures for each test case are prioritized in the descending order of their probabilities (see Section 7.3). Then, according to user specified values x and y (see Section 7.4), a stop point for a test suite is computed (for single objective optimization). Finally, the overall test procedure sequence is obtained by combining the result of both test case prioritization, the result of test procedure prioritization and the stop point.

Since we only have one retestable test suite, we focus on analyzing and presenting the results of test case prioritization. Section 9.3.1 and Section 9.3.2 presents the results of the single- and multi-objective test case prioritization approaches, respectively. Then we compare the effectiveness of mutant detection abilities for test sequences generated by our test prioritization approaches against randomly generated test sequences in Section 9.3.3.

Recall that there are three ways to compute the risk of a test case: retestable control flow entities, retestable definitions and retestable uses. The risk value is further used to get the expected risk of a retestable test case. In this case study, there are 229 control flow entities, 17 definitions, 23 uses detected in the second version model. Table 15 summaries the changed model elements in this case study. Among these changed model elements, only one changed transition and one changed decision are on the paths exercised by the retestable test cases. According to the definitions of retestable control flow entities in Section 7.2.1.2.1, they are identified as retestable transition/decision. Therefore we only have two retestable control flow entities. On the other hand, since there is no changed category, which results in retestable data flow entities, we don't have any retestable definitions or uses. As a result, if the coverage of retestable definitions/uses is used as risk values for test cases, all the test cases are assigned a risk value of zero. Thus we only choose the coverage of retestable control flow entities to get the risk value of test cases in this case study.

Table 15: Changed Elements

Elements	Added	Changed	Deleted	Retestable
Transition	22	1	3	1
Activity	17	0	2	0
StopState	3	0	0	0
Decision	2	1	0	1
NoteInActivityDiagram	2	0	0	0
Note Connection	2	0	0	0
StartState	2	0	0	0

9.3.1 Single-Objective Approach Result

The prioritization result of the single-objective approach shown as “single-objective test sequence” is presented in Figure 36 by plotting the cumulative *nERisk* versus the percentage of executed test cases. The area under the curve represents the rate at which a

test case sequence increases the cumulative *nERisk*. Meanwhile the single-objective prioritization result is compared with the results of cumulative *nERisk* of randomly generated test case sequences. Since the single-objective approach only produces one test case sequence¹⁸, generating 100 random test case sequences is enough for comparison purposes. Figure 36 shows the worst, the best and the average increasing rate of cumulative *nERisk* among these 100 random test case sequences, shown as “worst of random test sequences”, “best of random test sequences” and “average of random test sequences”, respectively. As shown in Figure 36, the “single-objective test sequence” increases its cumulative *nERisk* quickly. By only executing the first 50% of its test cases, the “single-objective test sequence” achieves more than 80% of the cumulative *nERisk*. The front part of the curve of “single-objective test sequence” is steeper than the later part, because the single-objective prioritization approach always places test cases with higher (cumulative) *nERisk* in the front of the test case sequence. Figure 36 also shows that the results of random test case sequences are always under the curve of “single-objective test sequence”, which means that “single-objective test sequence” outperforms the result of random test case sequences in terms of the rate of increasing cumulative *nERisk*.

¹⁸ There is a chance of randomness in the generation of the single-objective sequence: when several test cases have the same additional risk, one is selected randomly. In this section, since we consider risk, all the possibly different sequences resulting from this random choice are equivalent, and we only report on one single-objective sequence. In our case study, we did not see such randomness: executing the single-objective optimization several times leads to the same sequence.

Figure 36: result of single objective prioritization approach

As described in Section 7.2.4, with the aim of effectively increasing the cumulative $nERisk$, users are asked to specify two values of x and y to determine the stop point of the execution of the test cases of a test suite: the stop point indicates a test case in the sequence after which the increase of cumulative $nERisk$ is not deemed worth the additional cost. We illustrate this principle here though we only have one retestable test suite. Table 16 shows a set of user defined values used in this case study. It also reports on the stop points generated according to these values. For instance, if the user specifies $x=0.2$ and $y=0.2$ (i.e, new test cases should at least increase the cumulative $nERisk$ by 20% for a cost smaller than 20% of the total cost to be included), then the results suggest that the execution of the test suite terminates after executing the 18th test case. Figure 36

shows that the cumulative *nERisk* of the 18th test case (i.e., 30.5% of the set of test cases) is 63.59%. The cumulative *nERisk* of the 29th test case (i.e., 49.1% of the set of test cases), which is the last test case of the next 20% of the test cases after the 18th test case (recall that $x=0.2$), is 83.06%: these are illustrated by the two vertical bars in Figure 36. Therefore, the difference of cumulative *nERisk* between these two points (i.e., 19.47) is smaller than 20% (recall that $y=0.2$), which is what we expected.

As shown in Table 16, if we keep the value of x the same, and decrease the value of y , the stop point is gradually moved from the beginning of the test sequence to the end, since a smaller value of y indicates that test cases after the stop point increase cumulative *nERisk* slower than those with a larger value of y . For example, taking 0.2 as the value of x , which represents 11 test cases in this case study, consider what happens when the value of y is decreased from 0.2 to 0.05. In this case, the stop point is moved from the 18th test case to the 38th test case. Similarly, if we keep the y value constant and increase the value of x , the stop point is moved to the end of the test sequence. Given a larger value of x , the test cases after the stop point determined by that x have to increase slower than those with a smaller value of x . For example, if we take 0.15 as the value of y , and then increase the value of x from 0.2 to 0.3, the stop point is changed from the 26th test case to the 29th test case. The above observation is expected because test cases in the back part of the test sequence contribute less to the cumulative *nERisk* than test cases in the front part: as a result, the cumulative normalized eRisk curve is convex.

User defined values	The stop points
x=0.2, y=0.2	the 18 th test case
x=0.2, y=0.15	the 26 th test case
x=0.2, y=0.1	the 32 nd test case
x=0.2, y=0.05	the 38 th test case
x=0.3, y=0.2	the 25 th test case
x=0.3, y=0.15	the 29 th test case
x=0.3, y=0.1	the 33 rd test case
x=0.3, y=0.05	the 40 th test case

Table 16: User defined values and stop points

9.3.2 Multi-Objective Approach Result

For the multi-objective test case prioritization approach, two optimization schemes are proposed to satisfy users' different testing requirements. Optimization scheme one is used when the time budget is the major concern while optimization scheme two is applied when users want to reach a certain threshold of the cumulative *nERisk*.

The execution time is considered in this approach, and is measured on version one of the system. We measure the execution time of the retestable test procedures on a Pentium 4 3.2GHz processor with 1 GB of RAM: The start and stop times are measured by method `System.currentTimeMillis()` in Java. The execution time of a retestable test case is the summation of the execution times of the retestable test procedures within that test case. Figure 37 shows the distribution of retestable test cases' execution time. Most retestable test cases need less than 250 milliseconds except one that requires more than 400 milliseconds. The test case that needs more execution time than others triggers a longer scenario than the other use cases: the user performs two transactions in sequence; and it contains five retestable test procedures (a larger number than other test cases).

Figure 37: The distribution of retestable test cases' execution time (ms)

We discuss the result of optimization scheme one and two in Sections 9.3.2.1 and 9.3.2.2, respectively.

9.3.2.1 Optimization Scheme One

Optimization scheme one is meant to help users who might not have enough time to run the whole test suite. Thus, users are asked to provide a time budget for running the retestable test cases of a test suite. Once the time budget is met, the test suite stops executing. For the time budgets used in this case study, we choose 50%, 60% and 70% of the total execution time because they are reasonable time budgets. Thus, given 5484 time units as the execution time for all the retestable test cases, we use 2742, 3290 and 3838 time units as the time budgets. Because the results of applying these three time budgets are similar, we only present and discuss the results of using 2742 time units as the time budget (i.e., 50% of the execution of the whole test suite).

Table 17 shows the results of the genetic algorithm at 0, 200, 400, 800, 1200, 1600, 2000 generations when the optimization scheme one is applied. This table also reports how long the SPEA2 algorithm takes to reach those generations (i.e. 26 secs for 200 generations and 4.2 mins for 1200 generations) and the number of non-dominated solutions (in the archive) generated at a specific generation. Recall that in optimization scheme one (section 7.2.3.2.5), two fitness functions are defined: f_1 (the rate of increasing the cumulative $nERisk$ value) and f_2 (the cumulative $nERisk$ Value). We also report the worst, best, and average values of f_1 and f_2 at different generations in Table 17.

Table 17: Summary of results for optimization scheme one

Generations	Execution time	Number of solutions
200	26 secs	2
400	58 secs	3
800	2.39 mins	3
1200	4.2 mins	3
1600	6.74 mins	4
2000	10.34 mins	4

Generations	Average of f_1	Best of f_1	Worst of f_1
200	1313.057	1313.79	1310.125
400	1316.422	1317.159	1315.205
800	1315.823	1317.347	1313.534
1200	1315.621	1317.347	1313.716
1600	1315.828	1317.363	1313.734
2000	1315.688	1317.363	1313.734

Generations	Average of f_2	Best of f_2	Worst of f_2
200	0.808196808	0.808196	0.808196796
400	0.813237309	0.813237	0.813237228
800	0.813237351	0.813237	0.813237228
1200	0.813237369	0.813237	0.813237228
1600	0.813237357	0.813237	0.813237228
2000	0.813237364	0.813237	0.813237228

As shown in Table 17, the SPEA2 algorithm continues to improve the fitness value of f_1 until it reaches 800 generations (after that we observe negligible improvements) and the

fitness value of f_2 until it reaches 400 generations. The improvement of the best value of f_1 or f_2 tends to level off after 800 generations or 400 generations. This result suggests that 800 generations are enough to be a practical stop point.

At 800 generations, the SPEA2 algorithm provides four non-dominated test sequences in total. Figure 38 shows the pairs of values (f_1 , f_2) for these test sequences: The x-axis is the value of f_1 and the y-axis is the value of f_2 . As shown in Figure 38, point A represents a test sequence that performs best in f_1 but worst in f_2 , and point B represents a test sequence that performs best value in f_2 but worst in f_1 . The other two points represents test sequences that have tradeoffs in terms of f_1 and f_2 between those two points. According to this figure, users can pick a solution according to their objectives. For example, if users are in favor of the rate of increasing cumulative *nERisk*, they can pick solutions close to point B. However, the differences between the values of (f_1 , f_2) pairs for these test sequences are very small: the difference of values of f_2 is around 0.00003% and the difference of values of f_1 is around 0.3%. In other words, the differences between these four solutions are not practically significant. If we plot curves for test sequences A and B, which are two extreme test sequences, in a graph where the x-axis is the percentage of execution time and the y-axis is the cumulative *nERisk* (Figure 39), we can hardly see differences. This means when the genetic algorithm optimizes one objective, it optimizes the other objective as well. Thus, perhaps only using one of the two objectives in optimization scheme one is sufficient.

Figure 38: Distribution of solutions for scheme one

Figure 39: curves of test sequence A and B

Additionally, our prioritization results are compared with randomly-generated test case sequences to show the effectiveness of our prioritization approach. From our prioritization results, we select three test sequences: one yields the best value of $f1$ but the worst of $f2$, shown as point A in Figure 38; one yields the best value of $f2$ but the worst of $f1$, shown as B in Figure 38; one shows a tradeoff between $f1$ and $f2$, shown as C in

Figure 38. In addition, since the SPEA2 algorithm evaluates 12837 chromosomes up to 800 generations in total, we generate the same number of random test sequences to have a fair comparison.

Since in optimization scheme one, there is a time budget to execute a test case sequence. We also apply this time budget to random test cases sequences so that each random test case sequence also has a stop point. The stops points of random test case sequence range from 21st test case to 33rd test case. On average, random test case sequences stop at their 26th test case. Table 18 shows the comparison between the results of the optimization scheme one and the results of random test case sequences. Table 18 reports the values of f_1 , the values of f_2 and stop points for the optimized test case sequences. After executing the first 35 test cases of each test case sequence, they all yield values of f_1 over 1313 and exceed 81% cumulative $nERisk$ within the time budget. On the contrary, the average value of f_1 for the random test case sequences is only around 625 (47% of the average f_1 value of optimized sequences) and even the best one is only about 979 (75% of the average f_1 value of optimized sequences). This means the random test case sequences increase their cumulative $nERisk$ value slowly. Within the time budget, they only are able to achieve around 62% cumulative $nERisk$ on average and 79% cumulative $nERisk$ at most, which are lower than the $nERisk$ values obtained from the optimized test case sequences. Thus, the statistics in Table 18 demonstrates that our prioritization approach outperforms random test case sequences either in terms of the rate of increasing cumulative $nERisk$ or the value of $nERisk$.

Table 18: optimization scheme one VS random test sequences

	value of f1	value of f2	Stop point (test cases)
Test sequence A	1317.363	0.8132372	35
Test sequence B	1313.7345	0.81323746	35
Test Sequence C	1315.8699	0.8132374	35

	Values for f1			Values for f2			Stop point		
	Average	Best	Worst	Average	Best	Worst	average	max	min
Random test sequences	625.38	979.93	0.29	0.62	0.79	0.45	26	33	21

9.3.2.2 Optimization Scheme Two

In some cases, users may not want to execute the whole test suite but instead, to exercise a certain percentage of the cumulative *nERisk* so that they can gain a certain confidence level about the tested functionality as quickly as possible. Optimization scheme two is designed for this purpose. In this scheme, users need to specify a threshold of the cumulative *nERisk* that they want to achieve. For the thresholds in this case study, we use 60%, 70% and 80% since they seem reasonable thresholds that users may want to achieve in practice. Similarly to the previous section, we only describe and analyze the result of applying a 70% threshold since the other values we tried lead to similar conclusions.

Similarly to the optimization scheme one, Table 19 shows the results of the genetic algorithm at 0, 200, 400, 800, 1200, 1600, 2000 generations when optimization scheme two is applied. This table also reports how long the SPEA2 algorithm takes to run those generations (i.e. 19 secs for 200 generations and 5.51 mins for 1200 generations) and the number of non-dominated solutions in the archive generated at each generation. Recall that in optimization scheme two (section 7.2.3.2.5), two fitness functions are used: f3 (the rate of reaching the threshold of the cumulative *nERisk*) and f4 (the overall execution time to reach the threshold). We also report the average, best and average values of f3

and f4 at different generations in Table 19. (Recall that lower values of f3 and f4 indicate better solutions.)

Table 19: summary of results of scheme two

Generations	Execution time	Number of solutions
200	19 secs	1
400	51 secs	10
800	3.46 mins	11
1200	5.51 mins	15
1600	14.7 mins	17
2000	26.2 mins	17

Generations	Average of f3	Best of f3	Worst of f3
200	642.9038	642.9038	642.9038
400	628.3426	626.1748	630.0358
800	626.6505	623.9436	629.3674
1200	625.7725	623.6411	628.2678
1600	626.1138	623.5437	628.2351
2000	626.4927	623.5021	628.2351

Generations	Average of f4	Best of f4	Worst of f4
200	2012	2012	2012
400	2121.17	1967	2216
800	2111.925	1951	2200
1200	2130.357	1951	2248
1600	2109.593	1951	2215
2000	2095.796	1951	2247

As shown in Table 19, the SPEA2 algorithm continues to improve the fitness value of f3 until it reaches 1200 generations and the fitness value of f4 until it reaches 400 generations. The improvement of the best value of f3 or f4 tends to level off after 1200 generations or 800 generations. This result suggests that 1200 generations are enough to be a practical stop point. Notice that initial best values for f3 and f4 (e.g., at 200 generations) are better than during later generations (e.g., 800 generations). This indicates that better tradeoffs can be found later and that during early generations best solutions tend to favor one fitness function over the other more so than during later generations.

At 1200 generations, the SPEA2 algorithm provides 15 non-dominated test sequences in total. Figure 40 shows the pairs of values (f_3 , f_4) for these test sequences: The x-axis is the value of f_3 and the y-axis is the value of f_4 . As shown in Figure 40, point A and B represents two test sequences that perform best in one fitness function or in the other: point A for f_3 and B for point f_4 . The other points distributed between A and B show different tradeoffs between f_3 and f_4 . According to this figure, users can pick a solution according to their objectives. For example, if users are in favor of spending less time to reach the given threshold, they can pick solutions close to point B. However, the difference between the worst value of f_3 (point B) and the best value of f_3 (point A) is only around 0.74%, which means the rates of increasing cumulative *nERisk* of these test sequences are very close. In this case, we suggest that users pick test sequence B as the solution for the optimization scheme two since it spends the least time to reach the threshold. However, as opposed to the results of scheme 1, this doesn't mean we could simply prioritize test cases according to their execution time, because a test case with less execution time doesn't mean it also contributes a larger value to the cumulative *nERisk*. (The difference in f_4 's value between A and B is 13%.) Therefore it's still necessary to apply the multi-objective genetic algorithm in this situation to obtain test case sequences that spend less execution time to reach the threshold at a fast rate.

Figure 40: Distributions of solutions for scheme two

Similarly to optimization scheme one, the results of optimization scheme two are also compared with randomly-generated test case sequences. From the solutions of our prioritization results, we select three test sequences: one yields the best value for f_3 but the worst for f_4 (point A in Figure 40); one yields the best value for f_4 but the worst for f_3 (point B in Figure 40); one is a tradeoff between A and B (point C in Figure 40). In addition, since the SPEA2 algorithm evaluates 32398 chromosomes up to 1200 generations when the optimization scheme two is applied, we generate the same number of random test sequences to have a fair comparison.

In optimization scheme two, a user defined threshold of cumulative $nERisk$ is used so that a test case sequence would stop once it reaches the threshold. We also apply this threshold to random test cases sequences so that each random test case sequence also has a stop point. The stops points of random test case sequence range from the 32nd test case to the 45th test case. On average, random test case sequences stop at their 39th test case. Table 20 shows the comparison between the results of optimization scheme two and the

results of random test case sequences. Table 20 reports the values of f_3 , the values of f_4 and stop points for test case sequences A, B and C. These test case sequences spend at most around 2248 milliseconds to reach the threshold. The values of f_3 for test case sequences are lower than 629. On the contrary, the random test case sequences have to spend at least 2781 milliseconds to reach the threshold (23% more). And the best value of f_3 of random test sequences is only around 954, which is larger than the worst case of prioritized test cases sequences (51% larger). Thus, the statistics in Table 20 demonstrates that our prioritization approach outperforms the random test case sequences either in terms of the rate of increasing cumulative $nERisk$ or the overall execution time.

Table 20: Optimization scheme two VS random test sequences

	value of f_3	value of f_4	Stop point
Test sequence A	623.6083	2248.0	29
Test sequence B	628.235	1951.0	26
Test Sequence C	625.4275	2138.0	27

	Values for f_3			Values for f_4			Stop point		
	Average	Best	worst	Average	Best	Worst	average	max	min
Random test sequences	1374.276	954.485	1864.534	3906.0348	2781.0	4437.0	39	45	32

9.3.3 Effectiveness

In this section, we examine the effectiveness of our prioritization approach in terms of mutant detection ability. Since the result of our multi-objective prioritization approach is a set of test procedure sequences, we compare them to check whether they are equivalent in terms of mutant detection. Then, in order to demonstrate the benefits of our prioritization method, we compare the test procedure sequences generated by our prioritization method with random test procedure sequences, in terms of mutant detection ability.

For mutant seeding, a java mutation system, MuJava [44], is applied to automatically and systematically generate mutants for the source code of the ATM system. To avoid bias, we apply all the traditional and class-level mutant operators provided by MuJava. Since MuJava only allow users to choose which java classes to be mutated, we only select the modified java classes to insert mutants since faults are most likely to be inserted into the changed code after modifications have been made. The total number of mutants that are inserted into changed classes is 955. Since most code changes are made in the body of methods of java classes, to be more precise, we go on to select mutants that are inserted into changed methods and mutants that are inserted into changed statements, resulting in 461 and 22 mutants, respectively. However, when we execute our test procedure sequences to kill mutants that are inserted into changed statements, we find that by only executing the first 10 test procedures in sequences is sufficient to kill all those mutants. Since our sequences have longer sizes than 10, this would not lead to interesting, useful comparison results. Thus, we decide to use the mutants that are inserted into changed methods in our case study. Additionally, there are 154 out of 461 mutants that are killed by at least one test procedure. We only focus on these killed mutants instead of all seeded mutants.

To study the effectiveness of prioritization techniques, S. Elbaum et al. [45] introduced the cost-cognizant APFD ($APFD_c$) metric, which accounts for the test cost and faults severities. The $APFD_c$ score of a test case sequence represents its rate of “units-of-fault-severity-detected-per-unit-test-cost”. We adapt the $APFD_c$ metric in our context to measure the rate of “units-of-fault-severity-detected-per-unit-test-cost” for test procedure

sequences. Our $APFD_c$ metric is the same as the original one except that we use the term “test procedure” while the original one uses the term “test case”. It is shown as follows:

$$APFD_C = \frac{\sum_{i=1}^m (f(i) * (\sum_{j=TF_i}^n t(j) - \frac{1}{2}t(TF_i)))}{\sum_{i=1}^n t(i) * \sum_{i=1}^m f(i)}.$$

In the above metric, m is the number of faults detected, n is the number of test procedures that have been executed, TF_i is the first test procedure in a test procedure sequence that detects mutant i , $t(j)$ is the percentage of execution time of test procedure j and $f(i)$ is the severity of mutant i . The larger $APFD_c$ score indicates a better result.

To use the $APFD_c$ metric, we need a notion of severity of fault/mutant. In our context, we define the severity of a mutant based on the test cases and test procedures that execute the statement where the mutant is inserted. Given a mutant and a statement where the mutant is inserted, we observe the following three properties about the association between the severity of the mutant and the test cases/test procedures that execute the mutated statement: 1) If the number of test cases that execute the mutated statement increases, this means the mutant has a higher probability to be killed, which decreases the severity of the mutant; 2) Within a test case that executes the mutated statement, if the number of test procedures that execute the mutant increases, the mutant has a higher probability to be killed too, which decreases the severity of the mutant; 3) If the expected risk of one of the test cases that execute the mutated statement increases, the severity of the mutant also increases. Based on these three properties, we defined the severity of mutant i as follows:

$$f(i) = \frac{\sum_{j=1}^{p_i} e(TC_i(j))}{\sum_{j=1}^{p_i} num(TC_i(j))}$$

In the above metric, let $TC_i(1), TC_i(2), \dots, TC_i(p_i)$ be the set of test cases that execute the statement where mutant i is inserted; p_i is the size of this set; $e(TC_i(j))$ is the *nERisk* of a test case j ; $num(TC_i(j))$ is the number of test procedures in test case j that executes the mutated statement.

This function has the properties we discussed earlier. If we keep the number of test cases and the number of test procedures of each test case constant, and increase the expected risk of a test case, then the denominator of the formulae is not changed and the numerator is increased, thus the value of the metric is increased too. If we keep the expected risks of test cases and the number of test cases the same, and increase the number of test procedures within a test case, the numerator of the formulae is not changed and the denominator is increased, thus the value of the metric is decreased. If we keep the expected risk value of test cases and the number of test procedures of each test case the same, and add one more test case, both the numerator and denominator of the metric are increased. However, a *nERisk* value is always lower than 1 and the number of test procedures of a test case is always equal to or larger than 1. In other words, the addition to the denominator is larger than the addition to the numerator, which results in the reduction of the value of the metric. As a result, the metric defined above shows the properties that we expect.

In the following subsections, we compare the APFD_c scores of test procedure sequences obtained from our approach with randomly generated test procedure sequences.

9.3.3.1 Result of Single-objective approach VS random test sequences

Since the single-objective approach only produces one optimal result¹⁹, generating 100 random test procedure sequences is enough for comparison purposes. Recall that for the single objective approach, a stop point is computed according to two user defined values x and y : we use different value pairs and calculate the number of test procedures they require to be executed (i.e., the corresponding stop points). For each pair (x,y) , defining a stop point value, we therefore build random sequences of the same size (using the stop point for the pair). Since the stop point identifies a number of test cases, different stop point values (and therefore different sequences of test cases) will lead to different number of test procedures. Table 21 therefore shows the different pairs of values (x,y) we have selected (column 1), the corresponding stop points and number of test procedures (columns 2 and 3), and APFD_c scores of the single objective optimization result and random sequences (best, worst, average, and standard deviation values).

As shown in Table 21, the APFD_c scores of the single-objective approach increase when we execute more test procedures. This is to be expected: we execute more test procedures so we potentially detect more mutants.

¹⁹ Theoretically the single objective optimization can produce several equivalent results (in terms of increased expected risk) since when several test cases increase the expected risk by the same value, they are ranked randomly. During the case study, we therefore ran the single objective several times and did not observe random variations: the optimization never had to randomly rank test cases.

The best APFD_c scores of random test procedure sequences outperform all scores of the single-objective approach. If the stop point is smaller than 73, the APFD_c scores of the single objective approach are smaller than the average scores of random test procedure sequences. If the stop point is equal to or greater than 73, the APFD_c scores of the single objective approach are larger than the average scores of random test procedure sequences. We also study the distribution of APFD_c scores of different sets of random test procedure sequences. As shown in Figure 41 for 65 test procedures ($x=0.3, y=0.2$), most of the random test procedure sequences (77 out of 100) have APFD_c scores within range $[0.60, 0.65]$. Since the APFD_c score of the single objective approach is 0.688, it outperforms most of random test procedure sequences. When the number of executed test procedures (stop point) is larger than 65, we obtain similar results. However, if the number of executed test procedures is 53, the result of single-objective approach is not better than the corresponding random test procedure sequences. This suggests that users have to execute a certain number of test procedures (i.e. greater than 65 in our case study) otherwise they can not get benefit from the single objective approach. Moreover, even though users execute enough test procedures (i.e. more than 65), there are still a few of random test sequences that are better than the results of single-objective approach. This is because in our case study some test procedures have a very high capability to kill mutants. We studied the amount of mutants that a test procedure can kill. The result shows that 45 out of 118 (38%) test procedures have a very high capability to kill mutants: each of these 45 test procedures is able to kill more than 90 out of 154 mutants (58%). We have such highly effective test procedures because there is a series of nested if-else statements in changed methods and MuJava seeded a large number of mutants into

the conditions of these statements. We observe that a test procedure that exercises those statements is able to kill a large number of mutants. And it happens that those statements execute when main functionalities (i.e. use case perform transaction) are triggered. Therefore, random sequences have a high chance of selecting those highly effective test procedures early in the sequence, thereby yielding a superior APFD_c score.

Table 21: APFD_c scores for the single-objective approach and 100 random test sequences

User defined values	Number of executed test cases (up to)	Number of executed test procedures	APFD _c scores				
			Single-Objective optimization	Random			
				Worst	Avg	Best	StdDev
x=0.2, y=0.2	the 18 th	53	0.603	0.590	0.697	0.972	0.121
x=0.3, y=0.2	the 25 th	65	0.688	0.603	0.710	0.976	0.128
x=0.2, y=0.15	the 26 th	66	0.692	0.604	0.711	0.976	0.128
x=0.3, y=0.15	the 29 th	73	0.730	0.608	0.716	0.978	0.131
x=0.2 y=0.1	the 32 nd	76	0.744	0.610	0.718	0.979	0.132
x=0.3, y=0.1	the 33 rd	77	0.745	0.611	0.719	0.980	0.133
x=0.2, y=0.05	the 38 th	87	0.768	0.616	0.725	0.982	0.136
x=0.3, y =0.05	the 40 th	93	0.781	0.619	0.729	0.983	0.137

Figure 41: Distribution of APFD_c scores for random test sequences with 65 executed test procedures

9.3.3.2 Results of Optimization scheme one VS random test sequences

Table 22 shows the APFD_c scores for the results of optimization scheme one (multi-objective approach) and random test procedure sequences. Since the SPEA2 algorithm evaluates 12837 chromosomes up to 800 generations we also generate 12837 random test procedure sequences to have a fair comparison. Since the four non-dominated solutions of optimization scheme one have three different stop points for test procedure sequences (two of them have the same stop point), we apply these stop points to create three sets of random test sequences and then compute their APFD_c scores.

Table 22 (a) shows the APFD_c scores of the (four) results of the optimization: they are very good (APFD_c value greater than 0.920) and quite close from one another (small variation): The difference between the best and the worst score is less than 1%. This means the test procedure sequences obtained from optimization scheme one are comparable to each other in terms of mutant detection effectiveness.

Table 22 (b) shows the best, worst, average, and standard deviation APFD_c scores of random test procedure sequences with different numbers (stop points) of executed test procedures. It also shows the APFD_c value of the corresponding (same stop point) optimization solutions. (Note that two optimized solutions have 70 test procedures.) The APFD_c scores of the optimized solutions are smaller than the best scores of random test sequences but they are far better than the average scores of those random test sequences. Additionally, the optimized solutions outperform most of random test procedure sequence. For example, with length 71, the optimized solution outperforms 91.3% of the random test procedure sequences. Because the optimized solutions execute test procedures within test cases that have high expected risk values but have less execution

time, they are able to effectively kill severe mutants. This is why they outperform most of random test procedure sequences. Still, there are a few random test procedure sequences outperform the result of ours. Similar to what we discussed in the previous section, this is because these random test procedure sequences execute test procedures that have a high capability to kill mutants and are easily selected early at random.

Table 22: APFD_c scores for optimization scheme one and 12837 random test sequences

APFD _c scores	Worst	Avg	Best	StdDev
optimization scheme one	0.925	0.928	0.932	0.000045

(a)

Number of executed test procedures	Random test procedure sequences (APFD _c scores)				APFD _c scores solutions of optimization scheme one	Percentage of times optimization scheme one performs better than random
	Worst	Avg	Best	StdDev		
70	0.570	0.700	0.986	0.124	0.927	91.4%
					0.925	91.0%
71	0.542	0.687	0.987	0.117	0.928	91.3%
72	0.544	0.689	0.987	0.118	0.930	91.7%

(b)

9.3.3.3 Results of Optimization scheme two VS random test sequences

Table 23 shows the APFD_c scores for the results of optimization scheme two and random test procedure sequences. Since the SPEA2 algorithm evaluates 32398 chromosomes up to 1200 generations, we also generate 32398 random test procedure sequences to have a fair comparison. The 15 non-dominated solutions of optimization scheme two have ten different stop points, i.e., ten different lengths of test procedure sequences. We apply these stop points to create ten sets of random sequences of the corresponding lengths and then compute their APFD_c scores.

Table 23 (a) shows the APFD_c scores of the results of optimization scheme two. They are very good (over 0.919) and show very small variation: The variation between the best

and worst solutions is only around 1.1%. This means the test procedure sequences obtained from optimization scheme two have close ability to reveal mutants.

Table 23 (b) shows the best, worst, average, and standard deviation APFD_c scores of random test procedure sequences. We also compare those scores with the corresponding (i.e., same length) solutions of optimization scheme two. Similar to what we obtained for optimization scheme one, the APFD_c scores of optimized solutions are smaller than the best scores of random test sequences, but they are better than the average APFD_c scores of random test sequences. Again, the results show that all non-dominated solutions of optimization scheme two outperform most of random test sequences. For instance, for length 52, the optimized solution outperforms 93.5% of the random test procedure sequences. The reason is that optimization scheme two considers both the expected risk and time to prioritize test cases. This way, the results of optimization scheme two are able to effectively kill severe mutants. On the other hand, a few random test sequences perform better than optimized solutions. This is because some test procedures have high capability to kill mutants. If random test procedure sequences execute these high capability test procedures in the front of their test procedure orders, they will yield superior APFD_c scores.

Table 23: APFD_c scores for optimization scheme two and 32398 random test sequences

APFD _c scores	Worst	Avg	Best	StdDev
optimization scheme two	0.919	0.923	0.930	0.00007

(a)

Number of executed test procedures	Random test procedure sequences (APFDc scores)				APFDc scores solutions of optimization scheme two	Percentage of times optimization scheme one performs better than random
	Worst	Avg	Best	StdDev		
52	0.539	0.686	0.987	0.117	0.920	93.5%
53	0.542	0.687	0.987	0.117	0.919	93.2%
					0.923	93.7%
54	0.544	0.689	0.987	0.118	0.919	93.0%
55	0.547	0.690	0.987	0.119	0.920	93.5%
56	0.549	0.691	0.988	0.119	0.921	92.9%
57	0.551	0.692	0.988	0.120	0.924	93.2%
					0.921	92.7%
					0.922	92.8%
58	0.552	0.693	0.988	0.120	0.922	92.6%
59	0.553	0.694	0.988	0.121	0.923	92.6%
					0.923	92.6%
60	0.554	0.695	0.988	0.121	0.924	92.5%
62	0.558	0.697	0.989	0.122	0.927	92.7%
					0.930	93.2%

(b)

9.4 Conclusions of Case Study

Overall, the results of this case study indicate that our regression testing approach is effective at both test selection and prioritization. After test selection, the size of the regression test suite is significantly reduced. Only 118 test procedures out of 758 are selected to execute again. In terms of prioritization, the test case sequence obtained from the single objective test case prioritization approach improves the rate of increasing cumulative expected risk in comparison to randomly generated test case sequences. The multi-objective test case prioritization approach produces a set of tradeoffs test case sequences for both optimization schemes one and two. These tradeoffs test case sequences also improve the rate of increasing the cumulative expected risk value compared to random test case sequences. In addition, the test case sequences obtained from optimization scheme one are able to achieve a higher cumulative expected risk within the time budget than random sequences, and the test case sequences obtained from optimization scheme two spend less time to reach/exceed a user defined threshold of cumulative expected risk compared to randomly generated test case sequences.

The study of mutant detection effectiveness shows that the overall test procedure sequences obtained from our prioritization approach perform better than most randomly generated test procedure sequences. However, there is one exception for the single-objective approach. If user defined values x and y are not appropriately given (i.e $x=0.2$, $y=0.2$ in this case study), then the stop point of the single-objective approach is too short. In such case, the result of the single-objective approach doesn't execute enough test procedures and is not better than most randomly generated test procedure sequences. We also observed that the optimized solutions of the multi-objective approach have very slight variations in terms of APFD_c scores. This suggests that the optimized solutions of the multi-objective approach have close ability to reveal mutants. Conversely, randomly generated test procedure sequences have large differences between one another in terms of APFD_c scores. A few of them even perform better than the optimized solutions. However, the distribution of APFD_c scores of randomly generated test procedure sequences shows that a large percentage of them have worse performance than the optimized solutions. Additionally, some test procedures have a very high capability to kill mutants. Therefore, if random test procedure sequences first execute those high capability test procedures, they will have very good APFD_c scores. We observed when random sequences perform better than optimized ones, this is because of easy to kill mutants: 38% of the test procedures have a mutant score of 58%.

The limitations of this case study are:

- 1) The subject case study system is a student project that is small in size and which involves a few changes to the model.

- 2) Only one test suite is identified as a retestable test suite. Thus we don't have the chance to examine the results of test suite prioritization.
- 3) According to the requirement changes, most of the model changes are performed on control flow elements. We only choose the coverage of retestable control flow entities as the value of the risk of a test case in this case study. We did not have a chance to use and evaluate other risk measures (i.e., based on data flow).
- 4) The tradeoffs test case sequences of optimization scheme one in the multi-objective approach have very close fitness values in terms of both objectives. This suggests that we might only need one objective in this optimization scheme.
- 5) For this small ATM system, the time spent on prioritization is greater than the total test suite execution time. In this case, users could simply re-execute all retestable test procedures rather than spending time on prioritization. However, the purpose of the case study is to perform an initial proof of concept of our method, which shows our method is better than random prioritization technique. Moreover, the size and execution time of test procedures for an industry system could very large. We believe that industry users could benefit from applying our prioritization method.

10 CONCLUSION

UML is widely used in the software iterative development process, and various types of UML artifacts are involved in different activities, including requirement analysis, system design and testing. With a focus on testing, Siemens Corporate Research developed a CASE tool TDE/UML to generate functional test cases based on a UML test model. However, regression testing, which is an important and costly testing activity, is not supported by TDE/UML. The objective of this thesis is to propose a model-based regression testing approach to support regression testing in the context of TDE/UML.

Our regression testing approach is divided into two parts: regression test selection and regression test prioritization. In the regression test selection, we define a systematic way of detecting the changes to test models built for the system under test and derive traceability links between the original model elements and the existing test procedures. Then, based on the detected changes and traceability links, we formally and systematically define a set of criteria that can be used to classify test procedures of the existing test suites into three categories: obsolete, retestable and reusable. In addition to retestable test procedures, retestable test suites and retestable test cases (which contain retestable test procedures) are also identified. After that, a prioritization technique that orders retestable test procedures that have been obtained from the test selection is proposed. The prioritization technique begins with applying the AHP method to prioritize retestable test suites. Then, retestable test cases in a test suite are prioritized using either one objective or multiple objectives. The single objective approach only accounts for the expected risk of each test case, which is a measurement of potential risk for system under

test of not executing a test case. By running test cases with higher expected risk values early, users would reduce risk. The multiple objective approach considers test cases' expected risks and execution time to order them. More specifically, the SPEA2 algorithm is applied to solve these multi-objective test cases prioritization problem. Finally, the retestable test procedures within the same test cases are prioritized. According to the sequences for the three test artifacts, we obtain an overall sequence for the retestable test procedures.

We also perform a case study on a small academic project to examine the effectiveness of our regression testing approach. The result shows that the amount of test procedures that need to be retestable is significantly reduced. For the prioritization result, we found that the test sequences obtained either from the single-objective approach or the multi-objective approaches increase the cumulative expected risk faster than the randomly generated test sequences. In terms of mutant detection ability, APFD_c that combine the test cost and fault severity factors is applied to evaluate the effectiveness of our prioritization result. The result shows that the test procedure sequences generated by our approach are able to outperform most of the randomly generated test procedure sequences in most cases.

In summary, by providing a regression testing selection and prioritization approach based on a UML test model, we are able to: 1) achieve the efficiency in the test selection, so that we can reduce the number of test procedures that need to be rerun; 2) improve the rate of increase of the cumulative *nERisk* value and the mutant detection ability in test prioritization, compared to the random prioritization technique.

Finally, several possible further works can be mentioned. First, in order to address the limitations of the case study mentioned in section 9.4, we plan to perform a case study on a large scale industrial system, which could contains more than one retestable test suite as well as many more changes on the data flow entities of a UML test model. Second, more studies will be performed to check whether we need two objectives in the optimization scheme one of the multi-objective test case prioritization approach. Furthermore, as various regression test selection and regression prioritization techniques have been proposed in literatures, a comparison of our approach and the other selection and regression techniques could be investigated.

11 REFERENCES

- [1] "TDE/UML User Manual."
- [2] H. Leung and L. White, "Insights into regression testing," *Proceedings of the Conference on Software Maintenance*, pp. 60–69, 1989.
- [3] G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, pp. 173-210, 1997.
- [4] L. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," *Proc. International Conference on Software Maintenance*, pp. 252-261, 2002.
- [5] Y. Chen, R. Probert, and D. Sims, "Specification-based regression test selection with risk analysis," *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, 2002.
- [6] Chen, Y., Probert, R. L., and Ural, H., "Model-based regression test suite generation using dependence analysis", *Proc. of ACM ISSTA(AMOST)'07*, London, UK., pp.54-63, 2007.
- [7] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France, "Testing UML designs," *Information and Software Technology*, vol. 49, pp. 892-912, 2007.
- [8] O. Pilskalns and A. Andrews, "Regression Testing UML Designs," *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)-Volume 00*, pp. 254-264, 2006.
- [9] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929-948, 2001.
- [10] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA' 02)*, pp. 97-106, 2002.
- [11] J. Kim and A. Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments."
- [12] Z. Li, M. Harman, and R. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Transactions on Software Engineering*, vol. 33, pp. 225-237, 2007.

- [13] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos, "TimeAware test suite prioritization," *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 1-12, 2006.
- [14] B. Korel, L. Tahat, and M. Harman, "Test Prioritization Using System Models," *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 559-568, 2005.
- [15] B. Korel, G. Koutsogiannakis, and L. Tahat, "Model-based test prioritization heuristic methods and their evaluation," *Proceedings of the 3rd international workshop on Advances in model-based testing*, pp. 34-43, 2007.
- [16] H. Srikanth, L. Williams, and J. Osborne, " System test case prioritization of new and regression test cases," in *International Symposium on Empirical Software Engineering*, pp.62-71, 2007.
- [17] OMG, "Unified Modeling Language (UML)," Object Management Group V2.1.1, available at www.uml.org. [Accessed: May,14, 2009].
- [18] T. Ostrand and M. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, pp. 676-686, 1988.
- [19] B. Bruegge and A. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall, 2004.
- [20] "TDE/UML Plug-In Development Manual."
- [21] OMG, "Object Constraint Language (OCL)," Object Management Group, <http://www.omg.org/technology/documents/formal/ocl.htm>, 2007.
- [22] G. Rothmel and M. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 529-551, 1996.
- [23] T. Saaty, "The Analytic Hierarchy Process: Planning, Priority Setting," *Resource Allocation, McGraw-Hill, New York*, 1980.
- [24] J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE Software*, vol. 14, pp. 67-74, 1997.
- [25] G. Ruhe, A. Eberlein, and D. Pfahl, "Quantitative WinWin: a new method for decision support in requirements negotiation," *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pp. 159-166, 2002. [Accessed: May,14, 2009].
- [26] N. R. Mead, "Requirements Prioritization Case Study Using AHP," <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/534.html>. [Accessed: May,14, 2009].

- [27] L. Lehtola and M. Kauppinen, "Empirical Evaluation of Two Requirements Prioritization Methods in Product Development Projects," *Proceedings of the European Software Process Improvement Conference (EuroSPINo. 11)*, Springer, Trondheim, Norway, pp. 161-170, 2004.
- [28] D. Firesmith, "Prioritizing Requirements," *Journal of Object Technology*, vol. 3, pp. 35-47, 2004.
- [29] E. Gottesdiener, "Use Cases: Best Practices," http://www.ibm.com/developerworks/rational/library/content/03July/2500/2784/2784_use_cases.pdf. [Accessed: May, 14, 2009].
- [30] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*: Wiley, 2001.
- [31] D. Veldhuizen and G. Lamont, "Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art," *Evolutionary Computation*, vol. 8, pp. 125-147, 2000.
- [32] J. Holland, *Adaptation in natural and artificial systems*, MIT Press Cambridge, MA, USA, 1992.
- [33] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.
- [34] N. Srinivas and K. Deb, "Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation*, vol. 2, pp. 221-248, 1994.
- [35] M. Jensen, "Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 7, pp. 503-515, 2003.
- [36] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," in *EUROGEN*, pp. 95-100, 2001.
- [37] H. Cobb, J. Grefenstette, and N. R. L. W. DC, *Genetic Algorithms for Tracking Changing Environments*, Defense Technical Information Center, 1993.
- [38] J. Schaffer, R. Caruana, L. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proceedings of the third international conference on Genetic algorithms table of contents*, pp. 51-60, 1989.
- [39] R. Haupt and S. Haupt, *Practical genetic algorithms*, Wiley New York, 1998.
- [40] K. De Jong, "Learning with genetic algorithms: An overview," *Machine Learning*, vol. 3, pp. 121-138, 1988.

- [41] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Optimization*: Wiley-Interscience, 1999.
- [42] J. Smith and T. Fogarty, "Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm," in *Parallel Problem Solving from Nature IV*, pp. 441–450, 1996.
- [43] ECJ, "A Java-based Evolutionary Computation Research System," <http://cs.gmu.edu/~eclab/projects/ecj/>. [Accessed: May, 14, 2009].
- [44] MuJava, "mutation system for Java programs," <http://cs.gmu.edu/~offutt/mujava/>. [Accessed: May, 14, 2009].
- [45] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *International Conference on Software Engineering: Proceedings of the 23 rd International Conference on Software Engineering: Toronto, Ontario, Canada*, pp. 329-338, 2001.

Appendix A Addendum to OCL Expressions

A.1 Description of Class Helper

The formalization of methods defined in Helper is given as follows:

- 1) `getNodesForProcedure( t: TdeTestProcedure)`

-Description: This method returns all nodes (i.e.,) in a path tested by a given test procedure.

-Return type: Path.

-Parameters: t, an instance of TdeTestProcedure.

-Formalization:

```
context Helper::getNodesForProcedure(t:TdeTestProcedure):Path
post: result.nodesInPath->includeAll(t.tdeTestStep.corresponding)
```

- 2) `getActivityEdgeForProcedure( t: TdeTestProcedure)`

-Description: This method returns all transitions in a path tested by a given test procedure.

-Return type: Path.

-Parameters: t, an instance of TdeTestProcedure.

-Formalization:

```
context Helper::getActivityEdgesForProcedure(
    t:TdeTestProcedure):Path
post: let edges:Set(ActivityEdge)
    = t.tdeTestStep.corresponding.outgoing
    ->select(oclIsTypeOf(ActivityEdge))
    ->asSet()
in result.edgesInPath
    ->includeAll(edges).
```

- 3) `isNoteContainVD( note:NoteInActivityDiagram)`

-Description: This method checks whether a note in activity diagram contains variable declarations.

-Return type: Boolean.

-Parameters: note is an instance of NoteInActivityDiagram.

-Formalization:

```
context Helper::isNoteContainVD(
    note:NoteInActivityDiagram):Boolean
post: result = (note.vars->size() > 0)
```

- 4) `isVNChanged(n1:NoteInActivityDiagram, n2:NoteInActivityDiagram)`

-Description: This method checks whether two notes have the same variable declaration. If any variable is defined in one note but not in another note, then this method returns true.

-Return type: Boolean.

-Parameters: n1 and n2 are instances of NoteInActivityDiagram.

-Formalization:

```
context Helper::isVNChanged(n1:NoteInActivityDiagram,  
                             n2:NoteInActivityDiagram):Boolean  
post: (n1.vars->size()==0 and n2.vars->size()>0 )  
       implies result=true  
       or  
       n1.vars->size()>0 and n1.vars->size()==0 )  
       implies result=true  
       or  
       n1.vars.define.exists( v: Variable |  
                             v2.vars.define.excludes(v))  
       implies result=true  
       or  
       n1.vars.define.exists( v: Variable |  
                             v1.vars.define.excludes(v))  
       implies result=true
```

5) isVTChanged(n1:NoteInActivityDiagram, n2:NoteInActivityDiagram)

-Description: This method checks whether the types of variables defined in two notes have been changed. If any of variables defined in note n1 has a different type in note n2, this method returns true.

-Return type: Boolean.

-Parameters: n1 and n2 are instances of NoteInActivityDiagram.

-Formalization:

```
context Helper::isVTChanged(n1:NoteInActivityDiagram,  
                             n2:NoteInActivityDiagram):Boolean  
post: n1.vars.define.exists( v: VarDeclaration |  
                             v2.vars.define  
                             ->select( name=v.name  
                                     and type<>v.type)  
                             ->notEmpty())  
       implies result=true
```

6) isTestProcedureSatisfied(t:TdeTestProcedure, c: Constraint)

-Description: This method returns true if a test procedure can satisfy the given constraint.

-Return type: Boolean.

-Parameters: t is an instance of TdeTestProcedure, c is an instance of classes that implement interface Constraint.

7) getCategoriesofNoteInClassDiagram(note: NoteInClassDiagram)

-Description: This method returns all categories that have been involved in a note in the class diagram.

-Return type: Set.

-Parameters: note is an instance of NoteInClassDiagram.

-Formalization:

```
context Helper:: getCategoriesofNoteInClassDiagram(  
                             note: NoteInClassDiagram):Set  
pre: not(isVNChanged(n1,n2))  
post: result->size() > 0.
```

8) getAddedOrDeletedOutgoingEdgesOfDecision(d: Decision)

-Description: This method returns all added or deleted outgoing edges of a decision node.

-Return type: Set.

-Parameters: d is an instance of Decision.

9) `isAttValueChanged(aValuesV1:Set, aValuesV2:Set)`

- Description: This method checks whether the attribute values of two choices are different. If any of the attribute value in aValuesV1 is not appeared in aValuesV2 in any of the attribute value in aValuesV2 is not appeared in aValuesV1, then this method returns false. Otherwise, it returns true.

- Return type: Boolean.

- Parameters: aValuesV1 and aValuesV2 are a set of AttributeValues.

- Formalization:

```
context
Helper::isAttValueChanged(aValuesV1:Set, aValuesV2:Set):Boolean
post: aValuesV1.exists( a: AttributeValue |
                        aValuesV2->excludes(a)
                        implies result=true)
or
aValuesV2.exists( a: AttributeValue |
                aValuesV1->excludes(a)
                implies result=true)
```

10) `getInheritances( c: Category)`

- Description: This method returns all the inheritance relationships of a given a category.

- Return type: Set.

- Parameters: c is an instance of Category.

11) `getAssociations( c: Category)`

- Description: This method returns all the association relationships of a given a category.

- Return type: Set.

- Parameters: c is an instance of Category.

12) `isBaseCategoryChanged( c: Category)`

- Description: This method checks whether any of the base category/categories of a given category is a Changed Category. If yes, then this method returns true.

- Return type: Boolean.

- Parameters: c is an instance of Category.

13) `getNextNode( e: DiagramElement )`

- Description: This method returns the next node of a given node in the activity diagrams.

- Return type: DiagramElement.

- Parameters: e is an instance of DiagramElement.

- Formalization:

```
context Helper::getNextNode(e:DiagramElement)
pre: e.oclIsTypeOf(Decision) = true
```

```

or
e.ocIsTypeOf(ActivityStartState) = true
or
e.ocIsTypeOf(SyncItem) = true
or
e.ocIsTypeOf(ActivityStopState) = true
or
e.ocIsTypeOf(Activity) = true

```

14) isOnlyContainDecisions(e1 DiagramElement, e2
DiagramElement): Boolean

- **Description:** This method checks whether there is only one (or more) decision node on a subpath between two given nodes in the activity diagrams.

- **Return type:** Boolean.

- **Parameters:** e1 and e2 and instance of DiagramElement.

context Helper::isOnlyContainDecisions(e1 DiagramElement,
e2 DiagramElement): Boolean

pre: e1.ocIsTypeOf(Decision) = true
or
e1.ocIsTypeOf(ActivityStartState) = true
or
e1.ocIsTypeOf(SyncItem) = true
or
e1.ocIsTypeOf(ActivityStopState) = true
or
e1.ocIsTypeOf(Activity) = true
or
e2.ocIsTypeOf(Decision) = true
or
e2.ocIsTypeOf(ActivityStartState) = true
or
e2.ocIsTypeOf(SyncItem) = true
or
e2.ocIsTypeOf(ActivityStopState) = true
or
e2.ocIsTypeOf(Activity) = true

15) isSubCategory(vd: VarDeclaration)

-**Description:** Given a variable declaration with changed type, this method checks the new type is the sub category of the original type. If yes, this method returns true.

-**Return type:** Boolean.

-**Parameters:** vd is an instance of VarDeclaration.

16) isBaseCategory(vd: VarDeclaration)

-**Description:** Given a variable declaration with changed type, this method checks the new type is the base category of the original type. If yes, this method returns true.

-**Return type:** Boolean.

-**Parameters:** note is an instance of NoteInActivityDiagram.

17) hasSameBaseCategory(vd: VarDeclaration)

-Description: Given a variable declaration with changed type, this method checks the new type has the same base category as the original one. If yes, this method returns true.

-Return type: Boolean.

-Parameters: note is an instance of NoteInActivityDiagram.

18) isChoiceDefinedInBaseCategory(vd: VarDeclaration)

-Description: Given a variable, this method checks whether the value of the variable is the choice defined in the base category of the type of this variable. If yes, this method return true.

-Return type: Boolean.

-Parameters: note is an instance of NoteInActivityDiagram.

19) getConstraintsInvolvesVarsofChoice(c: Choice)

-Description: This method returns a set of constraints that use the given choice.

-Return type: Set.

-Parameters: c is an instance of Choice.

20) getConstraintsInvolvesVarsofChoice(c: Choice)

-Description: This method returns a set of constraints constraints that involve variables that take the given choice as their values.

-Return type: Set.

-Parameters: c is an instance of Choice.

21) getTypeChangedVarDeclaration(note: NoteInActivityDiagram)

-Description: This method returns a set of note that contains variable declarations with changed types.

-Return type: Set.

-Parameters: note is an instance of NoteInActivityDiagram.

A.2 Constraints on the Metamodel

We describe constraints for the metamodel in the following OCL expressions:

- Constraint on class ActivityDiagram: An activity diagram contains only one start node.

```
context ActivityDiagram
inv: self.diagramElements
    ->select(e | e.ocllsTypeOf(ActivityStarState))
    ->size()=1.
```

- Constraint on class UseCase: Each use case is associated with an activity diagram to describe its behavior.

```
context UseCase
inv: self.childDiagram.oclIsTypeOf(ActivityDiagram) = true.
```

- Constraint on class TdeTestProcedure: the set of use cases a procedure involves contains the starting use case of the corresponding test suite.

```
context TdeTestProcedure
inv: self.involves->includes(self.tdeTestCase.
tdeTestSuite.startingUseCase)
```

- Constraint on class Activity: If an activity has a stereotype <<usecase>>, then it refers to a use case. If an activity has a stereotype <<nested>>, then there is an activity diagram to describe its behavior.

```
context Activity
inv: self.stereotype=usecase
      implies= self.refer->notEmpty()
      or
      self.stereotype=nested
      implies self.childDiagram->NotEmpty()
```

- Constraint on class VariableChoice: For a variable-choice binding, the type of its variable should be the same as the category of its associated choice.

```
context VariableChoice
inv: self.correspondChoice.category=self.correspondVar.type
```

- Constraint on class UseCaseDiagram: The DiagramElements that a use case diagram contains could only be UseCase.

```
context UseCaseDiagram
inv: self.diagramElements
      ->forAll(e:DiagramElement |
              e.oclIsTypeOf(UseCase) = true)
```

- Constraint on class ActivityDiagram: The DiagramElements that an activity diagram contains could only be Decision, ActivityStartState, SyncItem, ActivityStopState, Activity, and Note.

```
context ActivityDiagram
inv: self.diagramElements
      ->forAll(e:DiagramElement |
              e.oclIsTypeOf(Decision) = true
              or e.oclIsTypeOf(ActivityStartState) = true
              or e.oclIsTypeOf(SyncItem) = true
              or e.oclIsTypeOf(ActivityStopState) = true
              or e.oclIsTypeOf(Activity) = true
              or e.oclIsTypeOf(Note) = true )
```


- Constraint on class ClassDiagram: The DiagramElements that a class diagram contains could only be Choice, Category, and Note.

```
context ClassDiagram
inv: self.diagramElements
    ->forall(e:DiagramElement |
        e.ocIsTypeOf(Choice) = true
        or e.ocIsTypeOf(Category) = true
        or e.ocIsTypeOf(Note) = true )
```

- Constraint on class Inheritance and Association: An Inheritance or an Association is only used to specify relationship between categories.

```
context Inheritance
inv: self.from.ocIsTypeOf( Category ) = true
    and self.to.ocIsTypeOf( Category ) = true

context Association
inv: self.from.ocIsTypeOf( Category ) = true
    and self.to.ocIsTypeOf( Category ) = true
```

- Constraint on class DependencyRelationship: A DependencyRelationship is only used to specify relationship between use cases.

```
context DependencyRelationship
inv: self.from.ocIsTypeOf( UseCase ) = true
    and self.to.ocIsTypeOf( UseCase ) = true
```

- Constraint on class ActivityEdge: An ActivityEdge is only used to connect two nodes in activity diagrams, such as decision nodes, fork nodes, joins nodes, start nodes, stop nodes and activities.

```
context ActivityEdge
inv: (self.from.ocIsTypeOf(Decision) = true
    or self.from.ocIsTypeOf(SyncItem) = true
    or self.from.ocIsTypeOf(ActivityStartState) = true
    or self.from.ocIsTypeOf(ActivityStopState) = true
    or self.from.ocIsTypeOf(Activity) = true)
    and
    (self.to.ocIsTypeOf(Decision) = true
    or self.to.ocIsTypeOf(SyncItem) = true
    or self.to.ocIsTypeOf(ActivityStartState) = true
    or self.to.ocIsTypeOf(ActivityStopState) = true
    or self.to.ocIsTypeOf(Activity) = true)
```

- Constraint on class NoteConnection: A NoteConnection is only used to connect a note and a node in activity diagram.

```
context NoteConnection
inv: (self.from.ocIsTypeOf(Decision) = true ||
    self.from.ocIsTypeOf(SyncItem) = true ||
    self.from.ocIsTypeOf(ActivityStartState) = true ||
    self.from.ocIsTypeOf(ActivityStopState) = true ||
    self.from.ocIsTypeOf(Activity) = true)
    and
```

```
(self.to.ocIsTypeOf( Note ) = true )
```

- Constraint on class NoteInActivityDiagram: A NoteInActivityDiagram only can be presented in activity diagrams.

```
context NoteInActivityDiagram
inv: self.diagram.ocIsTypeOf(ActivityDiagram) = true.
```

- Constraint on class NoteInClassDiagram: A NoteInClassDiagram only can be presented in class diagrams.

```
context NoteInClassDiagram
inv: self.diagram.ocIsTypeOf(ClassDiagram) = true.
```

- Constraint on class TdeTestProcedure: Given a test procedure, the use cases that it involves include the use case that is the starting use case of the test suite that it belongs to.

```
context TdeTestProcedure
inv: self.involved
    ->includes(self.tdeTestcase.tdeTestsuite.startingUseCase).
```

- Constraint on class Path: For a path tested by a test procedure, the outgoing transition of the i^{th} nodes of the path is the i^{th} transition of the path except the last node. The incoming transition of the last node is the last transition.

```
context Path
inv: {1..self.nodesInPath->size()-1}
    ->forall(i:Integer|self.nodesInPath
    ->at(i).outgoing =
        self.edgesInPath->at(i)
    and
    self.nodesInPath->at( self.nodesInPath->size()).incoming
    = self.edgesInPath->at( self.edgesInPath->size()).
```

- Constraint on class TDEProperty: Each kind of property is associated with a limited number of values. For instance, the property coverage-algorithm can be only set to all-paths, all-activities, all-transitions, round-trip or happy-path. This is the purpose of the following OCL expression where the extension point indicates that the constraints for other property kinds and values would be defined in the same way.

```
context TDEProperty
inv: (self.kind=Propertykind.coverage-algorithm)
    =(self.value=PropertyValue.all-paths
    xor self.value=PropertyValue.all-activities
    xor self.value=PropertyValue.round-trip
    xor self.value=PropertyValue.happy-path)
    and
    (self.kind=PropertyKind.choice-coverage-algorithm)
```

```

        =(self.value= ...)
...           //other property kinds and values
              //would be defined in the same way

```

A.3 Traceability links for Model Elements

This Appendix section specifies the definitions of traceability links for model elements and test procedures.

- Traceability between test procedures and the include/extend relationships that they involve (i.e., the include/extend relationships shared by the use cases they involve).

```

context TdeTestProcedure
inv:  self.involves->outgoing
        ->select(oclIsTypeOf(DependencyRelationship))
        ->asSet()
        ->forAll(rel:DependencyRelationship|
                self.traceabilitylink
                ->exists(t :TraceabilityLink|
                        t.tdeTestprocedure=self
                        and t.diagramRelationship=rel))

```

- Traceability between nodes (i.e. decisions, forks/joins, start/end nodes, and activities) and the test procedures that involve them (operation `getNodesForProcedure()` of class `Helper` returns the nodes triggered by a test procedure):

```

context TdeTestProcedure
inv:  Helper.getNodesForProcedure(self)
        ->forAll(e: DiagramElement |
                self.traceabilitylink
                ->exists(t:TraceabilityLink|
                        t.tdeTestprocedure=self
                        and t.diagramElement=e))

```

- Traceability between activity edges and the test procedures that involve them (operation `getActivityEdgesForProcedure()` of class `Helper` returns the edges triggered by a test procedure):

```

context TdeTestProcedure
inv:  Helper.getActivityEdgesForProcedure(self)
        ->forAll(edge :ActivityEdge |self.traceabilitylink
                ->exists(t:TraceabilityLink|
                        t.tdeTestprocedure=self
                        and t.diagramRelationship=edge)

```

- Traceability between note connections and test procedures.

```

context TdeTestProcedure
inv:  Helper.getNodesForProcedure(self).outgoing

```

```

->select (oclIsTypeOf (NoteConnection))
->forall (nc : NoteConnection |
    nc.traceabilitylink
    ->exists (t:TraceabilityLink |
        t.tdeTestprocedure= self
        and t.diagramRelationship= nc))

```

- Traceability between notes specified in activity diagrams and test procedures

```

context TdeTestProcedure
inv: Helper.getNodesForProcedure (self) -> outgoing
->select (oclIsTypeOf (NoteConnection))
->forall ( nc : NoteConnection |
    nc.to.traceabilitylink
    ->exists (t:TraceabilityLink |
        t.tdeTestprocedure= self
        and t.diagramElement= nc.to))

```

- Traceability between categories and test procedures: If a category is used when defining a variable in a test procedure, the test procedure is linked to that category. Moreover, if a variable of a test procedure is typed by a category that inherits from another category, and that variable is assigned a choice that is defined in the super category, then the super category is also linked to the test procedure. However, if a category is used several times in a test procedure, there is only one traceability link. (Note that we do not account for categories used in guard conditions. We are not doing any data flow analysis but simply want to identify which categories are used in a test procedure. One simple way to identify those is to look for their use in variable definitions.)

```

context TdeTestProcedure
inv: self.tdeTestStep.variableChoice.variable.type -> asSet ()
->forall (c:Category |
    self.traceabilitylink
    ->exists (t :TraceabilityLink |
        t.tdeTestprocedure= self
        and t.diagramElement= c))
and
self.tdeTestStep.variableChoice
->forall (vc: VariableChoice |
    not ( vc.variable.type.oclIsTypeof (
        vc.choice.category.OclType))
    implies self.traceabilitylink
    ->exists (t :TraceabilityLink |
        t.tdeTestprocedure= self
        and t.diagramElement= vc.choice.category))

```

- Traceability between inheritance relationships and test procedures: if a variable of a test procedure is typed by a category that has an inheritance relationship with another category, and that variable is assigned a choice that is defined in the super

category, then a traceability link should also be built between the inheritance relationship and the test procedure.

```

context TdeTestProcedure
inv: self.tdeTestStep.variableChoice
    ->forAll(vc: VariableChoice|
        not( vc.variable.type.oclIsTypeof(
            vc.choice.cateogry.OclType))
        implies self.traceabilitylink
        ->exists(t :TraceabilityLink|
            t.tdeTestprocedure=self
            and
            t.diagramElements=vc.choice.category.from))

```

- Traceability between choices and test procedures that involve them:

```

context TdeTestProcedure
inv: self.tdeTestStep.variableChoice.choice ->asSet()
    ->forAll(c:Choice|
        self.traceabilitylink
        ->exists(t :TraceabilityLink|
            t.tdeTestprocedure=self
            and t.diagramElement=c))

```

- Traceability between NoteInClassDiagram and test procedures: In class diagrams, notes stereotyped by <<TDE/UML>> are used to specify global constraints. Because each global constraint may be associated with one or more categories and those categories can be used in different activities, the related test procedures of a global constraint should combine the related test procedures of those associated categories. For example, Figure 11 b) shows two global constraints in a note. The categories involved in the first global constraint, are Shape and Colour. So the related test procedures for that global constraint should be the combination of related test procedures for categories Shape and Colour.

```

context TdeTestProcedure
inv let   categoris:Set(Category)
        = self.tdeTestStep.variableChoice.varialbe.type
        ->asset()
In      NoteInClassDiagram.allInstance()
        ->forAll( n: NoteInClassDiagram |
        not (categories.excludesAll(
            getCategoryiesInGlobalConstraint(n))
        implies self.traceabilitylink
        ->exist( t: TraceabilityLink|
            t.tdeTestProcedure=self
            and t.diagramElement=n ))

```

A.4 OCL Expressions for Model Element Changes

This Appendix section contains the OCL expressions that were deemed simple enough (or similar enough to other expressions) to be omitted from Chapter 5.

A.4.1 Changes to use case relationships

- Deleted use case relationship.

```
context DependencyRelationship
inv: if self.change.version = 1 then
    DependencyRelationship.allInstance->select (
        version=2 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif
```

- Added use case relationship.

```
context DependencyRelationship
inv: if self.change.version = 2 then
    DependencyRelationship.allInstance->select (
        version=1 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif
```

- Changed use case relationship.

```
context DependencyRelationship
inv: if self.change.version = 1 then
    DependencyRelationship.allInstance->select (
        version=1 and id=u.id
        and (from<>self.from or to<>self.to))->notEmpty()
    implies self.change.type=ChangeType.changed
endif
```

- Unchanged use case relationship.

```
context DependencyRelationship
inv: if self.change.version = 1 then
    DependencyRelationship.allInstance->select (
        version=1 and id=u.id
        and (from=self.from and to=self.to))->notEmpty()
    implies self.change.type=ChangeType.notChanged
endif
```

A.4.2 Changes to Nodes

- Deleted Activity.

```
context Activity
inv: if self.change.version=1 then
    Activity.allInstance->select (change.version=2
        and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif
```

- Added Activity.

```
context Activity
inv: if self.change.version=2 then
    Activity.allInstance->select(change.version=1
                                and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif
```

- Changed fork/join.

```
context SyncItem
inv: if self.change.version = 1 then
    (self.outgoing.id <> SyncItem.allInstance
     ->select(change.version=2 and id=self.id).outgoing.id
    implies (self.change.type=ChangeType.changed
            and self.change.isOutgoingChanged=true)
    or
    self.incoming.id <> SyncItem.allInstance
     ->select(change.version=2 and id=self.id).incoming.id
    implies (self.change.type=ChangeType.changed
            and self.change.isIncomingChanged=true)
endif
```

A.4.3 Changes to transitions

- Deleted transition.

```
context ActivityEdge
inv: if self.change.version = 1 then
    ActivityEdge.allInstance->select(
        change.version=2 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif
```

- Added transition.

```
context ActivityEdge
inv: if self.change.version = 2 then
    ActivityEdge.allInstance->select(
        change.version=1 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif
```

A.4.4 Changes to Notes in the Activity Diagram

- Deleted NoteInActivityDiagram.

```
context NoteInActivityDiagram
inv: if self.change.version = 1 then
    NoteInActivityDiagram.allInstance->select(
        change.version=2 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif
```

- Added NoteInActivityDiagram.

```
context NoteInActivityDiagram
inv: if self.change.version = 2 then
    NoteInActivityDiagram.allInstance->select(
```

```

        change.version=1 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.5 Changes to Note Connections

- Deleted note connection.

```

context NoteConnection
inv: if self.change.version = 1 then
    NoteConnection.allInstance->select(
        change.version=2 and id= self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif

```

- Added note connection.

```

context NoteConnection
inv: if self.change.version = 2 then
    NoteConnection.allInstance->select(
        change.version=1 and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.6 Changes to Associations

- Deleted association.

```

context Association
inv: if self.change.version = 1 then
    Association.allInstance->select(version=2 and id=self.id)
        ->isEmpty()
    implies self.change=ChangeType.deleted
endif

```

- Added association.

```

context Association
inv: if self.change.version = 2 then
    Association.allInstance->select(version=1 and id=self.id)
        ->isEmpty()
    implies self.change=ChangeType.added
endif

```

A.4.7 Changes to Choices

- Deleted choice

```

context Choice
inv: if self.change.version = 1 then
    Choice.allInstance->select(change.version=2 and
id=self.id)
        ->isEmpty()
    implies self.change.type=ChangeType.deleted
    or
    Choice.allInstance->select(change.version=2 and
        (id=c.id and name<>c.name))->notEmpty()
    implies self.change.type=ChangeType.deleted
endif

```


- Added choice.

```

context Choice
inv: if self.change.version = 2 then
    Choice.allInstance->select(change.version=1 and
id=self.id)
    ->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.8 Changes to Inheritances

- Deleted Inheritance.

```

context Inheritance
inv: if self.change.version = 1 then
    Inheritance.allInstance>select(change.version=2
and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
endif

```

- Added Inheritance

```

context Inheritance
inv: if self.change.version = 2 then
    Inheritance.allInstance>select(change.version=1
and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.9 Changes to Attributes

- Deleted attribute

```

context Attribute
inv: if self.change.version = 1 then
    Attribute.allInstance->select(change.version=2
and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
or
    Attribute.allInstance->select(change.version=2
and id=self.id and name<>self.name)->notEmpty()
    implies self.change.type=ChangeType.deleted
endif

```

- Added attributes

```

context Attribute
inv: if self.change.version = 2 then
    Attribute.allInstance->select(change.version=1
and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.10 Changes to Categories

- Deleted Category

```

context Category
inv: if self.change.version = 1 then
    Category.allInstance->select(change.version=2
        and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.deleted
    or
    Category.allInstance->select(change.version=2
        and id=self.id and name<>self.name)->notEmpty()
    implies self.change.type=ChangeType.deleted
endif

```

- Added Category

```

context Attribute
inv: if self.change.version = 2 then
    Category.allInstance->select(change.version=1
        and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

A.4.11 Changes to Notes in the class diagram

- Deleted NoteInClassDiagram

```

context NoteInClassDiagram
inv: if self.change.version = 1 then
    NoteInClassDiagram.allInstance->select(change.version=2
        and id=self.id )->isEmpty()
    implies self.change.type=ChangeType.deleted
endif

```

- Added NoteInClassDiagram

```

context NoteInClassDiagram
inv: if self.change.version = 2 then
    NoteInClassDiagram.allInstance->select(change.version=1
        and id=self.id)->isEmpty()
    implies self.change.type=ChangeType.added
endif

```

- Changed NoteInClassDiagram

```

context NoteInClassDiagram
inv: if self.change.version = 1 then
    NoteInClassDiagram.allInstance->select(change.version=2
        and id=self.id and text<>self.text)->notEmpty()
    implies self.change.type=ChangeType.changed
endif

```

A.5 The rationale for fitness function f_3 in optimization scheme two.

The fitness function, to measure the rate at which a test case order reaches/exceeds the threshold of the cumulative nERisk, is defined as:

$$f3 = \sum_{i=1}^n \frac{1}{2} \times \text{Time}(i) \times ((S_nERisk - \sum_{j=1}^{j=i-1} nERisk(j)) + (S_nERisk - \sum_{j=1}^{j=i-1} nERisk(j)))$$

$$+ \frac{1}{2} \times \left(S_nERisk - \sum_{j=1}^{j=n} nERisk(j) \right)^2 \times \frac{\text{time}(n)}{eNRisk(n)};$$

This fitness function is the area bounded by the curve of cumulative expected risk, the y-axis and the (horizontal) line of the user specified expected cumulative risk to reach. To explain the rationale of the above fitness function, we reuse the two test case sequences of Table 9. Figure 42 shows the cumulative $nERisk$ value, versus the total execution time, for those two test orders. As illustrated in Figure 42, the area represented by $f3$ is the sum of a number of trapezoid areas (from point A to point B) plus the area of a triangle (triangle B, C, D). These form the two parts of the sum in the equation of $f3$.

The area of each trapezoid is:

$$\sum_{i=1}^n \frac{1}{2} \times \text{Time}(i) \times ((S_nERisk - \sum_{j=1}^{j=i-1} nERisk(j)) + (S_nERisk - \sum_{j=1}^{j=i-1} nERisk(j))),$$

where $S_nERisk - \sum_{j=1}^{j=i} nERisk(j)$ and $S_nERisk - \sum_{j=1}^{j=i-1} nERisk(j)$ are the lengths of the bases and $\text{Time}(i)$ is the altitude of the trapezoid. Remember that $\text{Time}(i)$ is the execution time of test case i and that time values are added on the x-axis.

Computing the area of the triangle is not immediate since the x value of point D is in general unknown: the last test case in the sequence may not exactly reach the expected (user defined) cumulative risk, but may exceed it (point D'). The area of the triangle is $\frac{1}{2} \times BC \times BD$. However, computing BD is not immediate. Fortunately, we know that

$$\tan(\angle BCD) = \frac{BD}{BC} = \frac{B'D'}{B'C}, \text{ and the coordinates of B', D', B and C are all known. Then,}$$

$$BD = \tan(\angle BCD) * BC = \frac{B'D'}{B'C} * BC. \text{ Therefore, the area of the triangle is } \frac{1}{2} \frac{B'D'}{B'C} * BC^2.$$

B'D' is the execution time of the last test case, $time(n)$, and B'C is the expected normalized risk of the last test case $eNRisk(n)$. Thus the area of the triangle is:

$$\frac{1}{2} \times \left(S_nERisk - \sum_n^{n-1} nERisk(i) \right)^2 \times \frac{time(n)}{eNRisk(n)}$$

Figure 42 Justifying fitness function f_3