INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

Regression Test Selection Based on UML Models

By

George A. M. Soccar, B. Eng.

**A thesis submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements of the degree
of
Masters of Engineering
Ottawa-Carleton Institute of Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, K1S 5B6
Canada**

January 2002

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-68323-0

Canada

The undersigned recommend to the
Faculty of Graduate Studies and Research

Acceptance of the thesis:

Regression Test Selection Based on UML Models

Submitted by

George A. M. Soccar

In partial fulfillment of the requirements

for the degree of

Masters of Engineering

A handwritten signature in black ink, appearing to read "Lionel Wong", written over a horizontal line.

Thesis Supervisor

A handwritten signature in black ink, appearing to read "S. Rafferty", written over a horizontal line.

Chair

Carleton University

© George Soccar, 2002

Abstract

In this research, we investigate the use of UML analysis and design models to perform impact analysis and regression test selection. We propose a model to extract the information from the UML models and present a host of algorithms to manipulate these models. We developed a Regression Test Selection Tool based on the proposed models and algorithms. The tool loads, stores and compares the UML models and then classifies the different test cases in the regression test suite. The tool is designed to handle Object Oriented issues such as inheritance and polymorphism and collects many useful statistics for the impact analysis process. Finally, a real industrial system has been used to perform comprehensive testing of the developed tool.

Acknowledgments

I would like to thank my supervisors, Lionel Briand and Yvan Labiche, for their understanding, guidance, support, and feedback.

I would like also to thank my parents and my family for their continued support.

Table of Contents

Chapter 1 – Introduction	1
1.1 Motivation	2
1.2 Contributions of This Thesis	4
1.3 Scope of This Thesis	5
1.4 Basic Definitions	6
1.4.1 The Nature of Software Changes	6
1.4.2 The Software Design Process	7
1.4.3 Inheritance and Polymorphism	8
1.4.4 The Message Sequence Specification (MgSS)	8
1.4.5 Impact Analysis	9
1.4.6 Object Oriented Testing	10
1.4.7 System and Subsystem Testing	10
1.4.8 Regression Testing	11
1.4.9 Traceability	12
1.5 Related Work	12
1.6 Thesis Organization	17
Chapter 2 - Analysis, Design and Algorithms	19
2.1 Problem Definition	20
2.1.1 Changes in Models	21
2.1.2 Issues	25
2.1.2.1 Attribute Changes	25
2.1.2.2 Identifying Methods in Sequence Diagrams	26
2.1.3 Impact on Test Cases	28
2.2 Outline of the Proposed Solution	30
2.2.1 The ClassDiagram Subsystem	31
2.2.2 The UseCaseDiagram Subsystem	33
2.2.3 The RegressionTestSuite Subsystem	35
2.3 Algorithms	37
2.3.1 Class Diagram Changes Algorithm	38
2.3.1.1 Input/Output	41
2.3.1.2 Algorithm	41
2.3.1.3 Internal Functions and Procedures	42
2.3.1.3.1 Function findChangedClasses	42
2.3.1.3.2 Function compareClasses	43
2.3.1.3.3 Function findChangedAttributes	44
2.3.1.3.4 Function findChangedClassAttributes	45
2.3.1.3.5 Function compareAttributes	46
2.3.1.3.6 Function findChangedMethods	46

2.3.1.3.7	Function findChangedClassMethods.....	47
2.3.1.3.8	Function compareMethods	48
2.3.1.3.9	Function compareMethodParameters.....	49
2.3.1.3.10	Procedure handleAttributeChanges	50
2.3.1.3.11	Procedure handleMethodInheritance.....	51
2.3.2	Use Case Diagram Changes Algorithm	52
2.3.2.1	Input/Output	54
2.3.2.2	Algorithm	54
2.3.2.3	Internal Functions.....	55
2.3.2.3.1	Function findChangedUseCases	55
2.3.2.3.2	Function findChangedMethods	56
2.3.2.3.3	Function findUseCaseChanges	57
2.3.2.3.4	Function compareMethods	58
2.3.2.3.5	Function compareMethodParameters.....	59
2.3.3	Consistency Check Algorithm	60
2.3.3.1	Input/Output	61
2.3.3.2	Algorithm	61
2.3.3.3	Internal Functions and Procedures	61
2.3.3.3.1	Procedure checkMethodSets	61
2.3.3.3.2	Function convertToInvokedMethod.....	64
2.3.4	Regression Test Suite Algorithm	64
2.3.4.1	Input/Output	65
2.3.4.2	Algorithm	65
2.3.4.3	Internal Functions and Procedures	66
2.3.4.3.1	Procedure classifyTestCases	66
2.3.4.3.2	Function mergeSets	68
2.3.4.3.3	Function testCaseMatcher	69
Chapter 3 - Regression Test Selection Tool	70	
3.1	<i>Description and Main Features</i>	70
3.2	<i>RTSTool Design.....</i>	73
3.2.1	Use Case Diagram.....	76
3.2.2	Activity Diagram.....	83
3.3	<i>RTSTool Implementation.....</i>	84
Chapter 4 - Case Study	87	
4.1	<i>The Discovery System.....</i>	87
4.1.1	General Description.....	87
4.1.2	Main Components of the System	88
4.1.3	Design of the Discovery Subsystem.....	89
4.1.3.1	Use Case Diagram.....	89
4.1.3.2	Class Diagrams.....	92
4.2	<i>Test Cases for the Discovery System.....</i>	95
4.2.1	Generated Test Cases	96
4.2.1.1	Generating Use Case Sequences	96

4.2.1.1.1	Representation of Use Case Sequences.....	96
4.2.1.1.2	Analysis of Use Case Weighted Graphs	98
4.2.1.2	Identifying Use Case Scenarios.....	99
4.2.1.2.1	Expressing Sequence Diagrams as Regular Expressions..	100
4.2.1.2.2	Identifying Path Realization Conditions	101
4.2.1.2.3	Specifying Operation Sequences.....	102
4.2.1.2.4	Identifying Test Oracles.....	103
4.2.1.2.5	Constructing Decision Tables	103
4.2.1.3	Generating Variant Sequences	104
4.2.2	Existing Test Cases	106
4.3	<i>Regression Testing Results for the Discovery System</i>	107
4.3.1	Impact Analysis Results.....	107
4.3.2	Regression Test Selection Results	109
Chapter 5 – Conclusions		112
5.1	<i>Limitations</i>	113
5.2	<i>Future Work</i>	115
References		116
Appendix A: The Discovery System		120
<i>Use Case Description</i>		120
<i>Discovery Original Sequence Diagrams</i>		128
<i>Discovery Modified Sequence Diagrams</i>		139
<i>Data Dictionary</i>		150
Appendix B: Data Dictionary for the RTSTool Design		177

List of Figures

Figure 1: Problem Definition	20
Figure 2: Method Changes	23
Figure 3: Identifying Methods in Sequence Diagrams.....	27
Figure 4: Packages and their Relationships.....	30
Figure 5: ClassDiagram Subsystem	32
Figure 6: UseCaseDiagram Subsystem	34
Figure 7: RegressionTestSuite Subsystem	36
Figure 8: Class Diagram Changes Algorithm Sequence of Messages	40
Figure 9: Use Case Diagram Changes Algorithm Sequence of Messages.....	53
Figure 10: RTSTool Description.....	71
Figure 11: RTSTool Subsystems and their Relationships.....	74
Figure 12: RTSTool Use Case Diagram	77
Figure 13: RTSTool Activity Diagram	84
Figure 14: The Discovery System.....	88
Figure 15: Discovery System Use Case Diagram	91
Figure 16: Original Discovery Class Diagram.....	93
Figure 17: Modified Discovery Class Diagram	94
Figure 18: The Discovery System Activity Diagram.....	97
Figure 19: Weighted Graph Representation of the Activity Diagram	98
Figure 20: CreateHelper Sequence Diagram.....	101
Figure 21: Original AddAgent Sequence Diagram.....	128
Figure 22: Original AddAgentOnStandaloneCard Sequence Diagram.....	129
Figure 23: Original AddCard Sequence Diagram	130
Figure 24: Original CreateHelper Sequence Diagram	131
Figure 25: Original DeleteAgent Sequence Diagram	132
Figure 26: Original DeleteAgentOnStandaloneCard Sequence Diagram	133
Figure 27: Original DownloadGlobalTable Sequence diagram.....	134
Figure 28: Original DownloadLocalTable Sequence Diagram.....	135
Figure 29: Original RegisterAgent Sequence Diagram.....	136
Figure 30: Original RegisterEphemeral Sequence Diagram	137
Figure 31: Original UpdateGlobalTable Sequence Diagram	138
Figure 32: Modified AddAgent Sequence Diagram	139
Figure 33: Modified AddAgentOnStandaloneCard Sequence Diagram	140
Figure 34: Modified AddCard Sequence Diagram	141
Figure 35: Modified CreateHelper Sequence Diagram.....	142
Figure 36: Modified DeleteAgent Sequence Diagram.....	143
Figure 37: Modified DeleteAgentOnStandaloneCard Sequence Diagram	144
Figure 38: Modified DownloadGlobalTable Sequence Diagram	145
Figure 39: Modified DownloadLocalTable Sequence Diagram	146
Figure 40: Modified RegisterAgent Sequence Diagram	147
Figure 41: Modified RegisterEphemeral Sequence Diagram	148
Figure 42: Modified UpdateGlobalTable Sequence Diagram.....	149

List of Tables

Table 1: Matrix Representation of the Weighted Graph	99
Table 2: Decision Table for CreateHelper Use Case	103
Table 3: Impact Analysis Results.....	108
Table 4: Regression Test Selection Results	109

Chapter 1 – Introduction

Regression testing plays a crucial role in validating new versions of existing systems. The purpose of regression testing is to reveal regression faults caused by side effects, incompatibilities, and undesirable feature interactions between two versions (an original and a modified version) of the system [Binder 99]. It is a common practice during an iterative development process. Given two instances of a certain design, where the second one is a modified version of the first one; a regression test case is typically a test case that has passed during the testing of the original version and is expected to pass when rerun on the modified version.

As the software evolves after several release cycles, a regression test suite can become quite large. Some test cases become obsolete as the application's interface, capabilities, and implementation evolve. Some test cases will simply fail to run and must be removed. Hence the need for regression test selection, where we classify the different test cases in the original regression test suite based on the changes introduced in the modified version.

This thesis explores the possibilities of reducing the size of the regression test suite by analyzing the impact of changes on UML design and analysis models and using traceability between these models and regression test suites to decide whether test cases are obsolete, must be retested or do not need to be rerun, thus leading to up-to-date, leaner regression test suites.

1.1 Motivation

The main purpose of impact analysis [Bohner+96] is to identify the potential consequences and side effects of a software change. The major technology areas for impact analysis are code based analysis techniques. Source-code analysis can be complex and time consuming especially with large software systems.

Traceability refers to the ability to define and trace relationships among entities. It is the association of data generated in a particular analysis and design activity with other data generated in predecessor and successor activities.

UML analysis and design models offer an attractive alternative to perform impact analysis and regression test selection. The reasons for that can be summarized as follows:

1. Changes to UML models are produced in the early analysis and design stages of the software design process. Hence, impact analysis of the proposed changes can be determined earlier, at the design level.
2. The effort involved in analyzing the UML models is much less than the effort involved in analyzing the code especially with large and complex systems.
3. Traceability between design changes and regression test cases is easier to manage than it is between code and regression test cases.
4. Having the regression test selection results in the early design stages, allows for better test planning and allocation of testing resources.

A significant reduction in the regression test suite size can be obtained using the information collected from the UML models assuming there is

traceability information between the models and the regression test cases. Thus, UML models represent a good candidate for regression test selection. The main approach used for this case is to classify the regression test cases in one of three categories: obsolete, retestable, or reusable.

A test case is an obsolete test case if it contains an invalid sequence of messages based on the new UML sequence diagrams associated with the modified version. A retestable test case is a test case that includes a changed method and must be executed as part of the regression test suite for the modified version. A reusable test case is a valid test case that does not contain any of the changed methods and therefore does not need to be rerun.

Our goal is to investigate the potentials of employing UML models in regression test selection and impact analysis. The approach adopted to achieve this goal is through the incorporation of the information collected from the design changes and the traceability information between UML models and regression test cases. This approach has been implemented and packaged in a fully automated tool that takes as input two versions of UML models, infers the design changes and traceability information and then classifies the regression test cases according to the above-described three categories.

A regression test selection utility has been developed as part of this research effort with the objective of offering the end user a fully automated analysis tool with a high-end Graphical User Interface (GUI).

1.2 Contributions of This Thesis

This thesis lays the groundwork to use UML analysis and design models to perform comprehensive impact analysis and regression test selection for O-O software systems. Algorithms are developed to compare the different versions of various UML models. Other algorithms use the comparison results to perform consistency checks and regression test selection. A fully automated tool that implements these algorithms has also been developed. The Regression Test Selection Tool (RTSTool) features a GUI, an XMI parser to load the UML models and a text parser to load the regression test suite.

A first algorithm compares two versions of class diagram models to detect the sets of added, deleted and modified attributes, classes and methods. Similarly, a second algorithm compares two versions of use case diagram models and their associated sequence diagrams to detect the sets of added, deleted and modified methods and use cases.

Based on the comparison results above, consistency checking and regression test selection can be performed by analytical algorithms. The consistency check algorithm uses the sets of methods generated by the previous algorithms to signal inconsistencies in methods used by the class and use case diagrams.

Regression test selection aims at reducing the size of the regression test suite. This is achieved through the last algorithm developed in this thesis. The proposed algorithm classifies the different test cases as obsolete, retestable, or reusable based on the results of impact analysis.

1.3 Scope of This Thesis

In order to perform effective and accurate impact analysis and regression test selection, three types of UML models [Bruegge+99] will be considered in this thesis:

1. **Class Diagrams:** Describes the structure of the system in terms of classes and objects. Classes are abstractions that specify the attributes and behavior of a set of objects. Objects are entities that encapsulate state and behavior.
2. **Sequence Diagrams:** Describes patterns of communication among a set of interacting objects. An object interacts with another object by sending messages. The reception of a message by an object triggers the execution of an operation, which in turn may send messages to other objects.
3. **Use Case Diagrams:** Use Cases are used during requirements elicitation and analysis to represent the functionality of the system. The use cases focus on the behavior of the system from an external point of view.

From these three types of models and through the use of our proposed models and algorithms, our aim is to perform the impact analysis of the following types of changes by comparing two UML models:

1. Addition/deletion of a class, class relationship, attribute, method, method parameter, sequence diagram or use case.
2. Call sequence changes: Method calling sequence changes or addition/deletion of a message in a sequence diagram.
3. Modifications of attributes: These changes include the attribute type, scope (public, private, or protected) or visibility (static, virtual, etc.).

4. **Modifications of methods:** These changes include the method return-type, scope, visibility, or number of parameters.
5. **Modifications of method parameters:** These changes include parameter type or direction changes (in, out or inout).
6. **Modification of class relationships:** These changes include the relationship type (inheritance, aggregation or association) or target class.

Assuming that the UML design diagrams are generated before the code implementation and assuming that the UML diagrams provides a reasonable level of details (class attributes, class methods, class relationships, methods parameters and types, etc.), the regression test selection algorithm classifies the system/subsystem regression test cases based on the modifications detected from the UML models. The consequences of having an incomplete UML models will be discussed in the tool practical limitations (Section 5.1).

1.4 Basic Definitions

In this section we define some of the most commonly used terms related to impact analysis and regression testing.

1.4.1 The Nature of Software Changes

Software requirements are continually changing to accommodate user expectations, new operational environment needs and new feature requirements [Jalote 96]. The typical reasons for software design changes can be: requirements changes, design trade-offs and elaboration, interface changes, scope and visibility issues, performance timing, sizing issues, and feedback from prototypes. Typical reasons for program changes include bug fixes, algorithm-coding adjustments,

arithmetic-precision modifications, data-structure modifications, initialization modifications, control and sequence changes, and parameters changes.

1.4.2 The Software Design Process

The software design process [Pfleeger 98] includes the following activities:

- Requirements analysis and definition.
- System design.
- Implementation.
- Unit and integration testing.
- System testing.
- System delivery.

The software design process starts with the software requirements. Once the requirements are defined, a system design is created to meet the specified requirements. The system design includes a complete description of the functions and interactions involved. When system design is approved, the overall design is used to generate the designs of the individual subsystems involved. The code implementation follows the approval of the different system components. The individual pieces of the code are unit tested before they can be linked together.

Once we are convinced that the different pieces work as desired, we put them together and make sure that they work properly when joined with other modules. This second testing phase is often referred to as integration testing. The final testing phase, called system testing, involves a test of the whole system to make sure that the functions and interactions specified initially have been

implemented properly. At last the final product is delivered. The software process is a recursive process, during this process part or all of the above activities may be performed as needed.

1.4.3 Inheritance and Polymorphism

Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and overriding or embellishing these with capabilities the new class requires [Deitel+98]. Software reusability saves time in program development and encourages the reuse of proven and debugged high-quality software, thus reducing problems after a system becomes functional. When creating a new class, instead of writing completely new data members and member functions, the programmer can designate that the new class is to inherit the data members and member functions of a previously defined base class.

Polymorphism is one of the most important capabilities of O-O languages like C++ [Deitel+98]. It is the ability of different classes related by inheritance to respond differently to the same message. The message sent to many different types of objects takes on “many forms”, hence the term polymorphism. Polymorphism is implemented via virtual functions. Polymorphism enables us to write programs in a general fashion to handle a wide variety of existing and yet-to-be-specified related classes.

1.4.4 The Message Sequence Specification (MgSS)

The MgSS [Hajla 96] is an important concept as it can be derived early in the design process from UML sequence diagrams. It defines the inter-class

interactions by specifying the order in which messages are sent out from one object to another. It can be used to detect call sequence changes in sequence diagrams by comparing the corresponding MgSS between two versions of the design models. The regular expression formalism [Linz 97] is used to describe the MgSS.

1.4.5 Impact Analysis

Impact analysis estimates what will be affected in a software system if a proposed change is implemented. The impact analysis information can be used for planning changes, accommodating certain types of software changes and tracing through the effects of changes. It also provides visibility into the potential effects of changes at an early stage before the changes are implemented and thus aids in the decision making process [Bohner+96].

There are two major technology areas for impact analysis: dependency (or source-code) analysis and traceability analysis. Both dependency analysis and traceability analysis have many supporting methods and tools.

Dependency analysis: involves examining detailed dependency relationships among program entities (variables, logic, modules, etc). It provides a detailed evaluation of low-level dependencies in code. It is totally based on source-code analysis and depends on the size and complexity of the code. Impact analysis techniques that support these sets of dependency include data-flow analysis, control flow analysis, program slicing, test-coverage analysis, cross referencing, and browsing.

Traceability analysis: involves examining dependency relationships among all types of software artifacts. It addresses the impact analysis from a broader perspective by tracing the relationships among software artifacts so that side effects from one artifact to the other can be identified. Traceability relationships are often represented in a graph structure, which is amenable to navigation with a hypertext system.

1.4.6 Object Oriented Testing

The object-oriented paradigm [Binder 99] presents a unique blend of powerful constructs, bug hazards, and testing problems. This is an unavoidable result of the encapsulation of operations and variables into a class, the variety of ways a system of objects can be composed, and the compression of complex runtime behavior into a few simple statements. Some essential features of O-O languages pose new fault hazards:

1. Dynamic binding and complex inheritance structures create many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage.
2. Objects preserve state, but state control is typically distributed over an entire program. State control errors are likely.

1.4.7 System and Subsystem Testing

A system is composed of subsystems. A subsystem is a collection of classes that provides a public interface to other subsystems. It may consist of a small cluster of a few classes or a large hierarchy of source code directories with

thousands of source code directories with thousands of classes containing millions of lines of code. A subsystem [Binder 99] has the following characteristics:

- It is executable and re-testable as a whole, whatever its size.
- It has parts that can be tested in isolation.
- It implements a cohesive set of responsibilities.
- It does not provide all of the functionality of the application under test.

Subsystem testing is a precondition for system testing. It is a practical necessity to keep scope and complexity at a manageable level.

1.4.8 Regression Testing

Regression testing is a necessary part of an effective testing process. It is typically done during iterative development. A regression test case [Pfleeger 98] is a test case that has passed on a previous version of a system and which is expected to pass when rerun on a modified version. A regression test suite is composed of regression test cases. A test case that has previously passed but no longer passes reveals a regression fault.

Considerations for reducing a test suite: A full regression test suite includes every baseline test case. Time and cost may preclude running such a test suite, in which case a reduced regression test suite can be selected, supporting selective regression testing. There are two approaches for reducing the size of the regression test suite: safe reduction and unsafe reduction [Binder 99]. A safe regression test suite consists of all the tests that could possibly exhibit different output when run on the modified version. An unsafe reduction technique may omit test cases that might reveal a regression fault.

1.4.9 Traceability

Traceability is a technique used to link together the user requirements, the analysis and design models, the implementation and the test cases. It is designed to solve what we call the “requirement traceability problem”, that is the ability to trace between software artifacts generated and modified during the software product life cycle [Bohner+96]. Traceability analysis identifies the affected software entities using their traceability relationships. The main objectives of traceability are:

- To improve the requirement clarity.
- To reduce the requirement omissions.
- To ensure test coverage of requirements during testing.
- To support the regression test activities during the maintenance phase.

1.5 Related Work

In this section we briefly review the work done in the area of regression test selection and impact analysis.

Wilde and Huitt [Wilde+92] analyze the problems of dynamic binding, object dependencies, dispersed program structure, control of polymorphism, high-level understanding, and detailed code understanding. The authors also provided recommendations for possible tool support, particularly using the concepts of dependency analysis, external dependency graphs, and cluster methodologies.

Croker and Mayhauser [Croker+90] addressed problems relating to class hierarchy changes, class signature changes, polymorphism, high level understanding and detailed code understanding. The authors then proposed a set

of tools to help solve some of the problems. The tools provide information collection, storage, analysis, inference, and display capabilities.

Kung et al. [Kung+94] addressed the regression-testing problem for C++ programs. They described the impact of code change on classes and how to identify them. They [Kung+95] initially defined a class firewall to enclose all the classes affected by changes to one or more classes and presented a test order generation algorithm that provides a cost-effective test sequence of components that serves as a road map for a tester to retest classes in the class firewall. The results can be used to conduct cost-effective regression testing of classes at the class unit level and the class integration level.

Kung et al. [Kung+96] also proposed a change identification algorithm based on a combined control-flow and data-flow representation. In these references, the authors did not consider the case when polymorphism and dynamic binding are involved and did not mention how to select test suites for effective regression testing.

Abdullah and White [Abdullah+97] also proposed an approach based on the firewall concept. This approach identifies the different effects of both specification and code changes on the firewall construction, as well as the effects of polymorphism and dynamic binding. In this approach, after the construction of a class firewall, the polymorphic part of the object relationship diagram of the system is analyzed. Then the change information and object relation diagram are utilized to construct the object firewall for the part of the system that has changed and the affected parts. The information generated from the object firewall is then

applied back to the class firewall, and corresponding modifications and simplifications made in that firewall object. This strategy combines and utilizes both the compile and run-time information in constructing the class firewall when polymorphism is involved. The authors also defined four levels of test case reuse and indicated the required test level of unit and integration testing with respect to the code or specification changes.

Rothermel and Harrold [Rothermel+94] presented a technique for selective retest that handles O-O software. The algorithm constructs dependency graphs for classes and application programs, and uses these graphs to determine which tests in an existing test suite can cause a modified class or program to produce different output than the original. The algorithms apply to C++ application programs, classes, and derived classes. They [Rothermel+99] also presented another selection technique that addresses the regression test selection problem for C++ software. The technique constructs control flow graphs (CFG) instead of dependence graphs for classes and programs that use classes. It uses those representations to select tests, from an existing test suite, that execute code that has been changed for a new version of the software. Comparing with construction of program dependence graphs, this algorithm is more efficient since construction of control flow graphs is less complex. This technique has several advantages: it can be fully automated, it operates inter-procedurally, and it performs test selection for C++ applications programs, classes, and derived classes.

Luiu and Robson [Luiu+93] proposed a model of a regression test database (RTD) that emphasizes configuration management, traceability and

change impact analysis of data used in regression testing. Then they described the SEMST, the prototype system that manages all versions of specifications, test cases, and programs, as well as control relationships between these components.

Hartmann and Robson [Hartmann+89] examined several regression testing strategies, including methods for capturing the program portions that may be affected by maintenance modifications to a program. Leung and White [Leung+91] conducted a similar study using a formally defined cost model. They used data flow analysis and program understanding models.

Laski and Szermer [Laski+92] describe an algorithm for identifying the affected parts in program maintenance. The algorithm is based on differentials between the control flow graphs [Rothermel+94] of the original and the modified program.

Hajla [Hajla 96] presented a methodology for conducting and managing the regression testing activities of O-O software during an iterative incremental development process. The methodology is based on three concepts: traceability, the regular expression formalism and the class firewall. Traceability links together the user requirements, analysis, design and implementation models and test cases. The regular expression formalism models the class specification behavior and the interactions between classes using the regular expression formalism. Two new concepts were introduced: the Message Sequence Specification (MgSS) and the Method Sequence Specification (MtSS). The MgSS describes the order among methods supported by different objects. The MtSS of a class indicates the order in which methods can be invoked for the correct behavior of instances of the class.

For each class the MtSS and MgSS define the intra and inter class interactions. The class firewall models the relationship between classes such as aggregation, inheritance or association.

Most of the approaches for regression testing of O-O software are code based which focuses on the changes introduced to the code and their effects on the relevancy of test cases. However, research has already established that the exclusive focus on code is inadequate and that modifications that involved the other life-cycle work products, such as analysis and design models, should be considered as well. Selective retest strategies focus on the relationship between the components affected by the changes and the test cases to decide its relevance and whether it should be used to retest the program.

The impact analysis based on the design models generated in the analysis and design phase is a new area under research. Hajla [Hajla 96] addressed the impact analysis based on the analysis and design models however there were many limitations to his research:

1. His research only addresses the problem of identifying changes at a class level. Changes due to the modification of the class interactions, such as adding or deleting relationships, are not identified.
2. Class attribute changes (type, scope, visibility) and methods signature changes (scope, visibility, number of parameters, type of parameters, direction of parameters, etc.) cannot be detected.

3. The proposed regression test selection algorithm is based on the changes to the class specification only, any other changes due to class interactions or subsystem interactions are not taken into consideration.
4. Guard condition handling: The execution path in a sequence diagram or a test case may change if a guard condition changes. This results in changes in both the test case and the sequence diagram. This issue was not handled, as the MgSS representation did not include the guard conditions.

In this thesis, we will address issues 1-3 from the previous list through the use of UML models that provide a high level view of the proposed software changes. The guard condition handling will not be covered in this research.

1.6 Thesis Organization

This chapter provides some basic definitions to some of the most commonly used terms and describes the motivation, contributions and scope of this research. Chapter 2 defines the regression test selection problem in terms of the information needed to categorize the different test cases and describes the types of changes that can be detected from UML models and their impact on test cases. A new methodology to analyze the different types of UML models is also proposed. A host of new algorithms is introduced in this chapter to address the problem of detecting UML models design changes and regression test selection. Chapter 3 introduces the Regression Test Selection Tool (RTSTool) developed to prove the viability of our methodology. The tool description, main features, design and implementation are provided in this chapter. Chapter 4 is a case study based on a real industrial system. The test cases for this system are divided into

two categories: functional and non-functional test cases. The impact analysis results and regression test selection results are also included in this chapter. Chapter 5 provides the conclusion, limitations and outlines possible future work.

Chapter 2 - Analysis, Design and Algorithms

This chapter is divided into two main parts. The first part defines the impact analysis and regression test selection problem and describes the types of changes that can be detected from UML models and their impact on test cases. Two issues are also discussed in regards to attribute changes and identifying methods in sequence diagrams. Then a design model is proposed to extract the needed information from the UML models. The design model is divided into three subsystems: Class Diagram subsystem, Use Case Diagram subsystem and Regression Test Suite subsystem. The first two subsystems model and represent the UML Class diagrams and Use Case diagrams, respectively. The third subsystem models and stores the regression test suite.

The second part introduces a host of algorithms that address the problems of comparing two versions of UML models, performing basic consistency checks and regression test selection. Our goal is to derive algorithms at a logical level based on our proposed subsystems class diagrams.

In the derivation of the algorithms, we use Object Constraint Language (OCL) in conjunction with the assignment operator and control flow structures. OCL [Warmer+99] allows us to write algorithms at a reasonable high level of details that is easier to understand. This is achieved through the use of OCL navigation capabilities to navigate between packages and classes. OCL also provides a rich set of collection types to store different types of objects.

2.1 Problem Definition

As shown in Figure 1, this section addresses a two-fold problem. The first problem is concerned with detecting the design changes between two different versions of UML models. The first version of the system includes both the UML design models and the source code whereas; the second version includes only the design models. This is based on the assumption that the source code is usually implemented after approving the impact analysis results of the proposed changes. Detecting these changes requires representing the information collected from the models into an analyzable and easy-to-navigate form.

Figure 1: Problem Definition

The second aspect of this problem is to use the information about the design changes to categorize the different regression test cases. The test cases are categorized into one of three categories: obsolete, retestable or reusable based on their reusability with respect to the new version of the UML models.

This section is divided into three parts. In the first part we describe the possible types of changes that can be detected from the UML models. In the

second part we discuss the impact of these changes on the regression test selection activity. In the third part we discuss two issues related to attribute changes and identifying methods used in sequence diagrams.

2.1.1 Changes in Models

As described in Section 1.3, we will consider three types of UML models: class diagrams, sequence diagrams and use case diagrams.

Two versions of a given class diagram are compared to detect the sets of added, changed and deleted attributes, methods, relationships, and classes. The following are the rules used to detect the different types of changes in a class diagram:

- **Added/Deleted attribute:** An added/deleted attribute is an attribute that does not exist in the original/modified version of a given class but exists in the modified/original version of this class.
- **Added/Deleted method:** Assuming that a method is uniquely identified in a class by its name and number of parameters, an added/deleted method is a method that does not exist in the original/modified version of a given class but exists in the modified/original version of this class. Figure 2 shows an example that explains the previous statement. In this Figure, two versions of some class, A, are given. Method `M1(Int, Float)` in version 2, has been classified as an added method because it has an additional parameter of type `Float` compared to method `M1(Int)` in version 1. In a similar manner, method `M2(char, string)` in version 2, has been classified as an added method because it has an additional

parameter of type `String` compared to method `M2 (char)` in version 1.

On the other hand, method `M2 (char)` in version 1, has been classified as a deleted method because it has a different number of parameters than that of method `M2 (char, string)` in version 2.

- **Added/Deleted relationship:** A relationship that does not exist in the original/modified version between two given classes but exists in the modified/original version.
- **Added/Deleted class:** Assuming that class names are unique within one class diagram, an added/deleted class is a class that does not exist in the original/modified version of the class diagram but exists in the modified/original version.
- **Changed attribute:** An attribute that exists in both versions of a given class but with a different scope, type, or visibility.
- **Changed method:** A method that exists in both versions of a given class but with a different scope, return-type, visibility or parameter list. The method parameters are also compared to detect any changes in the parameters type or direction. Referring to the example shown in Figure 2, method `M1 (Float)` in version 2, has been classified as a changed method because it has the same number of parameters as method `M1 (Int)` in version 1 but with a changed parameter type.
- **Changed relationship:** A relationship that exists in both versions between two given classes but with a different type (i.e., aggregation, association, inheritance).

- **Changed class:** A class that exists in both versions of the class diagram but with an added/changed/deleted attribute, method or relationship.

Figure 2: Method Changes

Assuming that use cases are unique within one use case diagram, two versions of a given use case diagram are compared to detect the sets of added, changed and deleted use cases. Similarly, assuming that each use case in the use case diagram corresponds to one sequence diagram, two corresponding sequence diagrams from the two versions of sequence diagrams are compared to detect the sets of added, changed and deleted methods. The following are the rules used to detect the different types of changes in a use case diagram:

- **Added/Deleted use case:** A use case that does not exist in the original/modified version of the use case diagram but exists in the modified/original version.
- **Added/Deleted method:** A method that does not exist in the original/modified version of a given sequence diagram in terms of the method name, class name, number of parameters and types of parameters

(Section 2.1.3.2 discusses how parameter types are determined) but exists in the modified/original of the sequence diagram.

- **Changed use case:** A use case that exists in both versions of the use case diagram but with a changed sequence diagram. A changed sequence diagram has an added/changed/deleted method.
- **Changed method:** A method that exists in both versions of a given sequence diagram but with a changed return-type or parameter type (Section 2.1.3.2 discusses how return-types and parameter types are determined).

The class diagram of a system includes all the methods defined for the system whereas, the sequence diagrams may use some or all of the methods defined in the class diagram. Therefore, the sets of added, changed and deleted methods detected from the use case diagram subsystem should be subsets, if not the same sets, of the sets of added, changed and deleted methods detected from the class diagram subsystem. Consequently, if a method exists in one of the class diagram sets of methods but not in the corresponding use case diagram set of methods (because it is not used), this method is said to be an *unused defined method* and is added to the set of `unusedDefinedMethods`. On the other hand, if a method exists in one of the use case diagram sets of methods but not in the corresponding class diagram set of methods, this method is said to be an *undefined invoked method* and is added to the set of `undefinedInvokedMethods`.

To verify the consistency of the changes introduced to the UML models and to detect the sets of `unusedDefinedMethods` and `undefinedInvokedMethods` between the class diagram and the use case/sequence diagram, the sets of added, deleted and changed methods detected in comparing two versions of class diagrams are compared against the corresponding sets of added, deleted and changed methods detected in comparing two versions of use case diagrams.

2.1.2 Issues

Two issues are discussed in this section. The first issue addresses attribute changes and how to account for these changes in regression test selection. The second issue addresses the lack of information collected from the sequence diagrams; namely, methods' return and parameters types.

2.1.2.1 Attribute Changes

When a class attribute is changed (due to a changed type, scope or visibility), the methods that "use" this attribute may require some changes to adapt to the attribute change. Therefore, we need to identify those methods that use class attributes in order to classify them as changed methods when the attributes they use are changed.

Assuming that the design by contract approach is used to specify the methods pre- and post conditions in OCL, the methods post conditions are examined to determine the set of methods that uses a given attribute. If a given attribute appears in a method post condition, this may indicate the potential modification of the attribute inside that method. The following example from the

case study described in chapter 4, illustrates the situation where a method changes an attribute internally and it can be detected from the method post condition:

```
Helper::agentTableDownload(globalTable) : Void
pre: GlobalDiscoveryAgent::globalInformationTable->notEmpty
post: Helper::globalTable->notEmpty and sequenceNumber =
sequenceNumber@pre +1
```

In the above example the sequenceNumber attribute of the Helper class is changed internally as a result of invoking the Helper::agentTableDownload operation.

If the design by contract approach is not used to specify the methods pre- and post conditions, then a simple source code analysis (using any available search engine) can be used to walk through the code and identify the methods that use a given attribute.

2.1.2.2 Identifying Methods in Sequence Diagrams

The amount of information that can be detected from sequence diagrams is limited compared to the information obtained from class diagrams. For example, a method in a given class diagram is defined in terms of the method name, return-type, scope and visibility and parameters. Therefore, our approach is to couple the methods used in the sequence diagrams with the methods declared in the class diagram.

As shown in Figure 3, six simple examples are drawn to illustrate these problems and our strategy to solve them. The first three examples illustrate the method return-type problem whereas the last three examples illustrate the method identification problem.

Figure 3: Identifying Methods in Sequence Diagrams

In the first example, method $M1()$ return-type can be detected directly from the class diagram (m is a `Float` in this case). Similarly, in the second example, method $M2(b, c)$ return-type can also be detected from the class diagram as it is linked to attribute $A3$ type (`Float` in this case), therefore, parameters b and c are of type `int`. In the third example, method $M2(A1, A2)$ uses attribute names. The attributes ($A1$ and $A2$) types can be determined from the class diagram and consequently the method can be identified ($M2(\text{Int}, \text{Int})$ in this case) and the method return type f is `Float` in this case. The forth example, method $M2(b, m)$, uses parameters m and b which were used in a previous message. Having identified the type of parameters m and b (`int` is this case), this method can be identified ($M2(\text{Int}, \text{Float})$ in this case) and the method return type g is a `Float` in this case. The last example, is a worst case situation where the method $M2(b, d)$ is overloaded with different

parameter types, the method does not use attributes as parameters and one of the parameters it uses was not identified in a previous message (parameter d). The exact method (`M2 (Int, Int)` or `M2 (Int, Float)`) cannot be identified in this case. This clearly limits the amount of information that can be detected from the sequence diagrams and limits the regression test selection accuracy.

2.1.3 Impact on Test Cases

The results obtained from comparing two versions of class diagrams and two versions of use case diagrams/sequence diagrams are used to classify the regression test cases. A test case can be classified in one of three categories:

- **Obsolete:** A test case that tests an invalid sequence of messages (based on the modified sequence diagrams).
- **Retestable:** A test case that contains a changed method and needs to be rerun.
- **Reusable:** A test case that is still valid but need not be rerun, as it does not contain any changed methods.

Since sequence diagrams [Binder 99] are the only UML diagrams that describes how a sequence of messages exchanged among a set of objects can accomplish some result of interest and since test cases describes sequence of messages to test certain functionalities of the system, therefore, to derive traceability information between the design models and test cases, we need to compare test cases and sequence diagrams. To facilitate the comparison process, the sequence of messages in test cases and sequence diagrams are represented using regular expressions [Linz 97].

Our regression test selection activity considers two types of test cases: functional and non-functional test cases:

- Functional test cases [Bruegge+99] aim at identifying differences between the observed functionality and the expected one (e.g., as defined by use cases and sequence diagrams).
- Non-functional test cases aims at identifying the user visible aspects of the system that are not directly related with the functionality of the system. Non-functional test cases do not represent complete use case scenarios but only parts of them. Non-functional test cases include test cases to test the performance and reliability of the system.

In the case of functional test cases, a test case is obsolete if it contains an invalid sequence of methods based on the modified version of sequence diagrams. A retestable test case is a test case that contains a valid sequence of messages and includes one or more changed methods. A reusable test case is test case that contains a valid sequence of messages and no changed or deleted methods.

The situation is different for non-functional test cases as they may not correspond to sequence diagram: we cannot verify the sequences of method calls they contain using sequence diagrams. A non-functional test case is obsolete if it contains a deleted method, which means that the test case cannot be executed because of this deleted method. A retestable test case is a test case that contains a changed method and a reusable test case is test case that does not contain any changed or deleted methods.

2.2 Outline of the Proposed Solution

Figure 4 shows the major components of the proposed solution. It consists of three subsystems: ClassDiagram, UseCaseDiagram and RegressionTestSuite.

Figure 4: Packages and their Relationships

As their name implies, the first two subsystems handle the different types of UML diagrams considered in this thesis, while the last subsystem handles the regression test suite analysis.

ClassDiagram and **UseCaseDiagram** subsystems compare two versions of class diagrams and use case diagrams, and provide the comparison results to another component, **ConsistencyCheck** which performs consistency checking between the class diagram and the use case diagram of the same version.

The **RegressionTestSuite** subsystem loads and stores the different test cases that constitute the test suite. The regression test selection algorithm merges the different sets of methods obtained from the class diagrams comparison

with the sets of methods obtained from the use case diagrams comparison. These sets of methods along with the modified version of sequence diagrams are then used to classify the different test cases. This is achieved by comparing the test cases' sequence of messages against the modified version of sequence diagrams using the pattern matching algorithms (Section 2.3.4.3.3).

The next subsections describe these subsystems, their roles, and responsibilities. A number of methods are not shown in the subsystems' class diagrams for simplicity. For a complete list of methods associated with each class, refer to the RTSTool Data Dictionary provided in Appendix B.

2.2.1 The ClassDiagram Subsystem

The `ClassDiagram` subsystem (Figure 5)* is responsible for loading, storing and processing class diagrams. The different classes that constitute the class diagram are stored along with their corresponding methods and attributes as well as the relationships between them. The pair of attributes (name, version) uniquely identifies one class diagram.

The `ClassDiagramChanges` class is created to handle the comparison process between two versions of a class diagram. It is associated with two `ClassDiagram` classes and stores the sets of added, changed and deleted attributes, classes and methods generated from comparing these two class diagrams. Assuming that classes are unique within one class diagram, the different classes from these two class diagrams are compared to detect the sets of added, changed and deleted attributes, classes and methods. This is achieved by

using the findClassDiagramChanges algorithm as described in Section 2.3.1 and based on the set of rules defined in 2.1.1.

Figure 5: ClassDiagram Subsystem

* Classes ClassDiagram, Class, Relationship, Attribute, DefinedMethod, and FormalParameter are part of what is called the UML class diagram metamodel [Booch+99].

The relationships between the attributes of a class and their access methods (methods that “uses” this attribute) are modeled by the association `accessMethods`. The attribute access methods will be determined based on simple code analysis of the original version of the UML models, as they cannot be determined from the UML models.

2.2.2 The UseCaseDiagram Subsystem

The `UseCaseDiagram` subsystem (Figure 6) is responsible for loading, storing and processing use case diagrams. The different use cases that constitute the use case diagram are stored along with and their corresponding sequence diagrams.

The `UseCaseDiagram` class stores the use case diagram name, version and all the use cases in the use case diagram. The pair of attributes (name, version) uniquely identifies one use case diagram. The `UseCase` class represents each use case in the use case diagram and stores the use case name and sequence diagram associated with the use case as a regular expression (attribute `MgSS`). The sequence diagram is stored as a regular expression in the `MgSS` format described before (Section 1.4.4) to simplify the sequence diagrams comparison process. The `InvokedMethod` class represents every method in a sequence diagram and stores the name, return-type (when available) and class name of the method. The `ActualParameter` class represents the different method parameters and stores the type and value of each method parameter.

The `UseCaseDiagramChanges` class is created to handle the comparison process between two versions of a use case diagram. It is associated

with two UseCaseDiagram classes and stores the sets of added, changed and deleted methods and use cases generated from comparing these two use case diagrams.

Figure 6: UseCaseDiagram Subsystem

Assuming that use cases are unique within one use case diagram, the different use cases and their corresponding sequence diagrams from these two use case diagrams are compared to detect the sets of added, changed and deleted methods and use cases. This is achieved by using the `findUseCaseDiagramChanges` algorithm as described in Section 2.3.2 and based on the set of rules defined in 2.1.1.

2.2.3 The RegressionTestSuite Subsystem

The `RegressionTestSuite` subsystem (Figure 7) is responsible for loading, storing and classifying the different regression test cases. Using the information obtained from the `ClassDiagram` and the `UseCaseDiagram` subsystems about the added, changed, and deleted methods. The regression test selection algorithm (Section 2.3.4) implemented in this subsystem, classifies the different regression test cases as being obsolete, retestable or reusable and generates the sets of obsolete, retestable and reusable test cases.

The `RegressionTestSuite` class stores the regression test suite name, version and associated test cases. The pair of attributes (name, version) uniquely identifies one regression test suite. The `TestCase` class represents a test case and stores the test case name, type and version in addition to the test case sequence of messages as a regular expression [Linz 97]. The `InvokedMethod` class used by the `UseCaseDiagram` subsystem is also used here to represent the different methods in a test case sequence of messages. Each method is represented by its name, return-type (when available) and class name. Similarly, the `ActualParameter` class used by the `UseCaseDiagram` subsystem is

used here to represent the different method parameters. Each parameter is represented by its type (when available) and value. The `TestSuiteClassification` class is associated with each test case and stores the test case classification (obsolete, retestable or reusable). The sets of obsolete, retestable and reusable test cases are generated based on this classification and stored with the `RegressionTestSuite` class.

Figure 7: RegressionTestSuite Subsystem

2.3 Algorithms

The purpose of this section is to introduce the algorithms that will be used to analyze the different UML models as described in Section 2.2.

This section is organized as follows. First, we detect the class diagram changes: added, changed and deleted attributes, classes, and methods by comparing the class diagrams from the original and modified version as described in Section 2.3.1. As test cases are composed of sequences of messages - no attributes are involved - hence the need to map the attribute changes to their corresponding access methods as described in Section 2.1.3.1. Then we compare the use case diagrams and their corresponding sequence diagrams from the original and modified versions to detect the added, changed and deleted methods and use cases as described in Section 2.3.2.

The comparison results between the different UML diagrams are then compared to detect the sets of undefined invoked methods and unused defined methods as described in Section 2.3.3.

Finally, using the information obtained from the class diagrams comparison and the use case diagram comparison, the regression test selection algorithm classifies the regression test cases as being obsolete, retestable or reusable as described in Section 2.3.4.

In deriving the algorithms, the OCL *Set* collection type was used because it contains elements without duplicates. Some of the OCL *Sets* operations used are:

<code>set3 := set1 - set2</code>	<code>set3</code> , is the set of elements in <i>set1</i> , which are not in <i>set2</i> .
<code>set3:= set1->intersection(set2)</code>	<code>set3</code> is the set containing all the elements that are in both <i>set1</i> and <i>set2</i> .
<code>set3 := set1->union(set2)</code>	<code>set3</code> is the set containing all elements in both <i>set1</i> and <i>set2</i> .

To facilitate the comprehension of the most complex algorithms, their sequence diagrams describing the flow of messages between the different RTSTool objects will be provided.

2.3.1 Class Diagram Changes Algorithm

The class diagram changes algorithm compares two class diagrams to detect the sets of added, changed and deleted attributes, classes and methods. It is based on the class diagram representation described earlier in this chapter. It compares the different classes from the original and modified versions of the class diagram. The mapping between attributes and their corresponding access methods is also handled in this algorithm along with the method inheritance feature.

The `findClassDiagramChanges` procedure is located in the `ClassDiagramChanges` class (Figure 5). Assuming that `classDiagram1` is the original class diagram and `classDiagram2` is the modified class diagram, the sequence of messages used to generate the sets of added, changed and deleted attributes, classes and methods is shown in Figure 8.

The algorithm starts by generating the sets of all attributes and all methods associated with each class diagram. Then the function `findChangedClasses`

is called to generate the set of changed classes by comparing the different corresponding classes from the two class diagrams. If a class has an added, changed, or deleted attribute, method or relationship, the class is considered changed and is added to the set of changed classes. Similarly, the `findChangedAttributes` and `findChangedMethods` functions are called to generate the sets of changed attributes and methods. Then the sets of all classes, all attributes and all methods from each class diagram are compared to detect the sets of added and deleted classes, attributes and methods. The added/deleted attributes, classes and methods are detected by subtracting the common attributes/classes/methods and the changed attributes/classes/methods from the `allAttributes/allClasses/allMethods` sets. Then, the procedure `handleAttributeChanges` updates the sets of `addedMethods` and `changedMethods` according to attribute changes and the mapping between attributes and methods that uses them. Similarly, the procedure `handleMethodInheritance` updates the sets of `changedMethods` and `deletedMethods` according to the deleted inherited methods.

The ClassDiagram Subsystem

Figure 8: Class Diagram Changes Algorithm Sequence of Messages

2.3.1.1 Input/Output

This algorithm has no inputs as it uses the two class diagrams associated with the `ClassDiagramChanges` class to perform the comparison process. Similarly, it has no outputs as it updates the sets of added, changed and deleted attributes, classes and methods associated with the `ClassDiagramChanges` class with the comparison results.

2.3.1.2 Algorithm

```
Procedure findClassDiagramChanges();  
var allAttributes1 : Set of Attribute;  
var allAttributes2 : Set of Attribute;  
var allMethods1 : Set of DefinedMethod;  
var allMethods2 : Set of DefinedMethod;  
var commonAttributes : Set of Attribute;  
var commonClasses : Set of Class;  
var commonMethods : Set of DefinedMethod;  
begin  
    // Construct the sets of all attributes and all methods for class diagram 1  
    foreach element C1 in classDiagram1.classes do  
        allAttributes1 := allAttributes1->union(C1.classAttributes);  
        allMethods1 := allMethods1->union(C1.classMethods);  
    end  
    // Construct the sets of all attributes and all methods for class diagram 2  
    foreach element C2 in classDiagram2.classes do  
        allAttributes2 := allAttributes2->union(C2.classAttributes);  
        allMethods2 := allMethods2->union(C2.classMethods);  
    end  
    // Find the sets of changed classes, attributes and methods  
    changedClasses := classDiagram2.findChangedClasses(classDiagram1.classes);  
    changedAttributes := classDiagram2.findChangedAttributes(changedClasses);  
    changedMethods := classDiagram2.findChangedMethods(changedClasses);  
  
    // ClassDiagram2 classes minus the common classes between the two class
```

```

// diagrams, minus the changedClasses is the set of added classes. The reverse
// operation gives us the set of deleted classes.
commonClasses := classDiagram1.classes->intersection(classDiagram2.classes);
addedClasses := classDiagram2.classes - commonClasses - changedClasses;
deletedClasses := classDiagram1.classes - commonClasses - changedClasses;

// ClassDiagram2 attributes minus the common attributes between the two class
// diagrams, minus the changedAttributes is the set of added attributes. The
// reverse operation gives us the set of deleted attributes.
commonAttributes := allAttributes1->intersection(allAttributes2);
addedAttributes := allAttributes2 - commonAttributes - changedAttributes;
deletedAttributes := allAttributes1 - commonAttributes - changedAttributes;

// ClassDiagram2 methods minus the common methods between the two class
// diagrams, minus the changedMethods is the set of added methods. The reverse
// operation gives us the set of deleted methods.
commonMethods := allMethods1->intersection(allMethods2);
addedMethods := allMethods2 - commonMethods - changedMethods;
deletedMethods := allMethods1 - commonMethods - changedMethods;

// Handle the class attribute changes by mapping the attribute changes to its
// corresponding access method.
handleAttributeChanges();

// Handle the method inheritance feature
handleMethodInheritance();
end

```

2.3.1.3 Internal Functions and Procedures

The internal functions and procedures used by the findClassDiagramChanges algorithm are included in this section.

2.3.1.3.1 Function findChangedClasses

Description: This is an internal function to detect the set of changed classes. It is used to construct and return the set of changed classes and is located in the

ClassDiagram class. A class is considered changed if it has a changed attribute, method or relationship.

In: var classes: Set of Class;

Out: Set of Class;

Function findChangedClasses(classes : Set) : Set;

var changedClasses : Set of Class;

begin

 // Compare the classes one by one to find the changed classes

foreach element C1 in classes **do**

foreach element C2 in this.classes **do**

if ((C1.getName() == C2.getName()) **and** (!C1.compareClasses(C2))) **then**

begin

 // The two classes are not the same

 changedClasses := changedClasses->including(C1);

break;

end

end

end

end

2.3.1.3.2 Function compareClasses

Description: This is an internal function to compare two classes. It compares two classes relationships, attributes and methods. It returns true, if the two classes are the same otherwise false. This function assumes that class names are unique within one class diagram.

Function compareClassRelationships(relationships : Set of Relationship) : Boolean;

 // This is an internal function to compare between two sets of relationships.

 // It is located in the Class class and returns true if the two class's sets of

 // relationships are the same otherwise false.

Function compareClassAttributes(attributes : Set of Attribute) : Boolean;

```
// This is an internal function to compare between two sets of attributes.
// It is located in the Class class and returns true if the two classes sets of
// attributes are the same otherwise false.
```

```
Function compareClassMethods(methods : Set of DefinedMethod) : Boolean;
```

```
// This is an internal function to compare between two sets of methods.
```

```
// It is located in the Class class and returns true if the two classes sets of
// methods are the same otherwise false.
```

```
In: var class:           Class;
```

```
Out:                Boolean;
```

```
Function compareClasses(class : Class) : Boolean;
```

```
begin
```

```
    if ((this.getName() == class.getName()) and
        (this.compareClassRelationships(class.classRelationships)) and
        (this.compareClassAttributes(class.classAttributes)) and
        (this.compareClassMethods(class.classMethods))) then
```

```
        return true;
```

```
    else
```

```
        return false;
```

```
end
```

2.3.1.3.3 Function findChangedAttributes

Description: This is an internal function to detect the changed attributes. It is used to construct and return the set of changed attributes and is located in the ClassDiagram class. This function assumes that attribute names are unique within one class.

```
In: var changedClasses:      Set of Class;
```

```
Out:                Set of Attribute;
```

```
Function findChangedAttributes(changedClasses : Set) : Set;
```

```
var changedAttributes : Set of Attribute;
```

```
begin
```



```

foreach element C1 in changedClasses do
  foreach element C2 in this.classes do
    if (C1.getName() == C2.getName()) then
      begin
        changedAttributes := changedAttributes->
          union(C1.findChangedClassAttributes(C2.classAttributes));
        break;
      end
    end
  end
end

```

2.3.1.3.4 Function findChangedClassAttributes

Description: This is an internal function to detect the changed attributes at the class level. It is used to construct and return the set of changed attributes and is located in the Class class.

In: var attributes: Set of Attribute;

Out: Set of Attribute;

Function findChangedClassAttributes(attributes : Set) : Set;

var changedAttributes : Set of Attribute;

begin

```

  foreach element A1 in this.classAttributes do
    foreach element A2 in attributes do
      if ((A1.getName() == A2.getName()) and (!A1.compareAttributes(A2))) then
        begin
          changedAttributes := changedAttributes->including(A1);
          break;
        end
      end
    end
  end
end

```

2.3.1.3.5 Function compareAttributes

Description: This is an internal function to compare two attributes. It returns true if the two attributes are the same otherwise false. It is located in the Attribute class. An attribute is considered changed if it has a changed type, scope, or visibility. This function assumes that attribute names are unique within one class.

In: var attribute: Attribute;

Out: Boolean;

Function findClassChangedAttributes(attribute : Attribute) : Boolean;

begin

```
    if ((this.getName() == attribute.getName()) and
        (this.getClassName() == attribute.getClassName()) and
        (this.getScope() == attribute.getScope()) and
        (this.getType() == attribute.getType()) and
        (this.getVisibility() == attribute.getVisibility())) then
        return true;
```

else

```
        return false;
```

end

2.3.1.3.6 Function findChangedMethods

Description: This is an internal function to detect the changed methods. It is used to construct and return the set of changed methods and is located in the ClassDiagram class. This function assumes that class names are unique within one class diagram.

In: var changedClasses: Set of Class;

Out: Set of DefinedMethod;

Function findChangedMethods(changedClasses : Set) : Set;

var changedMethods : Set of DefinedMethod;

```

begin
  foreach element C1 in changedClasses do
    foreach element C2 in this.classes do
      if (C1.getName() == C2.getName()) then
        begin
          changedMethods := changedMethods->
            union(C1.findChangedClassMethods(C2.classMethods));

          break;
        end
      end
    end
  end
end

```

2.3.1.3.7 Function findChangedClassMethods

Description: This is an internal function to detect the changed methods at the class level. It is used to construct and return the set of changed methods and is located in the Class class.

In: var methods: Set of DefinedMethod;

Out: Set of DefinedMethod;

Function findChangedClassMethods(methods : Set) : Set;

var changedMethods : Set of DefinedMethod;

var same : Boolean;

same := false;

begin

foreach element M1 in this.classMethods **do**

foreach element M2 in methods **do**

if (M1.compareMethods(M2)) **then**

begin

 // It is the same method

 same := true;

break;

end

```

    end
    if (!same) then
    begin
        // We went through all the methods and could not find the same method.
        changedMethods:= changedMethods->including(M1);
    end
    else begin
        // Found the method, reset the flag.
        same := false;
    end
end
end
end

```

2.3.1.3.8 Function compareMethods

Description: This is an internal function to compare two defined methods. It returns true if the two methods are the same otherwise false. It is located in the DefinedMethod class. A defined method is considered changed if it has the same name, class name, and number of parameters but a changed parent method name, return-type, scope, visibility, parameter type or parameter direction.

In: var method: DefinedMethod;

Out: Boolean;

Function compareMethods(method : DefinedMethod) : Boolean;

```

begin
    if ((this.getName() == method.getName()) and
        (this.getClassName() == method.getClassName()) and
        (this.getParentName() == method.getParentName()) and
        (this.getScope() == method.getScope()) and
        (this.getType() == method.getType()) and
        (this.getVisibility() == method.getVisibility()) and
        (this.methodParameters->size == method.methodParameters->size) and
        (this.compareMethodParameters(method.methodParameters))) then

```

```

        return true;
    else
        return false;
    end
end

```

2.3.1.3.9 Function compareMethodParameters

Description: This is an internal function to compare two methods parameters. It is located in the DefinedMethod class. It returns true if the two method parameters are the same otherwise false.

```

Function compareParameters(parameter : FormalParameter) : Boolean;
// This is an internal function to compare two formal parameters. It is located in
// the FormalParameter class and returns true if the two parameters are the same
// otherwise false.

```

In: var methodParameters: Set of FormalParameter;

Out: Boolean;

```

Function compareMethodParameters(methodParameters : Set) : Boolean;
var same: Boolean;
same := false;
begin
    // If the number of parameters in both sets is not equal return false.
    if (this.methodParameters->size != methodParameters->size) then
        begin
            return false;
        end
    // If both sets contain zero number of parameters, return true.
    else if ((this.methodParameters->size == 0) and (methodParameters->size == 0))
        then begin
            return true;
        end
    // Iterate through the two sets to find the changed parameters.
    foreach element P1 in this.methodParameters do
        foreach element P2 in methodParameters do

```

```

        if (P1.compareParameters(P2)) then
            begin
                // Same parameter
                same := true;
                break;
            end
        end
    end
    if (!same) then
        begin
            // The parameter exists in the first method but not in the second
            return same;
        end
    else begin
        // Reinitialize and check for the next parameter if any.
        same := false;
    end
end
return true;
end

```

2.3.1.3.10 Procedure handleAttributeChanges

Description: This is an internal procedure to handle the attribute changes (addition, deletion or modification). It is located in the `ClassDiagramChanges` class. The sets of `addedMethods` and `changedMethods` are updated according to the attribute changes and the mapping between attributes and methods that uses them. For example, if a new attribute is added to a class and one or more of the access methods associated with it are not marked as added (they existed in the previous version), then these methods are added to the set of added methods. On the other hand if an attribute is

changed or deleted, its set of access methods are added to the set of changed methods.

```
Procedure handleAttributeChanges();  
begin  
    // Added attributes case  
    foreach element A1 in addedAttributes do  
        if (A1.accessMethods->notEmpty) then  
            begin  
                foreach element IM in A1.accessMethods do  
                    addedMethods := addedMethods->union(IM.accessMethods);  
                end  
            end  
        end  
    end  
    // Changed attributes case  
    foreach element A1 in changedAttributes do  
        if (A1.accessMethods->notEmpty) then  
            begin  
                changedMethods := changedMethods->union(A1.accessMethods);  
            end  
        end  
    end  
    // Deleted attributes case  
    foreach element A1 in deletedAttributes do  
        if (A1.accessMethods->notEmpty) then  
            begin  
                changedMethods := changedMethods->union(A1.accessMethods);  
            end  
        end  
    end  
end
```

2.3.1.3.11 Procedure handleMethodInheritance

Description: This is an internal procedure to handle the inheritance feature. It is located in the ClassDiagramChanges class. If a deleted method has a parent

class method then the method in the parent class will be called instead of the deleted one. In this case, the deleted method in the child class is not considered deleted but is considered changed. This method is removed from the set of deleted methods and added to the set of changed methods.

```

Procedure handleMethodInheritance();
begin
    foreach element M1 in deletedMethods do
        if (M1.parentExists()) then
            begin
                // This method should be marked as changed not deleted. Remove the
                // method from the deleted set of methods and add it to the set of
                // changed methods.
                deletedMethods := deletedMethods->excluding(M1);
                changedMethods := changedMethods->including(M1);
            end
        end
    end

```

2.3.2 Use Case Diagram Changes Algorithm

This algorithm compares two use case diagrams to detect the sets of added, changed and deleted methods and use cases. It is based on the use case diagram representation described earlier in this chapter. The algorithm compares the different use cases and their corresponding sequence diagrams from the original and modified versions of the use case diagram. Assuming that each use case in the use case diagram corresponds to one sequence diagram. If a sequence diagram has a changed sequence of messages or an added, changed, or deleted method, the associated use case is considered changed and is added to the set of changed use cases. The `findUseCaseDiagramChanges` procedure is located

in the `UseCaseDiagramChanges` class (Figure 6). The sequence of messages used to generate the sets of added, changed and deleted methods and use cases is shown in Figure 9.

The UseCaseDiagram Subsystem

Figure 9: Use Case Diagram Changes Algorithm Sequence of Messages

Assuming that `useCaseDiagram1` is the original use case diagram and `useCaseDiagram2` is the modified use case diagram, the algorithm starts by generating the set of all methods associated with each use case diagram and their corresponding sequence diagrams. Then the function `findChangedUseCases` is called to generate the set of changed use cases by comparing the different corresponding use cases and their associated sequence diagrams from the two use case diagrams. If a sequence diagram has a changed sequence of messages due to an added, changed, or deleted method, the sequence diagram is considered changed. Consequently, the associated use case is considered changed and is

added to the set of changed use cases. Then, the `findChangedMethods` function is called to generate the set of changed methods. Finally, the sets of all methods and all use cases from the two use case diagram are compared to detect the sets of added and deleted methods and use cases. The added/deleted methods and use cases are detected by subtracting the common methods/use cases and the changed methods/use cases from the allMethods and the use case diagrams sets of use cases.

2.3.2.1 Input/Output

This algorithm has no inputs as it uses the two use case diagrams associated with the `UseCaseDiagramChanges` class to perform the comparison process. Similarly, it has no outputs as it updates the sets of added, changed and deleted methods and use cases associated with the `UseCaseDiagramChanges` class with the comparison results.

2.3.2.2 Algorithm

```

Procedure findUseCaseDiagramChanges();
var allMethods1 : Set of InvokedMethod;
var allMethods2 : Set of InvokedMethod;
var commonMethods : Set of InvokedMethod;
var commonUseCases : Set of UseCase;
Begin
    // Construct the set of all methods for use case diagram 1
    foreach element UC1 in useCaseDiagram1.useCases do
        allMethods1 := allMethods1->union(UC1.useCaseMethods);
    end
    // Construct the set of all methods for use case diagram 2
    foreach element UC2 in useCaseDiagram2.useCases do
        allMethods2 := allMethods2->union(UC2.useCaseMethods);
    end

```

```

// Find the common methods and use cases between the two associated use case
// diagrams.
commonMethods := allMethods1->intersection(allMethods2);

commonUseCases := useCaseDiagram1.useCases->intersection(
                                useCaseDiagram2.useCases);

// Find the sets of changed use cases and methods.
changedUseCases :=
                                useCaseDiagram2.findChangedUseCases(useCaseDiagram1.useCases);
changedMethods := useCaseDiagram2.findChangedMethods(changedUseCases,
                                allMethod2);

// UseCaseDiagram2 methods minus the common methods between the two use case
// diagrams, minus the changedUseCases is the set of added methods. The
// reverse operation is the set of deleted methods.
addedUseCases := useCaseDiagram2.useCases - commonUseCases - changedUseCases;
deletedUseCases := useCaseDiagram1.useCases - commonUseCases - changedUseCases;

// UseCaseDiagram2 methods minus the common methods between the two use case
// diagrams, minus the changedMethods is the set of added methods. The reverse
// operation is the set of deleted methods.
addedMethods := allMethods2 - commonMethods - changedMethods;
deletedMethods := allMethods1 - commonMethods - changedMethods;

end

```

2.3.2.3 Internal Functions

The internal functions used by the findUseCaseDiagramChanges algorithm are included in this section.

2.3.2.3.1 Function findChangedUseCases

Description: This is an internal function to detect the changed use cases. It is used to construct and return the set of changed use cases and is located in the UseCaseDiagram class. A use case is considered changed, if its corresponding sequence diagram has a changed sequence of messages or an added, changed or

deleted method. This function assumes that use case names are unique within one use case diagram.

In: var useCases : Set of UseCase;

Out: Set of UseCase;

Function findChangedUseCases(useCases : Set) : Set;

var changedUseCases : Set of UseCase;

begin

 // Compare the use cases one by one to find the changed use cases

foreach element UC1 in useCases **do**

foreach element UC2 in this.useCases **do**

if ((UC1.getName() == UC2.getName()) **and**

 (UC1.getMgSS() != UC2.getMgSS())) **then**

begin

 // The two use cases are not the same

 changedUseCases := changedUseCases->including(UC1);

break;

end

end

end

end

2.3.2.3.2 Function findChangedMethods

Description: This is an internal function to find the set of changed methods and is located in the UseCaseDiagram class.

In: var changedUseCases : Set of UseCase;

var allMethods: Set of InvokedMethod;

Out: Set of InvokedMethod;

Procedure findChangedMethods(changedUseCases : Set, allMethods : Set) : Set;

var changedMethods : Set of InvokedMethod;

begin

```

foreach element UC1 in changedUseCases do
    foreach element UC2 in this.useCases do
        if (UC1.getName() == UC2.getName()) then
            begin
                changedMethods := changedMethods->union(UC1.findUseCaseChanges(
                                                                    UC2.useCaseMethods, allMethods));
                break;
            end
        end
    end
end

```

2.3.2.3.3 Function findUseCaseChanges

Description: This is an internal function to find the set of changed methods at the use case level. It is located in the UseCase class. This function compares the set of methods passed-in with the use case set of methods and returns true if the two sets are the same, otherwise false.

In: var methods: Set of InvokedMethod;

var allMethods: Set of InvokedMethod;

Out: Set of InvokedMethod;

Procedure findUseCaseChanges(methods, allMethods : Set) : Set;

var changedMethods : Set of InvokedMethod;

var same : Boolean;

same := false;

begin

foreach element M1 in this.useCaseMethods **do**

foreach element M2 in methods **do**

if (M1.compareMethods(M2)) **then**

begin

 // It is the same method

 same := true;

```

        break;
    end
end
if (!same) then
begin
    // We went through all the methods and could not find the same method.
    if (!allMethods->includes(M1)) then
        changedMethods:= ChangedMethods->including(M1);
    end
else begin
    // Found the method, reset the flag.
    same := false;
end
end
end
end

```

2.3.2.3.4 Function compareMethods

Description: This is an internal function to compare two invoked methods. It returns true if the two methods are the same otherwise false. It is located in the `InvokedMethod` class. An invoked method is considered changed if it has the same name, class name, and number of parameters but a changed return-type or parameter type.

In: var method: `InvokedMethod;`

Out: `Boolean;`

Function `compareMethods(method : InvokedMethod) : Boolean;`

```

begin
    if ((this.getName() == method.getName()) and
        (this.getClassName() == method.getClassName()) and
        (this.getType() == method.getType()) and
        (this.methodParameters->size == method.methodParameters->size) and
        (this.compareMethodParameters(method.methodParameters))) then

```

```

        return true;
    else
        return false;
    end
end

```

2.3.2.3.5 Function compareMethodParameters

Description: This is an internal function to compare two invoked method parameters. It returns true if the two methods parameters are the same otherwise false. It is located in the InvokedMethod class.

```

Function compareParameters(parameter : ActualParameter) : Boolean;
// This is an internal function to compare two actual parameters. It is located in
// the ActualParameter class and returns true if the two parameters are the same
// otherwise false.

```

In: var methodParameters: **Set of ActualParameter;**

Out: **Boolean;**

```

Function compareMethodParameters(methodParameters : Set) : Boolean;
var same: Boolean;
same:= false;
begin
    // If the number of parameters in both sets is not equal return false.
    if (this.methodParameters->size != methodParameters->size) then
        begin
            return false;
        end
    // If both sets contain zero number of parameters, return true.
    else if ((this.methodParameters->size == 0) and (methodParameters->size == 0))
        then begin
            return true;
        end
    // Iterate through the two sets to find the changed parameters.
    foreach element P1 in this.methodParameters do
        foreach element P2 in methodParameters do

```

```

        if (P1.compareParameters(P2)) then
            begin
                // The parameter type and value are the same
                same := true;
                break;
            end
        end
    end
    if (!same) then
        begin
            // The parameter exists in the first method but not in the second
            return same;
        end
    else begin
        // Reinitialize and check for the next parameter if any.
        same := false;
    end
end
return true;
end

```

2.3.3 Consistency Check Algorithm

The ConsistencyCheck class (Figure 4) handles the consistency check process. It is an association class between the ClassDiagram and UseCaseDiagram subsystems. The sets of added, changed and deleted methods generated by the class diagram comparison algorithm are compared against the corresponding sets of methods generated by the use case diagram comparison algorithm. This algorithm generates two sets of methods: the set of unusedDefinedMethods (the methods that are defined in the class diagram but are not used in the sequence diagrams) and the set of

undefinedInvokedMethods (the methods that are used in the sequence diagrams but are not defined in the class diagram).

2.3.3.1 Input/Output

This algorithm has no inputs as it uses the associations with the ClassDiagram and UseCaseDiagram subsystems (Figure 4) to perform the consistency check. Similarly, this algorithm has no outputs as it updates the sets of undefinedInvokedMethods and unusedDefinedMethods associated with this class.

2.3.3.2 Algorithm

Procedure findInconsistentMethods();

Begin

```
// Compare the sets of added methods.  
checkMethodSets(useCaseDiagramChanges.addedMethods,  
                classDiagramChanges.addedMethods);  
  
// Compare the sets of changed methods.  
checkMethodSets(useCaseDiagramChanges.changedMethods,  
                classDiagramChanges.changedMethods);  
  
// Compare the sets of deleted methods.  
checkMethodSets(useCaseDiagramChanges.deletedMethods,  
                classDiagramChanges.deletedMethods);
```

end

2.3.3.3 Internal Functions and Procedures

The internal functions used by the findInconsistentMethods algorithm are included in this section.

2.3.3.3.1 Procedure checkMethodSets

Description: This is an internal procedure that compares two sets of methods. The first set is an `InvokedMethod` set generated from comparing two use case diagrams. The second set is a `DefinedMethod` set generated from comparing two class diagrams. The sets of undefined invoked methods and unused defined methods are generated from this procedure.

In: `var useCaseDiagramMethods: Set of InvokedMethod;`
`var classDiagramMethods: Set of DefinedMethod;`

```

Procedure checkMethodSets(useCaseDiagramMethods, classDiagramMethods : Set);
var exist: Boolean;
var invokedMethod : InvokedMethod;
exist := false;
begin
    // Generate the set of undefinedInvokedMethods.
    foreach element M1 in useCaseDiagramMethods do
        foreach element M2 in classDiagramMethods do
            if ((M1.getName() == M2.getName()) and
                (M1.getClassName() == M2.getClassName()) and
                (M1.getType() == M2.getType()) and
                (M1.methodParameters->size == M2.methodParameters->size)) then
                begin
                    invokedMethod := M2.convertToInvokedMethod();
                    if (M1.compareMethodParameters(invokedMethod.methodParameters))
                        then begin
                            // The method exists in the use case diagram set of methods
                            exist := true;
                            break;
                        end
                    end
                end
            if (!exist) then
                begin

```

```

        // The method exist in the use case diagram set of methods but not in
        // the class diagram set of methods, so add the method to the set of
        // undefinedInvokedMethods.
        undefinedInvokedMethods := undefinedInvokedMethods->including(M1);
    end
    else begin
        exist := false;
    end
end
// Generate the set of unusedDefinedMethods.
exist := false;
foreach element M3 in classDiagramMethods do
    foreach element M4 in useCaseDiagramMethods do
        if ((M3.getName() == M4.getName()) and
            (M3.getClassName() == M4.getClassName()) and
            (M3.getType() == M4.getType()) and
            (M3.methodParameters->size == M4.methodParameters->size) and
            (M3.compareMethodParameters(M4.methodParameters))) then
            begin
                invokedMethod := M3.convertToInvokedMethod();
                if (M4.compareMethodParameters(invokedMethod.methodParameters))
                    then begin
                        // The method exists in the class diagram set of methods
                        exist := true;
                        break;
                    end
            end
        end
    end
    if (!exist) then
        begin
            // The method exists in the class diagram set of methods but not in
            // the use case diagram set of methods, so add the method to the set of
            // undefinedInvokedMethods.
            unusedDefinedMethods := unusedDefinedMethods->including(M3);
        end
    else begin

```

```

        exist := false;
    end
end
end
end

```

2.3.3.3.2 Function convertToInvokedMethod

Description: This is an internal function to convert a DefinedMethod to an InvokedMethod. It is located in the DefinedMethod class.

```

Function convertToActualParameter(parameter: FormalParameter): ActualParameter;
// This is an internal function to convert a FormalParameter to an
// InvokedParameter. It is located in the FormalParameter class.
Function createMethodParameter(parameter: ActualParameter): Void;
// This is an internal function to create a method parameter. The parameter is
// then added to the method set of parameters.
// It is located in the InvokedMethod class.

```

In: var definedMethod: DefinedMethod;

Out: var invokedMethod: InvokedMethod;

```

Function convertToInvokedMethod(definedMethod : DefinedMethod) : InvokedMethod;
var IM : InvokedMethod;
begin
    IM.setName(this.getName());
    IM.setClassName(this.getClassName());
    IM.setType(this.getType());
    foreach element FM in this.methodParameters do
        IM.createMethodParameter(FM.convertToActualParameter());
    end
end

```

2.3.4 Regression Test Suite Algorithm

The purpose of this algorithm is to classify the different test cases that make the regression test suite. The sets of added, changed and deleted defined

methods detected from the class diagram subsystem are converted to invoked methods and merged with their corresponding sets from the use case diagram subsystem. These sets of methods are then used along with the modified version of the UML models sequence diagrams to classify the different test cases. The test cases are classified in one of three categories: obsolete, reusable or retestable test case. The `regressionTestSelection` procedure is located in the `RegressionTestSuite` class (Figure 7).

2.3.4.1 Input/Output

This algorithm has no inputs as it uses the associations with the `ClassDiagram` and `UseCaseDiagram` subsystems (Figure 4) to perform the regression test selection. Similarly, it has no outputs as it updates the sets of obsolete, retestable, and reusable test cases associated with this class.

2.3.4.2 Algorithm

```

Procedure regressionTestSelection();
begin
    // Classify the different regression test cases
    classifyTestCases();
    foreach element testCase in this.testCases do
        if (testCase.getClassification() == obsolete) then
            begin
                obsoleteTestCases := obsoleteTestCases->including(testCase);
            end
        else if (testCase.getClassification() == retestable) then
            begin
                retestableTestCases := retestableTestCases->including(testCase);
            end
        else if (testCase.getClassification() == reusable) then
            reusableTestCases := reusableTestCases->including(testCase);
        end
    end

```

```
end  
end
```

2.3.4.3 Internal Functions and Procedures

The internal functions and procedures used by the regressionTestSelection algorithm are included in this section.

2.3.4.3.1 Procedure classifyTestCases

Description: This procedure uses the mergeSets function to merge the deleted and changed sets of methods obtained from the ClassDiagram and UseCaseDiagram comparison algorithms. The merged sets of methods along with the set of all sequence diagrams (allMgSS) generated in this procedure are used to classify the different regression test cases. The test cases sequence of messages is checked using the function testCaseMatcher. A functional test case is classified as obsolete if it contains an invalid sequence of messages and a non-functional test case is classified as obsolete if it contains a deleted method. A retestable test case is a test case that contains a changed method (and contains a valid sequence of messages in the case of functional test cases). A reusable test case is a test case that does not contain any changed or deleted methods (and contains a valid sequence of messages in the case of functional test cases).

```
Procedure classifyTestCases();  
var classified: Boolean;  
var allMgSS : Set of String;  
var allChangedMethods : Set of InvokedMethod;  
var allDeletedMethods : Set of InvokedMethod;  
classified := false;  
begin  
    // Construct the set of allChangedMethods and allDeletedMethods
```

```

allChangedMethods := mergeSets(useCaseDiagramChanges.changedMethods,
                                classDiagramChanges.changedMethods);
allDeletedMethods := mergeSets(useCaseDiagramChanges.deletedMethods,
                                classDiagramChanges.deletedMethods);

// Construct the set of allMgSS from all the use cases in the modified use
// case diagram (useCaseDiagram2).
foreach element UC in useCaseDiagramChanges.useCaseDiagram2.useCases do
    allMgSS := allMgSS->including(UC.getMgSS());
end

foreach element testCase in this.testCases do
    // Check what kind of test case is this test case?
    if (testCase.getType() == "Functional") then
        begin
            // Invoke the test case matcher to check for valid sequence of
            // messages in the test case.
            if (!testCaseMatcher(allMgSS, testCase.getTestCase())) then
                begin
                    testCase.testCaseClassification.setClassification(obsolete);
                    classified := true;
                end
            end
        end
        if (!classified) then
            begin
                foreach element method in allDeletedMethods do
                    if (testCase.testCaseMethods->includes(method)) then
                        begin
                            testCase.testCaseClassification.setClassification(obsolete);
                        end
                    end
                end
                foreach element method in allChangedMethods do
                    if (testCase.testCaseMethods->includes(method)) then
                        begin
                            testCase.testCaseClassification.setClassification(retestable);
                        end
                    else begin
                        testCase.testCaseClassification.setClassification(reusable);
                    end
                end
            end
        end
    end
end

```

```

        end
    end
end
else begin
    classified := false;
end
end
end
end

```

2.3.4.3.2 Function mergeSets

Description: This function merges two sets of methods and return the resulting set. The first set is an invoked methods set from the UseCaseDiagramChanges class and the second set is a defined methods set from the ClassDiagramChanges class. DefinedMethod's are converted into InvokedMethod's because the test cases are defined in terms of invoked methods.

In: var useCaseMethods: Set of InvokedMethod;

var classMethods: Set of DefinedMethod;

Out: var allMethods: Set of InvokedMethod;

Function mergeSets(useCaseMethods, classMethods : Set) : Set;

var allMethods : Set of InvokedMethod;

begin

// Add the useCase methods

allMethods := allMethods->union(useCaseMethods);

// Convert the class diagram defined methods to invoked methods and add them to

// the set of allMethods.

foreach element DM in classMethods **do**

allMethods := allMethods->including(DM.convertToInvokedMethod());

end

end

2.3.4.3.3 Function testCaseMatcher

Description: This is an internal function to check if a test case contains a valid sequence of messages. The test case is compared against all the sequence diagrams in the modified use case diagram. It returns true if the test case has a valid sequence of messages otherwise it returns false. This function uses the standard pattern-matching library from JDK1.4 [Java].

In: var allMgSS: Set of String;

var testCase: String;

Out: Boolean;

Function testCaseMatcher(allMgSS : Set, testCase : String) : Boolean;

var MgSS: String;

var MgSSPattern : Pattern;

var testCaseMatcher : Matcher;

begin

foreach element MgSS in allMgSS do

MgSSPattern := Pattern.compile(MgSS);

// Create a matcher for this pattern using the test case

testCaseMatcher := MgSSPattern.matcher(testCase);

// A test case matches if it is found in any of the MgSS's

if (testCaseMatcher.find()) **then**

begin

return true;

end

end

return false;

end

Chapter 3 - Regression Test Selection Tool

The RTSTool is developed to aid in the impact analysis process and reduce the size of the regression test suite. The tool generates many useful statistics for the impact analysis process (e.g., added, changed and deleted attributes, classes, methods and use cases). The tool also classifies the different regression test cases based on the information detected from the UML artifacts. It is based on the models described in 2.2 and uses the algorithms defined in 2.3.

This chapter is organized as follows: first we provide the basic description of the tool in terms of inputs, outputs and main features. Then the basic tool design in terms of the tool package diagram, use case diagram, activity diagram are provided and finally the tool implementation is described.

3.1 Description and Main Features

Figure 10, describes the RTSTool in terms of its main functionality, inputs and outputs. The inputs to the RTSTool are the UML artifacts for the original and modified version of the system under test (class diagrams, sequence diagrams and use case diagrams) and the regression test suite. These diagrams and the test suite are stored in the tool database for future reuse.

Figure 10: RTSTool Description

The output of the tool depends on the functionality chosen by the user. In the case of regression test selection, the tool generates the sets of obsolete, retestable and reusable test cases. In the case of consistency checks, the tool generates the sets of unused defined methods and undefined invoked methods. The final report generated by the tool contains: the sets of added, changed and deleted attributes, classes, methods and use cases and the sets of obsolete, retestable and reusable test cases.

The design of the RTSTool tool focuses on adapting a number of features that are deemed necessary for such a multi purpose tool, these features are:

- Class diagrams and use case diagrams are loaded into the tool in the XMI format that can be generated from most of the available commercial case tools. This makes it easier and convenient for the user to use the RTSTool

as he can use his previously created diagrams by any case tool. The regression test suite is loaded into the tool in a simple text format as described in Section 3.2.

- A storage facility is provided for class diagrams, use case diagrams, and regression test suites along with their corresponding classes, sequence diagrams and test cases respectively. This makes the tool more convenient as the user's can reuse any of the previously loaded diagrams and test suites.
- The tool also performs a comparison of two class diagrams at the user request. Thus two class diagrams that are selected by the user are compared and the sets of added, changed and deleted attributes, classes, and methods are generated.
- Similarly, two use cases diagrams selected by the user can be compared and the sets of added, changed and deleted methods and use cases are generated. The tool also detects any changes in the message calling order in the sequence diagrams.
- In addition, the tool can perform consistency check of the comparison results at the user request. In this case two class diagrams are compared and two use case diagrams from the same design versions are compared. The consistency of the generated results is then analyzed and two sets of methods are generated.
- Finally, the tool can reduce the size of the regression test suite. The comparison results from the class diagrams and use case diagrams

comparison process along with the set of modified sequence diagrams are used to classify the different regression test cases from the selected regression test suite as obsolete, retestable or reusable test cases.

3.2 RTSTool Design

The RTSTool is designed based on the model described in 2.2 and it implements the algorithms introduced in 2.3. In addition, it implements three packages for the tool main GUI and the parsers associated with the tool. As shown in Figure 11, it consists of six subsystems:

1. The ClassDiagram subsystem.
2. The RegressionTestSuite subsystem.
3. The UseCaseDiagram subsystem.
4. The RTSToolGUI subsystem.
5. The Text Parser subsystem.
6. The XMI Parser subsystem.

Figure 11: RTSTool Subsystems and their Relationships

The RTSToolGUI subsystem includes a LoadGUI and a CompareGUI:

The LoadGui creates a ClassDiagram, UseCaseDiagram or RegressionTestSuite according to the user selection. The parser associated with the selected diagram is created, if needed, to parse and load the user specified file. If the diagram/test suite name and version specified by the user exists in the tool database, the user is given the choice either to use the existing

diagram/test suite or change the name or version of the diagram/test suite he wishes to load.

The CompareGui creates a ClassDiagramChanges or UseCaseDiagramChanges class according to the user selection to handle the comparison process. The results of comparison process are stored with the ClassDiagramChanges or UseCaseDiagramChanges objects.

The tool also uses two XMI parsers (ClassDiagramParser and UseCaseDiagramParser) to parse the UML artifacts files and load them into the tool. A RegressionTestSuiteParser is part of the TextParser subsystem that is used to parse the regression test suite. It takes as input, the different test cases in a text file that follows a specific format. An example that helps to illustrate this format is the CreateHelper test case:

Test Case:

Name: CreateHelper

Version: 1.0

Type: Functional

Sequence of Messages:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent ·  
CreateHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper ·  
HelpersExistsSet(helperPid) BootDiscoveryAgent · IamAck(objPid) LocalDiscoveryAgent
```

As shown above, this format contains several identifies that separate the different items of a test case. The ***Test Case*** identifier is used to separate between the different test cases and is used before every test case. The test case name follows the ***Name*** identifier and provides the test case name. Similarly, the test

case version follows the identifier *Version* and is used to distinguish between different versions of a test case. The test case type follows the identifier *Type* and can take one of several values according to the test case type (functional, non-functional, etc.). Finally, the test case sequence of messages follows the identifier *Sequence of Messages*. The notation used to represent a test case sequence of messages is the same notation used to represent the flow of messages in sequence diagrams to facilitate the comparison process between the test cases and sequence diagrams. Each operation in a test case sequence of messages is represented in the following notation: `OperationName (OperationParameters) class`, which specifies the operation name, parameters and class.

In the following sections, the tool package diagram, use case diagram, and activity diagram are presented and discussed. The class diagrams of the tool have already been described in section 2.2. The class diagrams for the RTSToolGui TextParser and XMIParser subsystems will not be provided in this section. The data dictionary of the RTSTool is also provided in Appendix B.

3.2.1 Use Case Diagram

As shown in Figure 12, the RTSTool use case diagram consists of nine use cases: “Load”, “Create use case diagram”, “Create class diagram”, “Load regression test suite”, “Compare class diagrams”, “Compare use case diagrams”, “Check diagrams consistency”, “Regression test selection” and “Generate report”.

Figure 12: RTSTool Use Case Diagram

Actor (User): The user loads the different UML artifacts (class diagrams, use case diagram) and the regression test suite. The user may compare two class diagrams, or two use case diagrams to find the differences between an original version and a modified version. Another possible scenario is to perform consistency check or regression test selection on the results obtained from analyzing the class and use case diagrams. The user may also generate a report that contains many useful statistics for the impact analysis process and/or the regression test selection.

The description of the use cases (Figure 12) is defined in terms of the required entry condition, flow of events and expected exit condition. The use cases are invoked based on the functionality selected by the user.

Use case name: Check diagrams consistency

Participating actor: User

Entry condition:

1. The user selects two class diagrams and two use case diagrams to perform the consistency check on.

Flow of events:

2. The sets of added, deleted and changed methods detected from the class diagram subsystem (say set A) are compared to the sets of added and deleted methods detected from the use case diagram subsystem (say set B). If some methods in set B are not in set A, these methods are added to the set of undefined invoked methods. On the other hand, if some methods in set A are not in set B, these methods are added to the set of unused defined methods.

Exit condition:

3. The sets of undefined invoked methods and unused defined methods are generated.

Use case name: Compare class diagrams

Participating actor: User

Entry condition:

1. The user selects two class diagrams to analyze from the list of loaded class diagrams.

Flow of events:

2. The two class diagrams (original and modified) are compared to detect design changes.

Exit condition:

3. The sets of added, changed and deleted attributes, classes and methods are generated.

Use case name: Compare use case diagrams

Participating actor: User

Entry condition:

1. The user selects two use case diagrams to analyze from the list of loaded use case diagrams.

Flow of events:

2. The two use case diagrams (original and modified) are compared to detect design changes.
3. The use cases corresponding sequence diagrams are compared to detect any call sequence changes.

Exit condition:

4. The sets of added, changed and deleted methods and use cases are generated.

Use case name: Create class diagram

Participating actor: User

Entry condition:

1. A new class diagram is already parsed and ready to be populated into the ClassDiagram subsystem ("Load" use case).

Flow of events:

2. The `ClassDiagram` subsystem is populated with the loaded class diagram.

Exit condition:

3. The class diagram is loaded and stored in the RTSTool database.

Use case name: Create use case diagram

Participating actor: User

Entry condition:

1. A new use case diagram is already parsed and ready to be populated into the `UseCaseDiagram` subsystem ("Load" use case).

Flow of events:

2. The information loaded from the use case diagram XMI file is processed with the information from the class diagram XMI file (if it exists) to detect the sequence diagrams methods return-types and parameter types (whenever possible) as described in Section 2.1.3.2.
3. The `UseCaseDiagram` subsystem is populated with the loaded use case diagram.

Exit condition:

4. The use case diagram is loaded and stored in the RTSTool database.

Use case name: Generate report

Participating actor: User

Entry condition:

1. The sets of added, changed and deleted attributes, classes and methods ("Compare class diagrams" use case) and the sets of added, changed and deleted

methods and use cases (“Compare use case diagrams” use case) have already been generated.

2. The sets of undefined invoked methods and unused defined methods (“Check diagrams consistency” use case) have already been generated.

3. The sets of obsolete, reusable and retestable test cases (“Regression test selection” use case) have already been generated.

Flow of events:

4. The following information is displayed in the report:

- i. The name and versions of the original and modified UML artifacts diagrams (class diagrams/use case diagrams) being analyzed.
- ii. The name and version of the regression test suite used.
- iii. The set of added, changed and deleted attributes, classes, methods and use cases.
- iv. The sets of undefined invoked methods and unused defined methods.
- v. The sets of obsolete, retestable and reusable test cases.

Exit condition:

5. The final report is generated.

Use case name: Load

Participating actor: User

Entry condition:

1. A new class diagram or use case diagram is available in the XMI format and it does not exist in the database (name and version).

Flow of events:

2. The different diagrams of the UML artifacts are parsed to identify the required information based on our models (Section 2.2).

Exit condition:

3. The class diagram or use case diagram information is ready to be passed to the “Create class diagram” or “Create use case diagram” use cases.

Use case name: Load regression test suite

Participating actor: User

Entry condition:

1. The regression test suite is available in a specific text format.

Flow of events:

2. The RegressionTestSuite subsystem is populated with the regression test suite test cases.

Exit condition:

3. All the regression test cases are loaded and stored in the RTSTool database.

Use case name: Regression test selection

Participating actor: User

Entry condition:

1. The user selects two class diagrams, two use case diagrams and one regression test suite from the lists of stored diagrams and test suites.

Flow of events:

2. The sets of obsolete, retestable and reusable test cases are generated.

Exit condition:

3. The different regression test cases are classified as obsolete test cases, reusable test cases and retestable test cases.

3.2.2 Activity Diagram

The activity diagram of the RTSTool (Figure 13) describes some of the different scenarios of the RTSTool usage. It describes the sequential dependencies between the different use cases. One possible scenario is the regression test selection scenario where the user creates or reuses two class diagrams (original and modified version), two use case diagrams (original and modified version) and one regression test suite of the original version. The different diagrams from the original and modified versions are compared “Compare class Diagrams” and “Compare use case diagrams” to detect the differences between them. Then the “Regression test selection” use case classifies the different regression test cases as obsolete, retestable or reusable. The “Generate report” use case displays many useful statistics such as: the sets of added, changed and deleted attributes, classes, methods and use cases. The sets of obsolete, retestable and reusable test cases are also generated. These statistics can be used in the impact analysis process.

Figure 13: RTSTool Activity Diagram

3.3 RTSTool Implementation

The RTSTool is implemented using the Java language [Java] and Poet Object Oriented Database Engine [Poet]. The Java language was chosen because it offers the following features [Brady+96]:

- **Simple and easy to learn:** It is easy to learn and write Java programs without extensive training. It omits many rarely used and confusing features from C++, but still allows programmers who are familiar with C++ to grasp the concepts of Java programming with ease.

- **Object Oriented language:** Object oriented programming allows software components to be reused. Programmers are allowed to focus on the data in the application and methods that manipulate that data, rather than thinking strictly in terms of procedures.
- **Robust:** In software development, the word "robust" means that the software does not break easily. In comparison to C and C++, Java places more restrictions on the programmer, such as lack of pointer support, which increases the chances of a program running smoothly.
- **Platform independent:** The Java compiler creates code that is intended for use with a Java-enabled browser installed on a network. This browser may be running on any platform such as Windows 95, Windows NT, and Unix workstations. To enable Java applications the compiler generates an architecture-neutral object file format that is executable on many processors.
- **Multithreaded:** Java allows applications where several things can be happening at once. This allows for support of real-time, interactive behavior, which is important for GUI based applications such as Web browsers.

Similarly, the Poet O-O database was chosen because it offers the following features [Poet]:

- **Complies with industry Standards:** Poet complies with industry-standard ODMG Java API that enhances developer productivity and accelerates the development process.

- **Modular Architecture:** Allows the developers to incorporate only the features they need for their application, such as concurrency controls, XML import/export, and logging and synchronization.
- **Portable:** It is portable to any implementation compliant with Java 2 Standard Edition (J2SE™). The runtime engine is available for any operating system featuring a Java 2-compliant Java Virtual Machine.

The database is used to lever the efficiency of the RTSTool. Employing the database allows the user to select any of the previously loaded diagrams/test suites to compare, consistency check or perform regression test selection. For example, if the user enters a name and a version of an existing class diagram, use case diagram, or regression test suite, the tool prompts the user whether he/she wishes to use the existing diagram/test suite or provide a new name (or version) for the new diagram and have it loaded and stored by the tool.

A total of six packages (Section 3.2), 28 classes and 5500 lines of code (comments not included) make up the core of the RTSTool.

Chapter 4 - Case Study

4.1 The Discovery System

The discovery system [Hassine 00] is an actual system that is used to test the implementation of the RTSTool. It is part of an IP router developed by Nortel Networks. The discovery system provides a generic mechanism for software agents to discover and subsequently communicate with one another in the network. The main function of discovery is to register new agents as they join the network and add them to the discovered set of agents. Similarly, remove the terminated agents from the discovered set of agents.

4.1.1 General Description

The discovery system is involved in constructing and distributing a global discovery table during the startup process of the router. The discovery table maps each process ID in the network to its socket address (IP address, UDP port). The discovery system maintains and distributes the discovery table to other agents in the network. This table is used to send messages from one agent to another.

The software design of the router is built on many subsystems: the messaging subsystem, the discovery subsystem, the surveillance subsystem, the routing subsystem and so on. Each agent in the network receives a copy of the global discovery table during the agent initialization process. The global table is updated when an agent starts or terminates.

4.1.2 Main Components of the System

As shown in Figure 14, a complete system consists of two control cards (also called global cards) and two line cards. Each card contains many agents, the boot agent is the agent responsible for starting all other agents and booting the card, it must exist in every card in the system. Each agent consists of many different objects. One of these objects is a discovery agent object depending on the card type. The objects are specified at compile time and cannot be changed during run-time.

GDA: Global Discovery Agent

EDA: Ephemeral Discovery Agent

BDA: Boot Discovery Agent

LDA: Local Discovery Agent

Figure 14: The Discovery System

The global (control) card is the only card containing the Global Discovery Agent (GDA). The GDA keeps a table of all the discovered agents in the network. Any other card contains the Ephemeral Discovery Agent (EDA). It is the card

equivalent of the GDA in the case of standalone cards. It is created only during the initialization process of the router and it keeps a table of the local agents on its card. If a new agent starts on any card it gets added to the EDA card table. When the global discovery agent starts, the EDA uploads his table to it and terminates.

The Boot Discovery Agent (BDA) exists in every boot agent in every card in the system. It is used to communicate with the GDA. This object handles all the local registration messages from the local agents on his card. It has an interface to the local discovery agent, the ephemeral discovery agent and the global discovery agent. There is only one instance of this class per card in the boot agent.

The Local Discovery Agent (LDA) exists in every agent in the system for the purposes of registering that agent. It keeps a table of all the agents in the system similar to the one the global discovery agent has. This table contains the addresses of all the agents in the network. This table is used to communicate with other agents in the system as needed.

4.1.3 Design of the Discovery Subsystem

The discovery system use case diagram and class diagram are provided in following sections. The use case description, data dictionary and sequence diagrams for the original and modified versions are provided in Appendix A for reference.

4.1.3.1 Use Case Diagram

The discovery system use case diagram (Figure 15) consists of eleven use cases to add, delete and register agents as they join or leave the network. A standalone card can also function locally in the absence of the control card and

agents on this card are called standalone agents. The NodalSurveillanceAgent of the surveillance subsystem interacts with the discovery subsystem to reports terminated agents. The original and modified versions of the discovery system share the same use case diagram.

The AddAgent use case deals with adding a newly starting agent to the global discovery table of the GDA. The agent information (process ID, IP address, UDP port, etc.) is added to the global discovery table. Similarly, the AddAgentOnstandaloneCard use case, deals with registering a newly starting agent on a standalone card. The agent information is added to card discovery table of the EDA.

The DeleteAgent use case deals with removing a terminated agent from the global discovery table of the GDA. Similarly, the DeleteAgentOnstandaloneCard use case, deals with removing a terminated agent from the card discovery table of the EDA.

The AddCard use case deals with adding a standalone card to the network. The card discovery table of the EDA is added to the global discovery table of the GDA and agents on this card can communicate with other agents on other cards in the network.

Figure 15: Discovery System Use Case Diagram

The CreateHelper use case, deals with creating a helper process to handle the download of the global discovery table from the GDA to the BDA. The global discovery table size is usually large and as the BDA cannot block to wait for the table download to finish, hence the need for a helper process.

The DownloadGlobalTable use case, handles the global discovery table download process as the table is downloaded to the BDA through several

messages depending on its size. Similarly, the `DownloadLocalTable` use case, handles the download process of the discovery table from the BDA to the LDA.

The `RegisterAgent` use case, handles a new agent registration with the different agents in the global discovery table.

The `RegisterEphemeral` use case, handles the registration of the EDA of a standalone card with the GDA of the global card.

Finally, the `UpdateAgentTable` use case, handles the addition of new agents information to the global discovery table.

4.1.3.2 Class Diagrams

The discovery system class diagram consists of nine classes (Figure 16). Two of these classes, the `ObjPid` and `Process` are base classes that every process in the system must inherit from in order to be able to communicate with other processes. The `Timer` class is used to provide timer services to other classes. The `BaseDiscovery` class is an abstract base class that contains many functions used by the different discovery classes. The `BootDiscoveryAgent`, `Helper`, `EphemeralDiscoveryAgent`, `GlobalDiscoveryAgent` and `LocalDiscoveryAgent` are discovery classes used to add and delete agents as they join or leave the network. All the discovery objects are active objects.

Figure 16: Original Discovery Class Diagram

Figure 17: Modified Discovery Class Diagram

The class diagram of the second version of the discovery system (Figure 17) contains many changes with respect to version 1. Most of these changes are in terms of moved methods from one class to another. Some method parameters were also merged to reduce the size of the discovery tables and reduce the messaging overhead associated with downloading these tables.

For example, the `agentMap` attribute and the `nodeDown` method were moved from the BDA class to the LDA class.

4.2 Test Cases for the Discovery System

The discovery regression test cases are divided into two categories: functional and non-functional test cases.

The functional test cases are generated based on the strategy defined in [Briand+01]. Based on this strategy, regression test cases can be generated using UML artifacts such as use cases and their corresponding sequence diagrams. The sequence diagrams are expressed as regular expressions to identify possible use case scenarios. Then, the path realization conditions associated with each scenario are identified. Having identified the path realization conditions under which each scenario is going to be executed and tested, the precise operation sequences are identified. Finally, the test oracles are derived for each operation sequence based on the post-conditions of operations.

The non-functional test cases are provided by Nortel Networks to test different aspects of the discovery system (i.e., scalability, reliability, speed, etc.).

The discovery regression test suite contained 596 test cases. 546 functional test cases, generated based on the strategy described in [Briand+01] and 50 non-functional test cases. These test cases will be represented using regular language expressions in the format described before in Section 3.2.

4.2.1 Generated Test Cases

In this section we use UML artifacts to derive regression test cases in particular, we use the use cases and their corresponding sequence as our source of information for testing purposes as described in [Briand+01].

4.2.1.1 Generating Use Case Sequences

We need to identify *possible* execution sequences of use cases. Refer to the discovery system use case diagram provided before in Figure 15. A first step is the representation of use case sequences.

4.2.1.1.1 Representation of Use Case Sequences

We represent sequential dependencies between use cases by the means of an activity diagram (Figure 18) for each actor in the system. In such a diagram, the vertices are use cases and the edges are sequential dependencies between the use cases. The use cases are grouped into *swimlanes* with respect to their responsibilities in terms of manipulated objects.

The following is the use case sequential constraint diagram or activity diagram of the discovery system. It captures the sequential dependencies between use cases to help visualize such dependencies.

Figure 18: The Discovery System Activity Diagram

Then this activity diagram is represented as a weighted graph as shown in Figure 19, where vertices are sequential dependencies between use cases and edges represent use cases. In the discovery system activity diagram there is one special case: α . It is an edge that is systematically added for every final dependency in the swimlanes, to a final graph vertex (vertex 1 in Figure 19). The reason is to have every path in the graph end in a common, final vertex, and

facilitate the generation of regular expressions using matrices, as shown in the next section.

Our next objective is to generate regular expressions from the weighted graphs modeling sequential constraints between use cases (Figure 19). Those regular expressions will constitute one component of the system test requirements.

Figure 19: Weighted Graph Representation of the Activity Diagram

4.2.1.1.2 Analysis of Use Case Weighted Graphs

Once we have a weighted graph representation of the use case dependencies, this graph is analyzed to produce regular expressions that model possible paths in the graph. Then, we derive paths to be covered by analyzing the graph of sequential dependencies between use cases. This is done by the matrix representation of the weighted graph as provided in the Table 1. The edges' weights are placed into the matrix cells. A column entry specifies the links entering the node characterized by the column in the weighted graphs. A row entry specifies the links leaving the nodes.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0			A							H		J	
1													
2				B									
3					C								
4						D							
5							E						
6								F					
7									G				
8		α											
9											I		
10					C								
11													K
12		α											

Table 1: Matrix Representation of the Weighted Graph

The regular expressions that correspond to paths in the activity diagram, from vertex 0 to vertex 1 are: J.K, A.B.C.D.E.F.G, H.I.C.D.E.F.G.

The last two regular expressions are merged into one as they only differ in their start: (A.B|H.I).C.D.E.F.G. Then the regular expressions are formalized as the following regular expression: (J.K| $\emptyset$) | (((A.B|H.I).C.D.E.F.G)| $\emptyset$). Then from such an expression we can drive actual path to be covered by picking a use case from each interleaved term.

4.2.1.2 Identifying Use Case Scenarios

Each discovery use case corresponds to a sequence diagram. The sequence diagram shows how the use case is realized by the interactions of objects, which are instances of classes in the system class diagrams.

4.2.1.2.1 Expressing Sequence Diagrams as Regular Expressions

In order to represent sequence diagrams in an analyzable and compact form, the sequence diagrams may be re-expressed as a regular expression whose alphabet are the public methods of the objects playing a role in sequence diagrams. The notation `OperationName (OperationParameters) class` is used to denote which operation is executed, using which parameters and to which class it belongs. This is the same notation used before in Section 3.2 to represent test cases sequence of messages. For example, for the `CreateHelper` sequence diagram shown in Figure 20, this would yield the following regular expression:

CreateHelper ->

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent .  
(localTmerId.timeOut() LocalDiscoveryAgent . localTmerId.startTimer(timerPeriod) LocalDiscoveryAgent  
. Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent) * .  
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper .  
helpersExistsSet(helperPid) BootDiscoveryAgent . IamAck(objPid) LocalDiscoveryAgent
```


Any card

Figure 20: CreateHelper Sequence Diagram

4.2.1.2.2 Identifying Path Realization Conditions

From the procedure above, we obtained a regular expression for each of the sequence diagrams, each having a number of conditions associated with their executions. Associated with each path within a sequence diagram, one can derive the guard conditions associated with these paths, the conjunction of conditions that must be fulfilled for that path to be enabled.

In the case of the CreateHelper sequence diagram (Figure 20), no helper process should be running in order to create a new process (discovery

system precondition), therefore the path realization condition for this sequence diagram is:

```
BootDiscoveryAgent::helperExistsGet() = false
```

4.2.1.2.3 Specifying Operation Sequences

Having identified the test conditions under which each term is going to be executed and therefore tested, we need to identify the precise operation sequences to be executed for each term.

Using the `CreateHelper` example used in 4.2.1.2.1, the sequence diagram contains the iteration symbol (*). In order to generate the use case variant sequences for this sequence diagram, the iteration is bypassed, performed once, eight times and sixteen times (maximum number of agents on one card). The sequences of operations obtained using this strategy for the `CreateHelper` sequence diagram are:

CreateHelper, sequence 1:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper -
helpersExistsSet(helperPid) BootDiscoveryAgent - IamAck(objPid) LocalDiscoveryAgent
```

CreateHelper, sequence 2:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
localTmerId.timeOut() LocalDiscoveryAgent - localTmerId.startTimer(timerPeriod) LocalDiscoveryAgent -
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper -
helpersExistsSet(helperPid) BootDiscoveryAgent - IamAck(objPid) LocalDiscoveryAgent
```

CreateHelper, sequence 3:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
(localTmerId.timeOut() LocalDiscoveryAgent - localTmerId.startTimer(timerPeriod) LocalDiscoveryAgent -
.Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent)8 -
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper -
helpersExistsSet(helperPid) BootDiscoveryAgent - IamAck(objPid) LocalDiscoveryAgent
```

CreateHelper, sequence 4:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -  
(localTmerId.timeOut() LocalDiscoveryAgent - localTmerId.startTimer(timerPeriod) LocalDiscoveryAgent  
.Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent)16.  
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper -  
helpersExistsSet(helperPid) BootDiscoveryAgent - IamAck(objPid) LocalDiscoveryAgent
```

4.2.1.2.4 Identifying Test Oracles

Now we have defined the operation sequences to be executed and tested, we need to derive test oracles for each tested sequence. The main source for deriving a test oracle is the post-condition of operations in a sequence.

The test oracle for the CreateHelper sequence diagram is obtained from the post-conditions of the CreateHelper operation:

```
BootDiscoveryAgent::helperExistsGet() = true
```

4.2.1.2.5 Constructing Decision Tables

Once we have, for a given use case, identified the operation sequences to be tested, their initial conditions and oracles, we can formalize all this in a decision table that will be used as a formal set of test requirements, which will be part of the test plan.

The decision table for the CreateHelper sequence diagram (Figure 20) is:

Variants (Use case B)	Condition Section	Action Section	
		Messages to Actor	State Change
	A	I	
B1	Yes	Yes	No
B2	No	No	No

Table 2: Decision Table for CreateHelper Use Case

Initial Conditions:

A: BootDiscoveryAgent::helperExistsGet() = false

Messages to Agent LocalDiscoveryAgent:

I: IamAck(objPid)

4.2.1.3 Generating Variant Sequences

If we assume that for each use case, we have a decision table, we need to go further and devise a sequence of operations to be tested over an entire *use case sequence* (as defined in Section 4.2.1.1.2). In other words, we need to go from use case sequences to use case *variant* sequences, using use case *decision tables*.

The use case sequences for the discovery system are:

J.K, A.B.C.D.E.F.G, H.I.C.D.E.F.G

As described in Section 4.2.1.2.5 above, one variant corresponds to a possible path realization condition for one of the product terms in the sequence diagram regular expression. A variant may require several use cases, as iterations may be present in the corresponding product term. Each use case has many variants therefore the following is a set of the possible use case variant sequences for the discovery system:

J1.K1,
A1.B1.C1.D1.E1.F1.G1, A1.B1.C2.D1.E1.F1.G1, A1.B1.C3.D1.E1.F1.G1,
A1.B1.C4.D1.E1.F1.G1, A2.B1.C1.D1.E1.F1.G1, A2.B1.C2.D1.E1.F1.G1,
A2.B1.C3.D1.E1.F1.G1, A2.B1.C4.D1.E1.F1.G1, A3.B1.C1.D1.E1.F1.G1,
A3.B1.C2.D1.E1.F1.G1, A3.B1.C3.D1.E1.F1.G1, A3.B1.C4.D1.E1.F1.G1,
A4.B1.C1.D1.E1.F1.G1, A4.B1.C2.D1.E1.F1.G1, A4.B1.C3.D1.E1.F1.G1,
A4.B1.C4.D1.E1.F1.G1, H1.I1.C1.D1.E1.F1.G1, H1.I1.C2.D1.E1.F1.G1,
H1.I1.C3.D1.E1.F1.G1, H1.I1.C4.D1.E1.F1.G1, H2.I1.C1.D1.E1.F1.G1,
H2.I1.C2.D1.E1.F1.G1, H2.I1.C3.D1.E1.F1.G1, H2.I1.C4.D1.E1.F1.G1,
H3.I1.C1.D1.E1.F1.G1, H3.I1.C2.D1.E1.F1.G1, H3.I1.C3.D1.E1.F1.G1,

H3.I1.C4.D1.E1.F1.G1, H4.I1.C1.D1.E1.F1.G1, H4.I1.C2.D1.E1.F1.G1,
H4.I1.C3.D1.E1.F1.G1, H4.I1.C4.D1.E1.F1.G1

The first use case variant sequence J1.K1 represents a use case scenario in which an agent on a standalone card is started and registered with the discovery system then this agent is terminated and therefore deleted from the discovery subsystem tables. However, it is not possible to construct the test case by only joining the AddAgentOnStandaloneCard and the DeleteAgentOnStandaloneCard use cases. We must terminate the agent before the DeleteAgentOnStandaloneCard can be used. This is done by invoking the terminate() procedure on the LocalDiscoveryAgent of the newly started agent. The complete sequence is therefore:

```
registerAgent(objPid) LocalDiscoveryAgent2.localTimerId.startTimer(timerPeriod) LocalDiscoveryAgent
2.Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
IamAck(objPid) LocalDiscoveryAgent2.localTimerId.stopTimer() LocalDiscoveryAgent2 -
updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel) EphemeralDiscoveryAgent -
agentTableDownload(globalTable, seqNumber) BootDiscoveryAgent -
agentTableDownloadCompleteAck(seqNumber) EphemeralDiscoveryAgent -
agentTableDownload(globalTable, seqNumber) LocalDiscoveryAgent2 -
agentTableDownloadCompleteAck(seqNumber) BootDiscoveryAgent -
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) LocalDiscoveryAgent1 -
updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel) LocalDiscoveryAgent1 -
IamAck(objPid) BootDiscoveryAgent - updateAgentMap(objPid) BootDiscoveryAgent -
IamAck(objPid) LocalDiscoveryAgent2 - agentDiscoveryCompleted(objPid) BootDiscoveryAgent -
terminate() LocalDiscoveryAgent2 - nodeDown(objPid) BootDiscoveryAgent -
agentDown(objPid) EphemeralDiscoveryAgent - deleteAgent(objPid) EphemeralDiscoveryAgent -
ephemeralTimerId.startTimer(timerPeriod) EphemeralDiscoveryAgent -
deleteAgentReq(objPid) BootDiscoveryAgent - ((deleteAgentReq(objPid) LocalDiscoveryAgent1 -
deleteAgent(objPid) LocalDiscoveryAgent1 - deleteAgentReqAck(objPid) BootDiscoveryAgent -
```

```

updateAgentMap(objPid) BootDiscoveryAgent ) | (deleteAgentReq(objPid) LocalDiscoveryAgent2 -
deleteAgent(objPid) LocalDiscoveryAgent2 - deleteAgentReqAck(objPid) BootDiscoveryAgent -
updateAgentMap(objPid) BootDiscoveryAgent ) ) . ( (deleteAgentReq(objPid) LocalDiscoveryAgent2 -
deleteAgent(objPid) LocalDiscoveryAgent2 - deleteAgentReqAck(objPid) BootDiscoveryAgent -
updateAgentMap(objPid) BootDiscoveryAgent ) | (deleteAgentReq(objPid) LocalDiscoveryAgent1 -
deleteAgent(objPid) LocalDiscoveryAgent1 - deleteAgentReqAck(objPid) BootDiscoveryAgent -
updateAgentMap(objPid) BootDiscoveryAgent ) ) . deleteAgentReqAck(objPid) EphemeralDiscoveryAgent -
ephemeralTimerId.stopTimer() EphemeralDiscoveryAgent

```

As the test cases cover all the possible scenarios for the sequence A.B.C.D.E.F, then to construct the variant use case sequences for the scenario A.B.C.D.E.F.G, we need to append the use case “G” DeleteAgent to each test case. Similarly for the use case variant H.I.C.D.E.F.G, the generated test cases cover all the possible scenarios for the sequence H.I.C.D.E.F, then to construct the variant use case sequences for complete scenario H.I.C.D.E.F.G, we need to append use case “G” DeleteAgent to each test case.

4.2.2 Existing Test Cases

The discovery system non-functional test cases mainly tests the system ability to handle large networks, measure the speed and reliability of the system, and stress the system to its limits in terms of the maximum number of cards and maximum number of agents the system can handle. The consistency of the discovery tables maintained at the different discovery agents is checked as well.

An example of these test cases is the MaxNumberOfCardAgents test case that tests the ability of the discovery system to register the maximum number of agents per card (16 agents). The agents are created then the global discovery table is retrieved to make sure it contains the 16 agents.

Test Case:

Name: MaxNumberOfCardAgents

Version: 1.0

Type: NonFunctional

Sequence of Messages:

```
(Process::create(OUT: objPid).LocalDiscoveryAgent::registerAgent(IN: objPid))16
```

Test Oracle:

```
GlobalDiscoveryAgent::getAgentTable(OUT: globalInformationTable) .  
globalInformationTable->exists(agent : ObjPid | agent.objPid)
```

4.3 Regression Testing Results for the Discovery System

The design changes for the discovery system are divided into two main categories: The impact analysis results and the regression test selection results. The impact analysis results are provided in terms of the design changes in the discovery system modified UML models. The regression test selection results are provided in terms of the obsolete, retestable and reusable test cases.

4.3.1 Impact Analysis Results

The design changes for the discovery system detected by the RTSTool are summarized in Table 3. The table describes the total number of attributes, classes, DefinedMethods, InvokedMethods, and Sequence diagrams/Use Case diagrams in the original version of discovery and the number of added, changed and deleted elements in the modified version of discovery. For a complete list of the design changes please refer to Appendix A.

Impact Analysis Results	Total	Added	Changed	Deleted
Attributes	14	1	0	3
Classes	9	0	6	0
Class Diagram Methods	70	21	0	16

Use Case Diagram Methods	63	16	0	14
Sequence Diagrams/ Use Cases	11	11	0	0

Table 3: Impact Analysis Results

The RTSTool detected the exact number of added and deleted attributes. However, it could not detect the changed attributes. There were three changed attributes: `agentInformationTable`, `cardInformationTable` and `globalInformationTable`. These attributes are of type “Table”, the attributes changed because the number of entries in each table was reduced from five entries to four entries. As the RTSTool did not represent complex data types, these changes were not detected. The complex data types can be represented by another object in the RTSTool to model their internal structures.

Similarly, the RTSTool detected the exact number of added, changed and deleted classes. The class changes were mainly caused by attribute changes and/or method changes. There were no class relationships changes.

In the case of defined methods (class diagram methods), the RTSTool detected the exact number of added, changed and deleted methods. Ten methods that were classified as added/deleted methods were actually changed by removing one of the parameters of the original/modified methods, but were classified as added as described in Section 2.1.3.2.

Similarly, in the case of invoked methods (use case diagram methods), the RTSTool detected the exact number of added, changed and deleted methods. Eight methods that were classified as added/deleted methods were actually

changed by removing one of the parameters of the original/modified methods, but were classified as added as described in Section 2.1.3.2.

The consistency-check of the class diagrams and use case diagrams comparison results, resulted in zero methods invoked but not defined and seven methods classified as defined but not used. Five methods from the added methods set and two methods from the deleted methods set, these methods were used only for testing purposes.

4.3.2 Regression Test Selection Results

Based on the discovery system design changes described in the previous section, the discovery system regression test cases were classified as obsolete, retestable and reusable test cases. Table 4 displays the results of regression test selection:

Regression Test Selection Results	Total Number of Test Cases	Obsolete	Reusable	Retestable
Test Cases	596	564	32	0

Table 4: Regression Test Selection Results

The regression test suite contained 596 test cases (546 functional and 50 non-functional test cases), 564 test cases were classified as obsolete test cases. 546 functional test cases were classified as obsolete due to a changed sequence of messages in the modified version of discovery (due to an added, changed or deleted method) and 18 non-functional test cases were classified as obsolete because they contained one or more deleted method in their sequence of messages. Thirty-two non-functional test cases were classified as reusable because they did not contain any changed or deleted methods. No test cases were

classified as retestable test cases. However, most of the test cases that were classified as obsolete can be easily changed (by substituting the changed methods) and therefore reused for testing.

For example, the `CreateHelper` test case, described before in Section 4.2.1.2.1, was classified as an obsolete test case because it contained the `Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList)` method that has changed in the modified discovery version to:

`Iam(objPid, IPAddress, UDPPort, parentLevel, agentList)`. Therefore, the sequence of messages it contains does not match the new sequence of messages. However, this test case can be easily modified (by replacing the `Iam` method) and reused.

Test Case:

Name: `CreateHelper`

Version: 1.0

Type: Functional

Sequence of Messages:

```
Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent -  
(localTmerId.timeOut() LocalDiscoveryAgent - localTmerId.startTimer(timerPeriod) LocalDiscoveryAgent  
. Iam(objPid, IPAddress, UDPPort, agentParent, agentLevel, agentList) BootDiscoveryAgent) * .  
createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) Helper .  
helpersExistsSet(helperPid) BootDiscoveryAgent - IamAck(objPid) LocalDiscoveryAgent
```

On the other hand, the `MaxNumberOfCardAgents` test case, described before in 4.2.2, which tests the ability of the discovery system to register the maximum number of agents per card (16 agents), was classified as reusable test

case because it does not contain any deleted or changed methods. (Nortel Networks provided this test case)

Test Case:

Name: MaxNumberOfCardAgents

Version: 1.0

Type: NonFunctional

Sequence of Messages:

`(Process::create(OUT: objPid).LocalDiscoveryAgent::registerAgent(IN: objPid))16`

Chapter 5 – Conclusions

In this thesis we suggested the use of UML design diagrams to perform impact analysis and regression test selection. We have developed precise algorithms to manipulate UML models and detect the design changes among them. The design changes along with the traceability information between the design models and test cases are used to perform impact analysis and regression test selection. The regression test cases were classified in one of three categories: obsolete, retestable and reusable. The complete methodology and algorithms were implemented in the RTSTool to prove its validity and practicality. The RTSTool was then used to perform a thorough study of a real industrial system. The results of the case study have shown that limited design changes can cause major and complex changes to regression test suites and thus provide motivations for effective automation.

The regression test selection and impact analysis based on UML design models is intended to augment the regression test selection and impact analysis based on source code analysis. Obviously, the level of details that can be obtained from analyzing the source code cannot be obtained from analyzing the UML models that provides a high level view of the software design. Therefore, our purpose is to provide additional support for the regression test selection and impact analysis processes at an early stage before the code changes are implemented. Having the regression test selection early in the design phase allows for better test planning and allocation of testing resources.

5.1 Limitations

The RTSTool limitations are divided into three categories:

- **Practical Limitations:**
 - **UML models availability:** Some legacy or industrial systems may not have a complete set of UML design diagrams (class diagram, use case diagram and sequence diagram) or may have statechart diagrams instead of sequence diagrams. In the first case, the UML diagrams have to be generated before using the tool (using any reverse engineering techniques for example). In the second case, the statechart diagrams can be used to generate allowed sequences of method calls.
 - **The accuracy of the regression test selection results depends on the completeness of the UML models.** For example, if the user provides an incomplete sequence diagrams, then some test cases may be classified as reusable test cases when they should be classified as retestable.
 - **The results generated based on source code analysis will give more details than the results generated based on UML design models analysis especially in regards to changes made at the algorithmic level.**
- **Methodology Limitations:**
 - **Handling of overloaded methods (Section 2.1.3.2) where the method return type/parameter type cannot be identified.**

- **Implementation limitations:**
 - **Handling of complex data types (table, vector, etc.):** The current version of the RTSTool represents the different data types as strings. Complex data types need to be represented by separate objects to represent their internal structures. For example, if an attribute is of type `table`, and the table stores the values of two parameters: an `integer` and a `float`. If the type of the second parameter changes, i.e., from `float` to `integer`, the attribute should be marked as changed. The current version of the RTSTool does not recognize those changes.
 - **Guard condition handling:** As described in Section 1.5, this issue was not addressed in this research and therefore needs further study.
 - **Determining attributes' access methods:** The current version of the RTSTool does not implement the attribute access methods approach based on OCL contracts as described in Section 2.1.3.1.
 - **Handling of composition and aggregation class relationships:** The current version of the RTSTool represents both relationships types as an association relationship.
 - **In some cases more than one sequence diagram is associated with one use case.** The current version of the RTSTool assumes that only one sequence diagram can be associated with one use case. Further research is needed to address this issue in terms of

imposing certain rules on the sequence diagrams naming convention to be able to identify the sequence diagrams associated with each use case.

5.2 Future Work

Future work should be done to address the RTSTool limitations listed in Section 5.1. The RTSTool can also be integrated with other commercial tools such as:

- Extend the tool to handle other types of UML models (e.g., statechart diagrams).
- Integrate the tool with Case tools used to generate UML diagrams (i.e., Rose, Rhapsody, etc.) and test case generation and execution tools.
- Test the tool with other industrial systems as the tool was only tested using a telecommunication system.

References

[Abdullah+97]

K. Abdullah and L. White, "A Firewall Approach for the Regression Testing of O-O Software," Technical Report, Case Western Reserve university, Cleveland, Ohio, 1997.

[Binder 99]

R. Binder, "Testing Object Oriented Systems," Addison-Wesley, 1999.

[Bohner+96]

S. Bohner, R. Arnold, "Software Change and Impact Analysis," IEEE Computer Society Press, 1996.

[Booch+99]

G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide," Addison-Wesley, 1999.

[Brady+96]

K. Brady, <http://disc.cba.uh.edu/~rhirsch/fall96/kevin.htm>, 1996

[Briand+01]

L. Briand, Y. Labiche, "A UML-based Approach to System Testing," Proceedings of UML 2001.

[Bruegge+99]

B. Bruegge, A. Dutoit, "Object-Oriented Software Engineering," Prentice Hall, 1999.

[Cormen+90]

T. Cormen, C. Leiserson and R. Rivest, "Introduction to algorithms," Cambridge, Mass., M.I.T. Press. 1990.

[Croker+90]

R. T. Croker and A. Mayrhauser, "Maintenance Support Needs for O-O Software," In Proc. Conference on Software Maintenance, pp. 266-271, 1990.

[Deitel+98]

Deitel, Deitel, "How to Program C++," 2nd edition, Prentice Hall, 1998.

[Deitel+99]

Deitel, Deitel, "Java How to Program," 3rd edition, Prentice Hall, 1999.

[Fowler+99]

M. Fowler and K. Scott, "UML Distilled," Addison-Wesley, 1999.

[Gomaa 00]

H.Gomaa, "Designing Concurrent, Distributed, and Real-time Applications with UML," Addison-Wesley, 2000.

[Hajla 96]

H. Ben Hajla, "Traceability in Object-Oriented Quality Engineering," Masters in computer science, Ottawa-Carleton Institute for Computer Science, Ottawa, Canada, 1996.

[Hassine 00]

J. Hassine, "The Discovery System Detailed Design," Nortel Networks Corp. 2001.

[Hartmann+89]

J. Hartmann and D.J. Robson, "Revalidation during software maintenance phase," In *IEEE Conference on Software Maintenance*, pp. 70-80, 1989.

[Jalote 96]

P. Jalote, "An Integrated Approach to Software Engineering," 2nd edition, Springer-Verlag Inc., 1996.

[Java]

Sun Microsystems, www.sun.com, Java 2, Standard Edition, v 1.4.

[Kung+94]

D. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyosima and C. Chen, "Change Impact Identification in O-O Software Maintenance," In *Proceeding of the International Software Maintenance*, pp. 202-211, 1994.

[Kung+95]

D. Kung, J. Gao, P. Hsia, F. Wen Y. Toyosima and C. Chen, "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs," *Journal of Object-Oriented Programming*, 8(2), pp. 51-56, 1995.

[Kung+96]

D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyosima and C. Chen, "On Regression Testing of Object-Oriented Programs," *Journal of Systems and Software*, Vol. 32, No. 1, pp. 11-16, Jan. 1996.

[Larman 02]

C. Larman, "Applying UML and Patterns," Prentice Hall, 2002.

[Laski+92]

J. Laski and W.Szerner, "Identification of Program Modification and its Application in Software Maintenance," *In Proc. IEEE Transactions on Software Engineering*, pp. 18:1045- 52, 1992.

[Leung+91]

N.K.N Leung and L. White, "A cost model to compare regression test strategies," *In Proc. Conference on Software Maintenance*, pp. 201- 208, 1991.

[Linz 97]

P. Linz, "An Introduction to Formal Language and Automata," 2nd edition, Jones and Bartlett Publishers, 1997.

[Luiu+93]

L. Luiu, L.Robson and R. Ellis. "A Data Management System for Regression Testing," *International Conference on software Quality management* page 527-539. British Computer Society, Wessex Institute of Technology, Soothsampton, Marsh 1993.

[Pfleeger 98]

S. Pfleeger, "Software Engineering Theory and Practice," Prentice Hall, 1998.

[Poet]

FastObjects by Poet, www.poet.com, Poet 6.1, Poet Java Programmers Guide.

[Rothermel+94]

G. Rothermel and M.J. Harrold, "Selecting Regression Tests for Object-Oriented Software," *In Proc. Of the International Conf. on Software Maintenance*, " Victoria, CA, pp. 14-25, Sept.1994.

[Rothermel+99]

G. Rothermel, M. J. Harrold and J. Dedhia, "Regression Test Selection for C++ Software," Technical Report, Oregon State University, Jan. 1999.

[Sedgewick 88]

R. Sedgewick, "Algorithms," 2nd edition, Addison-Wesley, 1988.

[Stevens+00]

P. Stevens, R. Pooley, "Using UML Software Engineering with Objects and Components," 2nd edition, Pearson Education Limited, 2000.

[Wilde+92]

N. Wilde and R. Huitt, "Maintenance Support for Object Oriented Programs," *IEEE Trans. on Software Eng.*, Vol. 18, No. 12, pp. 1038 - 1044, Dec. 1992.

[Warmer+99]

J. Warmer and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML," Addison-Wesley, 1999.

Appendix A: The Discovery System

This appendix is organized as follows: first we give the discovery use case description, then the sequence diagrams for the original and modified versions and finally the complete data dictionary for the discovery system.

Use Case Description

Use case name: AddAgent

Description: This use case deals with adding a new agent to the network. The local discovery agent of the new agent handles the discovery process.

Participating Agent: Agent.

Entry Condition: An agent is starting on a card that is already discovered (part of the network).

Flow of Events:

1. The starting agent triggers his LocalDiscoveryAgent to register with the discovery system. The local discovery agent registers locally with his boot discovery agent then the boot discovery agent registers the new agent with the global discovery agent on the control card.
2. The global discovery agent updates his discovery table and sends it to the local discovery agent to register with the rest of the agents in the table.

Exit Condition: The new agent has registered with all the agents in the global discovery table.

Use case name: AddAgentOnStandaloneCard

Description: This use case deals with the registration of a new agent on a standalone card. The new agent registers with his boot discovery agent on his card then the boot discovery agent registers the new agent with the ephemeral discovery agent on the same standalone card.

Participating Agent: StandaloneAgent.

Entry Condition: A new agent is starting on a standalone card.

Flow of Events:

1. The starting agent on the standalone card triggers his LocalDiscoveryAgent to register with the discovery system. The local discovery agent registers locally with his boot discovery agent then the boot discovery agent registers locally also with the ephemeral discovery agent on the standalone card.
2. The ephemeral discovery agent updates his card discovery table and sends it to the local discovery agent to register with the rest of the local agents in the table.

Exit Condition: The newly starting standalone agent is ready to send and receive messages to other local agents on his card.

Use case name: AddCard

Description: This use case deals with adding a new card with several agents registered on it to the network.

Participating Agent: Card.

Entry Condition: A new standalone card is joining the network.

Flow of Events:

1. The ephemeral discovery agent on this card registers with the global discovery agent residing on the control card and sends his card discovery table to the global discovery agent.
2. The ephemeral discovery agent updates the boot discovery agent with the global discovery agent address and terminates.
3. The boot discovery agent requests the global discovery table from the global discovery agent and updates his local discovery agents with the global discovery table. The local discovery agents register with the rest of the agents in the global discovery table.

Exit Condition: The card has joined the network and agents on this card can communicate with any agents in the network. Also the ephemeral discovery agent has terminated.

Use case name: CreateHelper

Description: This use case deals with the creation of a helper process to handle the global discovery table download. The helper is created from the boot discovery agent and it terminates as soon as the table download process is complete. Only one helper can be running at one time.

Participating Agent: Agent.

Entry Condition: The boot discovery agent receives a registration message from a newly starting agent on his card.

Flow of Events:

1. The boot discovery agent checks if a helper exists or not, if a helper exists, no other helper is created and the registration message is ignored otherwise a helper process is created.

Exit Condition: The boot discovery agent has created a helper process.

Use case name: DeleteAgent

Description: This use case deals with deleting an agent from the network. Two subsystems are involved in this use case, the discovery subsystem and the surveillance subsystem. Whenever an agent terminates, the surveillance subsystem detects it and notifies the discovery subsystem to delete it from the discovery table.

Participating Agent: NodalSurveillanceAgent.

Entry Condition: One agent has terminated due to a dead lock, infinite loop or segmentation fault.

Flow of Events:

1. The surveillance subsystem detects the terminated agent and reports his information to the discovery subsystem.
2. The global discovery agent and local discovery agents delete the agent from their tables.

Exit Condition: The terminated agent is removed from all the local discovery agent tables and the global discovery table.

Use case name: DeleteAgentOnStandaloneCard

Description: This use case deal with deleting an agent on a standalone card from the card discovered agents. Two subsystems are involved in this use case, the

discovery subsystem and the surveillance subsystem. Whenever an agent terminates, the surveillance subsystem detects it and notifies the discovery subsystem to delete it from the discovery table.

Participating Agent: NodalSurveillanceAgent

Entry Condition: One standalone agent has terminated due to a dead lock, infinite loop or segmentation fault.

Flow of Events:

1. The surveillance subsystem detects the terminated agent and reports his information to the discovery subsystem.
2. The ephemeral discovery agent and the standalone local discovery agents delete the agent from their tables.

Exit Condition: The terminated agent is removed from the local discovery agent tables on the standalone card and the card discovery table.

Use case name: DownloadGlobalTable

Description: This use case deals with the downloading of the global discovery table to the card on which a new agent has started. The global discovery table is usually a big table that cannot be downloaded in one message.

Participating Agent: Agent or Card.

Entry Condition: One agent has started or terminated and the global discovery agent needs to download the new discovery table to it.

Flow of Events:

1. The global discovery agent receives a request to update his global discovery table from a certain boot discovery agent. The global discovery table is updated and then downloaded to the updating boot discovery agent.

Exit Condition: The global discovery table is received at the boot discovery agent.

Use case name: DownloadLocalTable

Description: This use case deals with the download of the global discovery table from the boot discovery agent to the local discovery agent of the newly starting agent.

Participating Agent: Agent or Card.

Entry Condition: One agent has started and the global discovery table has been received at the boot discovery agent.

Flow of Events:

1. The boot discovery agent downloads the global discovery table to the newly starting local discovery agent.

Exit condition: The discovery table has been received at the local discovery agent.

Use case name: RegisterAgent

Description: This use case deals with the registration of a new agent. Once the local discovery agent of the new agent has received the global discovery table, the local agent registers with all the agents in the discovery table.

Participating Agent: Agent.

Entry Condition: The global discovery table has been received at the local discovery agent of the starting agent.

Flow of Events:

1. The local discovery agent of the newly starting agent registers with all the agents in the global discovery table.
2. The local discovery agents update their tables with the new agent information and acknowledge the sending agent.

Exit Condition: The newly started agent is ready to communicate with other agents in the system.

Use case name: RegisterEphemeral

Description: This use case deals with the registration of the ephemeral discovery agent with the global discovery agent.

Participating Agent: Card.

Entry Condition: One standalone card is joining the network.

Flow of Events:

1. The ephemeral discovery agent registers with the global discovery agent on the control card.

Exit Condition: The ephemeral discovery agent of the standalone card has registered with global discovery agent.

Use case name: UpdateGlobalTable

Description: This use case is used to update the global discovery agent table. The global discovery agent keeps one global table for all the agents in the network. The global discovery agent updates his table whenever a new agent or a new card

joins or leaves the network. It can only be updated if there is no pending table downloads.

Participating Agent: Agent or Card.

Entry Condition: The global discovery agent receives table update message from any boot discovery agent.

Flow of Events:

1. The global discovery agent receives a table update message as a result of a new agent or a new card joining the network.
2. If there is no pending table downloads, the discovery table is updated otherwise the update message is ignored.

Exit Condition: The global discovery table is updated with the new agent information.

Discovery Original Sequence Diagrams

Figure 21: Original AddAgent Sequence Diagram

Figure 22: Original AddAgentOnStandaloneCard Sequence Diagram

Figure 23: Original AddCard Sequence Diagram

Any card

Figure 24: Original CreateHelper Sequence Diagram

Figure 26: Original DeleteAgentOnStandaloneCard Sequence Diagram

Figure 27: Original DownloadGlobalTable Sequence diagram

Any card in the system

Figure 28: Original DownloadLocalTable Sequence Diagram

Figure 30: Original RegisterEphemeral Sequence Diagram

Figure 31: Original UpdateGlobalTable Sequence Diagram

Discovery Modified Sequence Diagrams

Figure 32: Modified AddAgent Sequence Diagram

Figure 33: Modified AddAgentOnStandaloneCard Sequence Diagram

Any card in the system

Figure 35: Modified CreateHelper Sequence Diagram

Figure 36: Modified DeleteAgent Sequence Diagram

Figure 38: Modified DownloadGlobalTable Sequence Diagram

Any card

Figure 39: Modified DownloadLocalTable Sequence Diagram

Figure 40: Modified RegisterAgent Sequence Diagram

Figure 41: Modified RegisterEphemeral Sequence Diagram

Figure 42: Modified UpdateGlobalTable Sequence Diagram

Data Dictionary

In this section, the different actors and classes that constitute the discovery system are described.

Actors:

Agent

An agent contains instances of many different objects as needed. The objects contained in each agent are defined at compile time. Agents can start and terminate at different times. When an agent starts, it needs to register with every agent in the system through the discovery subsystem. This agent gets a copy of the global discovery table that contains all the agents in the system. Every agent in the system must contain an instance of a discovery agent object (local, boot or ephemeral).

Card

A card with one or more agents registered on it.

NodalSurveillanceAgent

The nodal surveillance agent is part of the surveillance subsystem. It exists in every agent in the system. It monitors the agents and notifies the discovery system if an agent terminates. The main function of the surveillance subsystem is to detect terminated agents and report them to the discovery subsystem to delete them from the discovery tables.

StandaloneAgent

An agent is starting on a standalone card. This agent follows the same procedures to register as Agent except that he registers with his ephemeral discovery agent instead of the global discovery agent.

Boundary Classes

BootDiscoveryAgent

This class handles all the local registration messages from the local agents on the same card. It has an internal interface to the local discovery agent, the ephemeral discovery agent and the global discovery agent and an external interface to the nodal surveillance agent of the surveillance subsystem. There is one instance of this class per card in the BootAgent.

inv: BootAgent->includes(BootDiscoveryAgent)

Entity Classes

BaseDiscovery

It is an abstract base class that contains many of the methods used by other classes in the system. All the discovery agent classes inherit from this base class.

EphemeralDiscoveryAgent

This class handles all the local registration messages directed to the global discovery agent in the absence of the global discovery agent and the control card. There is only one instance per card of this class in the BootAgent. It has internal interfaces to the boot discovery agent and the global discovery agent.

inv: BootAgent->includes(EphemeralDiscoveryAgent)

GlobalDiscoveryAgent

This class handles all the global registration messages. There is only one instance of this class in the BootAgent of the control card. This class handles the updates to the global discovery table. It has interfaces to the boot discovery agent and the ephemeral discovery agent.

inv: BootAgent->includes(GlobalDiscoveryAgent)

Helper

This helper process is created as needed and terminated as soon as its role is done. The boot discovery agent creates it to handle the discovery table download from the global discovery agent. Usually the table is big and takes several messages to complete.

LocalDiscoveryAgent

The local discovery agent class handles the discovery registration of any agent in the system. There is one instance of this class in every agent in the system. The local discovery agent has an internal interface to the boot discovery agent.

Control Classes

ObjPid

This class is a base class for any object in the system. It is used to provide a unique object ID to every object in the system.

Process

This class is a base class for any process in the system. It is used to provide a unique process ID to each process in the system. A process contains one or more objects.

Timer

This is a timer class used by all the objects in the system that needs a timer.

Attributes

agentInformationTable	Table that contains all the discovered agents in the system, it contains: objPid, IP_address, UDP_port, parent and level.
agentMap	Map used to keep track of messages sent and their corresponding acknowledgments.
bootTimerId	Integer, timer ID, used to keep track of messages sent from the boot discovery agent.
cardInformationTable	Table specific to one standalone card, contains all the local agents on that card. It contains: objPid, IP_address, UDP_port, parent and level.
downloadPending	Boolean used to indicate if a table download process is pending or not.
ephemeralTimerId	Integer, timer ID, used to track messages sent from the ephemeral discovery agent.
globalInformationTable	Table that contains all the discovered agents in the system. It contains: objPid, IP_address, UDP_port, parent and level.
globalPid	Integer, the global discovery objPid, used to communicate with the global discovery agent. It is stored in both the BootDiscoveryAgent and the EphemeralDiscoveryAgent.

globalTimerId	Integer, timer ID, used to keep track of messages sent from the global discovery agent.
helperExists	Boolean records whether a helper process exists or not. It is set to true, if the helper exists, and false otherwise.
helperPid	Boot discovery agent helper object ID, it is used to communicate with it.
helperTimerId	Integer, timer ID, used to keep track of messages sent from the boot discovery agent helper.
localTimerId	Integer, timer ID, used to keep track of messages sent from the local discovery agent.
objPid	Integer specifies an object ID.
processId	Integer specifies a process ID.
sequenceNumber	Integer indicating which part of the table has been received.
timerId	Integer indicating the timer ID. It is allocated when the timer is created.

Added Attributes

agentMap	Map added to the LocalDiscoveryAgent set of attributes. It is used to keep track of messages sent from the local discovery agent and their acknowledgments.
-----------------	---

Changed Attributes

agentInformationTable	Table that contains all the discovered agents in the system, it contains: objPid, IP_address, UDP_port and parentLevel. The parent and level fields were merged into one field due to table size constraints in large networks.
cardInformationTable	Table specific to the card that contains all the local agents on that card. It contains: objPid, IP_address, UDP_port and parentLevel. Same as before the parent and level fields were merged into one field due to size constraints.
globalInformationTable	Table of all the discovered agents in the system. It contains: objPid, IP_address, UDP_port and parentLevel. Same as before the parent and level fields were merged into one field due to size constraints.

Deleted Attributes

agentMap	Map deleted from the BootDiscoveryAgent set of attributes.
bootTimerId	Integer, timer ID, used to keep track of messages sent from the boot discovery agent.
globalTimerId	Integer, timer ID, used to keep track of messages sent from the global discovery agent.

Operations

BaseDiscovery::agentTableDownload(table, seqNumber) : Void

This is pure virtual function.

BaseDiscovery::agentTableDownloadComplete(seqNumber) : Void

This is pure virtual function.

BaseDiscovery::agentTableDownloadCompleteAck(seqNumber) : Void

This is pure virtual function.

BaseDiscovery::deleteAgent(objPid) : Void

This is pure virtual function.

BaseDiscovery::deleteAgentReq(objPid) : Void

This is pure virtual function.

BaseDiscovery::deleteAgentReqAck(objPid) : Void

This is pure virtual function.

BaseDiscovery::getAgentTable() : Void

This is pure virtual function.

**BaseDiscovery::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void**

This is pure virtual function.

BaseDiscovery::IamAck(objPid) : Void

This is pure virtual function.

**BaseDiscovery::updateAgentTable(objPid, IPAddress, UDPPort, agentParent,
agentLevel) : Void**

This is pure virtual function.

BootDiscoveryAgent::agentDiscoveryCompleted(objPid) : Void

An indication that the starting local discovery agent finished the discovery process.

pre: LocalDiscoveryAgent.agentInformationTable->notEmpty

BootDiscoveryAgent::agentTableDownload(globalTable, seqNumber) : Void

This message downloads the discovery table from the EphemeralDiscoveryAgent.

pre: EphemeralDiscoveryAgent.cardInformationTable->notEmpty

BootDiscoveryAgent::agentTableDownloadCompleteAck(seqNumber) : Void

This message is received to indicate that the discovery table has been successfully downloaded to the LocalDiscoveryAgent.

post: LocalDiscoveryAgent.agentInformationTable->notEmpty

BootDiscoveryAgent::createHelper(objPid, IPAddress, UDPPort, agentParent, agentLevel, globalPid) : Void

Creates a helper to handle the table download process from the global discovery agent.

pre: self.helperExists = false

post: self.helperExists = true

BootDiscoveryAgent::deleteAgentReq(objPid) : Void

Sent from the GlobalDiscoveryAgent to request the removal of the terminated agent from the LocalDiscoveryAgent's tables. The BootDiscoveryAgent forwards this message to his LocalDiscoveryAgent's.

pre: Not GlobalDiscoveryAgent.globalInformationTable->

exists(agent : ObjPid | agent.objPid = objPid)

BootDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Received from the LocalDiscoveryAgent's after deleting the terminated agent from their tables.

pre: Not LocalDiscoveryAgent.agentInformationTable->

exists(agent : ObjPid | agent.objPid = objPid)

BootDiscoveryAgent::globalPidSet(globalPid) : Void

Updates the globalPid to the passed in value.

post: self.globalPid = globalPid

BootDiscoveryAgent::helperExistsGet() : Boolean

Returns the value of the helperExists flag.

BootDiscoveryAgent::helperExistsSet(helperStatus) : Void

Sets the helperExists data member to the passed in value. True if a helper was created and false if the helper was terminated.

post: self.helperExists = helperStatus

**BootDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void**

This is a registration message sent by every new agent in the network. It contains the objPid, IPAddress, UDPPort, the agent parent (global or boot), the agent level (1 for global, 2 for boot, and 3 for local), and the local agents list that needs to be contacted.

pre: LocalDiscoveryAgent.agentInformationTable->

exists(agent : ObjPid | agent.objPid = objPid)

BootDiscoveryAgent::IamAck(objPid) : Void

This is an acknowledgment corresponding to a pending Iam message. It contains the agent process id that was sent in the Iam message.

post: self.helperExists = true

BootDiscoveryAgent::newGlobal(globalPid) : Void

Updates the address of the global discovery agent in the boot discovery agent.

pre: EphemeralDiscoveryAgent.globalPid > 0

post: self.globalPid = globalPid

BootDiscoveryAgent::nodeDown(objPid) : Void

Received from the nodal surveillance agent as a result of an agent termination.

BootDiscoveryAgent::updateAgentMap(objPid) : Void

The agent map contains the discovery agents in the network and a flag indicating if a certain discovery agent acknowledged the registration message sent to it.

pre: self.agentMap->exists(agent : ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::agentDown(objPid) : Void

Received from the boot discovery agent to indicate the terminated agent ID.

EphemeralDiscoveryAgent::agentTableDownloadCompleteAck(seqNumber) :

Void

This message is received to indicate that the discovery table has been successfully downloaded to the **BootDiscoveryAgent**.

EphemeralDiscoveryAgent::deleteAgent(objPid) : Void

Updates the card discovery table to delete the terminated agent.

post: Not self.cardInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Received from the boot discovery agent after deleting the terminated agent from the local discovery agent tables.

pre: Not self.agentInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::edaLockup() : Void

Locks the ephemeral discovery agent to ignore all subsequent registration messages from the boot discovery agent in preparation for the transfer of the card discovery agent table to the **GlobalDiscoveryAgent**.

pre: self.globalPid > 0 and self.globalInformationTable->notEmpty

EphemeralDiscoveryAgent::edaUnlock() : Void

Unlocks the ephemeral discovery agent after the global discovery table has been updated with the card agents.

pre: GlobalDiscoveryAgent.globalInformationTable->includesAll(
self.cardInformationTable)

EphemeralDiscoveryAgent::globalPidSet(globalPid) : Void

Sets the globalPid to the passed in value.

post: self.globalPid = globalPid

EphemeralDiscoveryAgent::IamAck(globalPid) : Void

This is an acknowledgment from the GlobalDiscoveryAgent corresponding to a pending Iam message. It contains the agent process id that was sent in the Iam message.

pre: GlobalDiscoveryAgent.globalInformationTable->notEmpty

post: self.globalPid = globalPid

EphemeralDiscoveryAgent::joinNetwork() : Void

Whenever a new card starts up, this message triggers the ephemeral discovery agent to contact the boot discovery agent of the control card.

EphemeralDiscoveryAgent::newGlobalAck(globalPid) : Void

Received from his boot discovery agent after receiving the new global discovery agent table.

pre: BootDiscoveryAgent.helperExists = false

EphemeralDiscoveryAgent::terminate() : Void

This message is invoked to terminate the EphemeralDiscoveryAgent.

pre: BootDiscoveryAgent.helperExists = false

EphemeralDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel) : Void

This message is sent whenever a new agent starts from the BootDiscoveryAgent. It contains the agent information objPid, IPAddress, UDPPort, the agent parent and level.

post: self.cardInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::agentDown(objPid) : Void

Received from the boot discovery agent to indicate the terminated agent ID.

GlobalDiscoveryAgent::agentTableDownload(globalTable, seqNumber) : Void

This message uploads the cardInformationTable from the EphemeralDiscoveryAgent to the GlobalDiscoveryAgent.

pre: EphemeralDiscoveryAgent.cardInformationTable->notEmpty

GlobalDiscoveryAgent::agentTableDownloadCompleteAck(seqNumber) : Void

This message is received to indicate that the discovery table has been successfully downloaded to the BootDiscoveryAgent helper.

pre: Helper.globalPid > 0

post: self.downloadPending = false

GlobalDiscoveryAgent::deleteAgent(objPid) : Void

Updates the global discovery table to delete the terminated agent.

post: Not self.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Received from the boot discovery agents after deleting the terminated agent from their local discovery agent tables.

pre: Not self.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

post: Not LocalDiscoveryAgent.agentInformationTable->exists(agent :
ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::downloadPendingGet() : Boolean

Return the downloadPending flag value.

GlobalDiscoveryAgent::downloadPendingSet(downloadPending) : Void

Sets the downloadPending flag to the passed in value. True if there is a
download process pending, false otherwise.

pre: self.globalInformationTable->notEmpty

post: self.downloadPending = downloadPending

GlobalDiscoveryAgent::getAgentTable() : Void

Request to get the global table downloaded.

pre: self.helperPid > 0

post: self.downloadPending = true

GlobalDiscoveryAgent::helperPidSet(helperPid) : Void

Set the helperPid data member.

pre: self.downloadPending = true

post: self.helperPid = helperPid

**GlobalDiscoveryAgent::senderPid(ephemeralPid, objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void**

Used to register the EphemeralDiscoveryAgent.

pre: BootDiscoveryAgent.globalPid > 0

**GlobalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void**

This message is sent whenever a new agent starts to register the agent with the GlobalDiscoveryAgent. It contains all the agent information.

pre: self.globalInformationTable->notEmpty

post: self.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

Helper::agentTableDownload(globalTable, seqNumber) : Void

This message downloads the globalInformationTable from the GlobalDiscoveryAgent.

pre: GlobalDiscoveryAgent.downloadPending = true and

GlobalDiscoveryAgent.globalInformationTable->notEmpty

Helper::agentTableDownloadComplete(seqNumber) : Void

This message is received to indicate that the discovery table has been finished downloading.

Helper::agentTableDownloadCompleteAck(seqNumber) : Void

This message is received to indicate that the discovery table has been successfully been received by the Helper.

pre: self.sequenceNumber = seqNumber

Helper::sequenceNumberGet() : Integer

Returns the stored sequence number.

Helper::sequenceNumberSet(seqNumber) : Void

Sets the sequence number.

post: self.sequenceNumber = seqNumber

Helper::terminate() : Void

This message is invoked to terminate the helper.

pre: GlobalDiscoveryAgent.downloadPending = false

post: BootDiscoveryAgent.helperExists = false

Helper::updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel)

: Void

This message is used to temporary store the downloaded global discovery table.

pre: self.sequenceNumber > 0

LocalDiscoveryAgent::agentTableDownload(globalTable, seqNumber) : Void

This message downloads the globalInformationTable from the BootDiscoveryAgent helper.

pre: GlobalDiscoveryAgent.globalInformationTable->notEmpty

LocalDiscoveryAgent::agentTableDownloadComplete(seqNumber) : Void

This message is sent at the end of the table download to indicate that the discovery table has finished downloading.

pre: self.agentInformationTable->notEmpty

LocalDiscoveryAgent::deleteAgent(objPid) : Void

Updates the agent discovery table to delete the terminated agent.

pre: Not GlobalDiscoveryAgent.globalInformationTable->exists(agent :
ObjPid | agent.objPid = objPid)

post: Not self.agentInformationTable->exists(agent : ObjPid |
agent.objPid = objPid)

LocalDiscoveryAgent::deleteAgentReq(objPid) : Void

Sent to request the removal of the terminated agent from the discovery table.

pre: Refer to LocalDiscoveryAgent::deleteAgent pre-condition.

**LocalDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void**

This is a registration message sent by every new agent in the network. It contains the objPid, IPAddress, UDPPort, the agent parent (global or boot), the agent level (1 for global, 2 for boot, and 3 for local), and the local agents list that needs to be contacted.

pre: Refer to LocalDiscoveryAgent::agentTableDownloadComplete pre-condition.

LocalDiscoveryAgent::IamAck(objPid) : Void

This is an acknowledgment corresponding to a pending Iam message. It contains the agent process id that was sent in the Iam message.

pre: Refer to LocalDiscoveryAgent::agentTableDownloadComplete pre-condition

post: self.agentInformationTable->exists(agent : ObjPid | agent.objPid =
objPid)

LocalDiscoveryAgent::registerAgent(objPid) : Void

This message is received whenever a new agent starts in the system. It triggers the local discovery agent of the new agent to register with the discovery system.

pre: self.agentInformationTable->isEmpty

**LocalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void**

This message is used to update the agent discovery table. It contains all the agent information.

post: self.agentInformationTable->exists(agent : ObjPid | agent.objPid =
objPid)

ObjPid::getObjPid() : Integer

Returns the agent process ID.

ObjPid::setObjPid(objPid) : Void

Sets the agent process ID.

post: self.objPid = objPid

Process::create() : Integer

Used to create a new process. The process ID is returned

Process::exists(objPid) : Boolean

Used to check if a certain process is running. It returns true if the process is running and false otherwise.

Process::terminate() : Void

Used to terminate a process.

Timer::startTimer(timerPeriod) : Void

Starts the specified timer with the specified period.

pre: self.timerId > 0

Timer::stopTimer() : Void

Stops the specified timer.

pre: self.timerId > 0

Timer::timeOut() : Void

It is invoked when the timer expires.

Added Operations

BaseDiscovery::Iam(objPid, IPAddress, UDPPort, agentParentLevel, agentList) :

Void

This is pure virtual function.

BaseDiscovery::updateAgentTable(objPid, IPAddress, UDPPort,

agentParentLevel) : Void

This is pure virtual function.

BootDiscoveryAgent::createHelper(objPid, IPAddress, UDPPort,

agentParentLevel, globalPid) : Void

Creates a helper to handle the table download process from the global discovery agent.

pre: self.helperExists = false

post: self.helperExists = true

BootDiscoveryAgent::globalPidGet() : Integer

Returns the global discovery objPid.

**BootDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParentLevel,
agentList) : Void**

This is a registration message sent by every new agent in the network. It contains the objPid, IPAddress, UDPPort, the agent parent (global or boot), the agent level (1 for global, 2 for boot, and 3 for local), and the local agents list that needs to be contacted.

pre: LocalDiscoveryAgent.agentInformationTable->
exists(agent : ObjPid | agent.objPid = objPid)

BootDiscoveryAgent::selfTerminate() : Void

Received from the boot discovery agent helper before it terminate.

pre: LocalDiscoveryAgent.agentInformationTable->notEmpty

post: self.helperExists = false

EphemeralDiscoveryAgent::deleteAgentReq(objPid) : Void

Sent to request the removal of the terminated agent from the card discovery table.

pre: Not LocalDiscoveryAgent.agentInformationTable->exists(agent :
ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::getAgentTable() : Void

pre: self.cardInformationTable->notEmpty

EphemeralDiscoveryAgent::globalPidGet() : Integer

Returns the global discovery objPid.

**EphemeralDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParentLevel) : Void**

This message is sent whenever a new agent starts from the
BootDiscoveryAgent. It contains the agent information objPid, IPAddress,
UDPPort, the agent parent and level.

post: self.cardInformationTable->exists(agent : ObjPid | agent.objPid =
objPid)

GlobalDiscoveryAgent::deleteAgentReq(objPid) : Void

Sent to request the removal of the terminated agent from the global
discovery table.

pre: Not LocalDiscoveryAgent.agentInformationTable->exists(agent :
ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::helperPidGet() : Integer

Returns the helperPid data member value.

**GlobalDiscoveryAgent::senderPid(ephemeralPid, objPid, IPAddress, UDPPort,
agentParentLevel) : Void**

Used to register the EphemeralDiscoveryAgent.

pre: BootDiscoveryAgent.globalPid > 0

**GlobalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParentLevel) : Void**

This message is sent whenever a new agent starts to register the agent with
the GlobalDiscoveryAgent. It contains all the agent information.

pre: self.globalInformationTable->notEmpty

post: self.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

Helper::updateAgentTable(objPid, IPAddress, UDPPort, agentParentLevel) :

Void

This message is used to temporary store the downloaded global discovery table.

pre: self.sequenceNumber > 0

LocalDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Sent as an acknowledgment to the message deleteAgentReq after deleting the terminated agent from the agent table.

pre: Not GlobalDiscoveryAgent.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

LocalDiscoveryAgent::getAgentTable() : Void

Request to get the local discovery agent table.

pre: self.agentInformationTable->notEmpty

LocalDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParentLevel, agentList) : Void

This is a registration message sent by every new agent in the network. It contains the objPid, IPAddress, UDPPort, the agent parentLevel, and the local agents list that needs to be contacted.

pre: Refer to LocalDiscoveryAgent::agentTableDownloadComplete pre-condition.

LocalDiscoveryAgent::nodeDown(objPid) : Void

Received from the nodal surveillance agent to indicate that an agent has terminated.

LocalDiscoveryAgent::updateAgentMap(objPid) : Void

The agent map contains the boot discovery agents in the network, the corresponding agent process ID and a flag indicating if the boot discovery agent acknowledged the registration message sent to it.

pre: self.agentMap->exists(agent : ObjPid | agent.objPid = objPid)

**LocalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParentLevel) : Void**

This message is used to update the agent discovery table. It contains all the agent information.

post: self.agentInformationTable->exists(agent : ObjPid | agent.objPid =
objPid)

Changed Operations

There are no changed operations between the original and modified version of the discovery system.

Deleted Operations

**BaseDiscovery::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void**

This is pure virtual function.

**BaseDiscovery::updateAgentTable(objPid, IPAddress, UDPPort, agentParent,
agentLevel) : Void**

This is pure virtual function.

**BootDiscoveryAgent::createHelper(objPid, IPAddress, UDPPort, agentParent,
agentLevel, globalPid) : Void**

Creates a helper to handle the table download process from the global discovery agent.

pre: self.helperExists = false

post: self.helperExists = true

**BootDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void**

This is a registration message sent by every new agent in the network. It contains the objPid, IPAddress, UDPPort, the agent parent (global or boot), the agent level (1 for global, 2 for boot, and 3 for local), and the local agents list that needs to be contacted.

pre: LocalDiscoveryAgent.agentInformationTable->
exists(agent : ObjPid | agent.objPid = objPid)

BootDiscoveryAgent::nodeDown(objPid) : Void

Received from the nodal surveillance agent as a result of an agent termination.

BootDiscoveryAgent::updateAgentMap(objPid) : Void

The agent map contains the discovery agents in the network and a flag indicating if a certain discovery agent acknowledged the registration message sent to it.

pre: self.agentMap->exists(agent : ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::agentDown(objPid) : Void

Received from the boot discovery agent to indicate the terminated agent ID.

EphemeralDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Received from the boot discovery agent after deleting the terminated agent from the local discovery agent tables.

pre: Not self.agentInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

EphemeralDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel) : Void

This message is sent whenever a new agent starts from the BootDiscoveryAgent. It contains the agent information objPid, IPAddress, UDPPort, the agent parent and level.

post: self.cardInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::agentDown(objPid) : Void

Received from the boot discovery agent to indicate the terminated agent ID.

GlobalDiscoveryAgent::deleteAgentReqAck(objPid) : Void

Received from the boot discovery agents after deleting the terminated agent from their local discovery agent tables.

pre: Not self.globalInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

post: Not LocalDiscoveryAgent.agentInformationTable->exists(agent :
ObjPid | agent.objPid = objPid)

GlobalDiscoveryAgent::senderPid(ephemeralPid, objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void

Used to register the EphemeralDiscoveryAgent.

pre: BootDiscoveryAgent.globalPid > 0

GlobalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void

This message is sent whenever a new agent starts to register the agent with
the GlobalDiscoveryAgent. It contains all the agent information.

pre: self.globalInformationTable->notEmpty

post: self.globalInformationTable->exists(agent : ObjPid | agent.objPid =
objPid)

Helper::updateAgentTable(objPid, IPAddress, UDPPort, agentParent, agentLevel)
: Void

This message is used to temporary store the downloaded global discovery
table.

pre: self.sequenceNumber > 0

LocalDiscoveryAgent::Iam(objPid, IPAddress, UDPPort, agentParent,
agentLevel, agentList) : Void

This is a registration message sent by every new agent in the network. It
contains the objPid, IPAddress, UDPPort, the agent parent (global or

boot), the agent level (1 for global, 2 for boot, and 3 for local), and the local agents list that needs to be contacted.

pre: Refer to LocalDiscoveryAgent::agentTableDownloadComplete pre-condition.

LocalDiscoveryAgent::updateAgentTable(objPid, IPAddress, UDPPort,
agentParent, agentLevel) : Void

This message is used to update the agent discovery table. It contains all the agent information.

post: self.agentInformationTable->exists(agent : ObjPid | agent.objPid = objPid)

Appendix B: Data Dictionary for the RTSTool Design

Boundary Classes

CompareGui:

The Gui responsible for the compare class diagrams or compare use case diagrams operation. It is created when the user selects “Compare Class Diagrams” or the “Compare Use Case Diagrams” menu items. It has two list selection menus that display all the loaded class diagrams/use case diagrams. The user must select the original diagram from the first list and the modified diagram from the second list. The two selected diagrams must be different.

LoadGui:

The Gui responsible for the load class diagram/use case diagram/regression test suite operation. It is created when the user selects “Load Class Diagram”, “Load Use Case Diagram” or “Load Regression Test Suite” menu items. It has two text boxes for the name and version of the diagram/test suite to be loaded. The user then selects the file that contains the class diagram/use case diagram in XMI format or the regression test suite in text format to be loaded. If the user enters the name and version of an existing diagram or test suite, it is retrieved from the database.

RTSToolGui:

This is the main GUI for the RTSTool, it is created on start up and it contains the message box (to display messages to the user), the consistency check and the regression test selection buttons. It has also a menu for the “Load” and “Compare” operations.

Entity Classes

ActualParameter:

This class represents an actual parameter associated with an invoked method in a sequence diagram or a test case. An actual parameter is represented by its type and value. The UseCaseDiagram and the RegressionTestSuite subsystems use this class.

Attribute:

This class represents a class attribute. Each attribute is represented by its name, class name, type, scope (private, public, protected) and visibility (static, etc). The access methods used to manipulate the attribute are represented by the association link "accessMethods".

Class:

This class represents an actual class from the class diagram. Each class is represented by its name. This class contains methods to manipulate the different methods and attributes of the different classes.

inv: self.classAttributes->size >= 0 **and** self.classMethods->size >= 1 **and**
self.classRelationships->size >= 0

ClassDiagram:

This class represents a class diagram. The name and version of the class diagram are stored in this class. This class stores the set of all the classes in the class diagram "classes". This class contains methods to detect the sets of added, changed and deleted attributes, classes and methods.

inv: self.classes->size >= 1

ConsistencyCheck:

This is an association class; it is associated with the `ClassDiagramChanges` and `UseCaseDiagramChanges` classes. This class checks the consistency of the changed methods (added, changed and deleted) between the use case diagram and the class diagram. The sets of unused defined methods and undefined invoked methods are stored in this class.

DefinedMethod:

This class represents a defined class method. Each defined method is represented by its name, type, scope (private, public, protected), visibility (static, virtual, etc), parent name, class name and the parameters associated with the method. This class is used by the `ClassDiagram` subsystem.

inv: self.methodParameters->size >= 0

FormalParameter:

This class represents an invoked method formal parameter. A formal parameter is represented by its name, type and direction (in, out or inout). This class is used by the `ClassDiagram` subsystem.

InvokedMethod:

This class represents an invoked method. An invoked method is represented by its name, type, class name and associated parameters. The `UseCaseDiagram` and the `RegressionTestSuite` subsystems use this class.

inv: self.methodParameters->size >= 0

RegressionTestSuite:

This class represents a regression test suite. The name and version of the regression test suite are stored in this class. This class has a set of all the test cases “testCases” in the regression test suite. It is responsible for classifying the different regression test cases as being obsolete, retestable or reusable.

inv: self.testCases->size > 0

Relationship:

This is an association class; it is associated with two different classes and it stores the relationship type (association, inheritance, aggregation), and the target class name to identify the target class.

TestCase:

This class represents a regression test case from the loaded regression test suite. The test case is represented by its name, type, version, and the regular expression that represents its sequence of messages.

inv: self.testCaseMethods->size >= 1

TestCaseClassification:

This class is associated with each test case in a particular regression test suite. It stores the test case classification with regards to that test suite. A test case can be classified as obsolete, retestable or reusable.

UseCase:

This class represents a use case from the loaded use case diagram. The use case is represented by its name, the regular expression that represents the sequence diagram associated with it and the methods called in the sequence diagram.

inv: self.useCaseMethods->size > 1

UseCaseDiagram:

This class represents a use case diagram. The name and version of the use case diagram are stored in this class. This class has a set of all the use cases “useCases” in the use case diagram. This class contains methods to detect the sets of added, changed and deleted methods and use cases.

inv: self.useCases->size >= 1

Control Classes

ClassDiagramChanges:

This is an association class; it is associated with two different class diagrams. This class manages the results of comparing the two associated class diagrams. The sets of added, changed and deleted attributes, classes and methods detected from comparing the two class diagrams are stored in this class.

inv: self.changedClasses->size <=
(self.classDiagram1.classes->size + self.classDiagram2.classes->size)

UseCaseDiagramChanges:

This is an association class; it is associated with two different use case diagrams. This class manages the results of comparing the two associated use case diagrams. The sets of added, changed and deleted methods and use cases are associated with this class.

inv: self.changedUseCases->size <=
(self.useCaseDiagram1.useCases->size + self.useCaseDiagram2.useCases->size)

Attributes:

accessMethods	Set associated with each attribute specifying the methods that uses the attribute.
addedAttributes	Set associated with the class diagram changes class specifying the set of added attributes between the two associated class diagrams.
addedClasses	Set associated with the class diagram changes class specifying the set of added classes between the two associated class diagrams.
addedMethods	Set associated with the ClassDiagramChanges/UseCaseDiagram changes class specifying the set of added methods between the two associated ClassDiagram's/UseCaseDiagram's.
addedUseCases	Set associated with the UseCaseDiagramChanges class specifying the set of added use cases between the two associated use case diagrams.
changedAttributes	Set associated with the ClassDiagramChanges class specifying the set of changed attributes between the two associated class diagrams.
changedClasses	Set associated with the ClassDiagramChanges class specifying the set of changed classes between the two associated class diagrams.
changedMethods	Set associated with the ClassDiagramChanges/UseCaseDiagram class specifying the set of

	changed methods between the two associated ClassDiagram's/UseCaseDiagram's.
changedUseCases	Set associated with the UseCaseDiagramChanges class specifying the set of changed use cases between the two associated UseCaseDiagram's.
classAttributes	Set associated with each class specifying the class attributes.
classDiagram1	Reference to the original class diagram. It is stored in the ClassDiagramChanges object.
classDiagram2	Reference to the modified class diagram. It is stored in the ClassDiagramChanges object.
classes	Set associated with each ClassDiagram specifying the class diagram classes.
Classification	Enumeration {obsolete, retestable, reusable}
classMethods	Set associated with each Class specifying the class methods.
ClassName	String identifies the class name of a specific attribute or method.
classRelationships	Set associated with each class specifying the class relationships.
deletedAttributes	Set associated with the ClassDiagramChanges class specifying the set of deleted attributes between the two associated class diagrams.

deletedClasses	Set associated with the ClassDiagramChanges class specifying the set of deleted classes between the two associated class diagrams.
deletedMethods	Set associated with the ClassDiagramChanges/UseCaseDiagramChanges class specifying the set of deleted methods between the two associated ClassDiagram's/UseCaseDiagrams .
deletedUseCases	Set associated with the UseCaseDiagramChanges class specifying the set of deleted use cases between the two associated use case diagrams.
Direction	Enumeration {in, out, inout}
methodParameters	Set associated with each method specifying the method parameters.
MgSS	Regular expression representing a sequence diagram.
Name	String identifies the name of an object.
obsoleteTestCases	Set associated with the regression test suite class specifying the set of obsolete test cases.
ParentName	String identifies the parent method to a specific defined method.
RelationshipType	Enumeration {inheritance, aggregation, association}
retestableTestCases	Set associated with the regression test suite object specifying the set of retestable test cases.

reusableTestCases	Set associated with the regression test suite object specifying the set of reusable test cases.
Scope	Enumeration {private, public, protected}
TargetClassName	String identifies the target class name of a specific class relationship.
testCase	Regular expression representing a test case.
testCaseClassification	Reference to the associated test case classification object. It is stored with the test case object.
testCaseMethods	Set associated with each TestCase object specifying the methods in the test case.
testCases	Set associated with the regression test suite object specifying the associated test cases.
Type	String identifies the type (integer, char, string, etc.) of an object.
useCaseDiagram1	Reference to the original associated UseCaseDiagram. It is stored in the UseCaseDiagramChanges object.
useCaseDiagram2	Reference to the modified associated UseCaseDiagram. It is stored in the UseCaseDiagramChanges object.
useCaseMethods	Set associated with each UseCase specifying the methods used in the use case sequence diagram.

useCases	Set associated with each UseCaseDiagram specifying the use case diagram useCases.
undefinedInvokedMethods	Set associated with the ConsistencyCheck class specifying the set of undefined invoked methods.
unusedDefinedMethods	Set associated with the ConsistencyCheck class specifying the set of unused defined methods.
Value	String indicates the actual value of a parameter.
Version	Real identifies the version number of the regression test suite, class diagram, use case diagram or a test case.
Visibility	Enumeration {static, extern, virtual}

Operations

ActualParameter::ActualParameter(parameterValue, parameterType) : Void

Creates an ActualParameter with the passed-in value and type.

ActualParameter::compareParameters(parameter) : Boolean

Compares the passed-in parameter with itself and returns true if they are the same otherwise false.

ActualParameter::getType() : String

Returns the type of the parameter.

ActualParameter::getValue() : String

Returns the value of the parameter.

ActualParameter::setType(parameterType) : Void

Sets the type of the parameter.

ActualParameter::setValue(parameterValue) : Void

Sets the value of the parameter.

**Attribute::Attribute(attributeName, attributeName, attributeScope,
attributeType, attributeVisibility) : Void**

Creates an Attribute with the passed-in name, class name, scope, type and visibility.

Attribute::addAttributeAccessMethod(accessMethod) : Void

Adds the passed-in method to the attribute set of access methods.

Attribute::compareAttributes(attribute) : Boolean

Compares the passed-in attribute with itself and returns true if they are the same otherwise false.

Attribute::getClassName() : String

Returns the class name of the attribute.

Attribute::getName() : String

Returns the name of the attribute.

Attribute::getScope() : Integer

Returns the scope of the attribute: private, public, protected.

Attribute::getType() : String

Returns the return-type of the attribute.

Attribute::getVisibility() : Integer

Returns the visibility of the attribute: static, extern or virtual.

Attribute::setClassName(className) : Void

Sets the class name of the attribute.

Attribute::setName(name) : Void

Sets the name of the attribute.

Attribute::setScope(scope) : Void

Sets the scope of the attribute.

Attribute::setType(type) : Void

Sets the attribute type.

Attribute::setVisibility(visibility) : Void

Sets the visibility of the attribute.

Class::addClassAttribute(attribute) : Void

Adds the passed-in attribute to the class set of attributes.

Class::addClassMethod(method) : Void

Adds the passed-in defined method to the class set of methods.

Class::addClassRelationship(relationship) : Void

Adds the passed-in relationship to the class set of relationships.

Class::Class(className) : Void

Creates a Class with the passed-in name.

Class::compareClassAttributes(attributes) : Boolean

Compares the passed-in set of attributes with its set of attributes and returns true if the two classes contains the same set of attributes otherwise false.

Class::compareClasses(class) : Boolean

Compares the passed-in class with itself and returns true if the two classes are the same otherwise false.

Class::compareClassMethods(methods) : Boolean

Compares the passed-in set of methods with its set of methods and returns true if the two classes contains the same set of methods otherwise false.

Class::compareClassRelationships(relationships) : Boolean

Compares the passed-in set of relationships with its set of relationships and returns true if the two classes contains the same set of relationships otherwise false.

Class::createClassAttribute(attributeName, attributeClassName, attributeScope, attributeType, attributeVisibility) : Attribute

Creates an attribute with the specified name, class name, scope, type and visibility. The attribute is then added to the class set of attributes.

Class::createClassMethod(methodName, parentName, methodScope, methodType, methodVisibility) : DefinedMethod

Creates a DefinedMethod with the specified name, parent name, class name, scope, type and visibility. The method is then added to the class set of methods.

Class::createClassRelationship(relationshipType, targetClassName) : Relationship

Creates a Relationship with the specified type and targetClassName. The relationship is then added to the class set of relationships.

Class::findChangedClassAttributes(attributes) : Set of Attribute

Compares between the passed in set of attributes and the class set of attributes and detects the changed attributes. A changed attribute is an attribute that has the same name and class name but has a changed type, scope, or visibility.

Class::findChangedClassMethods(methods) : Set of DefinedMethod

Compares between the passed in set of methods and the class set of methods and detects the changed methods. A changed method is a method that has the same name, class name and same number of parameters but a changed type, scope, visibility, parent, parameter type, or parameter direction.

Class::getName() : String

Returns the name of the Class.

Class::setName(name) : Void

Sets the name of the Class.

ClassDiagram::createClass(className) : Class

Creates a Class with the specified name and add it to the class diagram set of classes.

ClassDiagram::findChangedAttributes(classes) : Set of Attribute

Compares between the passed in set of classes and it's set of classes by calling the `Class::findChangedClassAttributes` on every two corresponding classes that share the same name.

ClassDiagram::findChangedClasses(classes) : Set of Class

Compares between the passed in set of classes and it's set of classes by calling the `self.compareClassAttributes`, `self.compareClassMethods`, `self.compareClassRelationships` on every two corresponding classes that share the same name.

ClassDiagram::findChangedMethods(classes) : Set of DefinedMethod

Compares between the passed in set of classes and it's set of classes by calling the `Class::findChangedClassMethods` on every two corresponding classes that share the same name.

ClassDiagram::getName() : String

Returns the name of the ClassDiagram.

ClassDiagram::getVersion() : Real

Returns the version of the ClassDiagram.

ClassDiagram::setName(name) : Void

Sets the name of the Class.

ClassDiagram::setVersion(version) : Void

Sets the version of the Class.

ClassDiagram::load(file) : Void

Loads a specific file that contains the classDiagram in XMI format into the RTSTool.

ClassDiagram::setAllMethodsAndAttributes() : Void

Creates the sets of allAttributes and allMethods by combining the sets of attributes/methods from all the classes in the class diagram.

ClassDiagramChanges::associateClassDiagrams(classDiagram1, classDiagram2)

: Void

Associates the ClassDiagramChanges object with the passed-in ClassDiagram's.

ClassDiagramChanges::ClassDiagramChanges() : Void

Creates a ClassDiagramChanges object.

ClassDiagramChanges::findClassDiagram(nameVersion) : ClassDiagram

Tries to find a ClassDiagram in the database with the specified name and version, if found it is returned otherwise the user is asked to specify the XMI file that contains it and it is loaded into the tool.

ClassDiagramChanges::findClassDiagramChanges() : Void

Compares the different class diagrams to find the sets of added, changed and deleted attributes/classes/methods.

ClassDiagramChanges::handleAttributeChanges() : Void

Maps the attribute changes to their corresponding access methods. As the test cases only include methods, hence the need to map attribute changes to method changes.

ClassDiagramChanges::handleMethodInheritance() : Void

Handles the method inheritance feature. A deleted method that has a parent method that can be called instead of it, is considered a changed method in the child class and is added to the changed methods set and removed from the deleted methods set.

ClassDiagramChanges::showAttributes(attributes) : Void

Prints the passed-in set of attributes.

ClassDiagramChanges::showClasses(classes) : Void

Prints the passed-in set of classes.

ClassDiagramChanges::showMethods(methods) : Void

Prints the passed-in set of methods.

ClassDiagramChanges::showParameters(parameters) : Void

Prints the passed-in set of parameters.

ConsistencyCheck::ConsistencyCheck() : Void

Creates a ConsistencyCheck object to perform the consistency check between the results obtained from comparing the class diagrams and use case diagrams.

**ConsistencyCheck::checkMethodSets(classDiagramMethods,
useCaseDiagramMethods) : Void**

Checks the two sets of passed methods, if some methods exists in the class diagram set but not in the use case diagram set, these methods are added to the set of unusedDefinedMethods. On the other hand, if a method exists in the use case diagram set of methods and not in the class diagram set of methods, it is added to set of undefinedInvokedMethods.

ConsistencyCheck::findInconsistentMethods() : Void

Checks that the added and deleted methods detected from the use case diagram are the same as the methods detected from the class diagram. The methods detected from the class diagram may be more than the methods detected from the use case diagrams as some of the methods maybe only used internally. The sets of inconsistent methods (that appear in one of the sets and not in the other) are added to either the set of undefinedInvokedMethods or the set of unusedDefinedMethods.

DefinedMethod::addMethodParameter(parameter) : Void

Adds the passed-in formal parameter to the method set of parameters.

DefinedMethod::compareMethodParameters(methodParameters) : Boolean

Compares the passed-in set of parameters with its set of parameters and returns true if the two method parameters are the same otherwise false.

DefinedMethod::compareMethods(method) : Boolean

Compares the passed-in method with itself and returns true if the two methods are the same otherwise false.

DefinedMethod::convertToInvokedMethod() : InvokedMethod

Creates an invoked method with its parameters (name, className, type) and returns it. The method formal parameters are converted also to actual parameters.

DefinedMethod::createMethodParameter(name, type, direction)

: FormalParameter

Creates a formal parameter with the specified name, type and direction and add it to its set of parameters then returns it.

**DefinedMethod::DefinedMethod(methodName, methodParentName,
methodScope, methodType, methodClassName,
methodVisibility) : Void**

Creates a DefinedMethod with the passed-in name, parentName, scope, type, class name and visibility.

DefinedMethod::getClassName() : String

Returns the method class name.

DefinedMethod::getName() : String

Returns the method name.

DefinedMethod::getParentName() : String

Returns the name of the parent class that contains the parent method.

DefinedMethod::getScope() : Integer

Returns the method scope.

DefinedMethod::getType() : String

Returns the method type.

DefinedMethod::getVisibility() : Integer

Returns the method visibility.

DefinedMethod::parentExists(method) : Boolean

Returns true if the method has a parent method otherwise false.

DefinedMethod::setClassName(className) : Void

Sets the method class name.

DefinedMethod::setName(name) : Void

Sets the method name.

DefinedMethod::setParentName(parentName) : Void

Sets the name of the parent class that contains the parent method.

DefinedMethod::setScope(scope) : Void

Sets the defined method scope.

DefinedMethod::setType(type) : Void

Sets the method type.

DefinedMethod::setVisibility(visibility) : Void

Sets the defined method visibility.

FormalParameter::compareParameters() : Boolean

Compares the passed-in parameter with itself and returns true if they are the same otherwise false.

FormalParameter::convertToActualParameter() : ActualParameter

Converts a formal parameter to an actual parameter.

FormalParameter::FormalParameter(name, type, direction) : Void

Creates a formal parameter with the specified name, type and direction.

FormalParameter::getDirection() : Integer

Returns the parameter direction (in, out, inout).

FormalParameter::getName() : String

Returns the parameter name.

FormalParameter::getType() : String

Returns the parameter type.

FormalParameter::setDirection(direction) : Void

Sets the parameter direction.

FormalParameter::setName(name) : Void

Sets the parameter name.

FormalParameter::setType(type) : Void

Sets the parameter type.

**InvokedMethod::InvokedMethod(methodName, methodType, methodClassName)
: Void**

Creates an InvokedMethod with the passed-in name, type and class name.

InvokedMethod::addMethodParameter(actualParameter) : Void

Adds the passed-in actual parameter to the method set of parameters.

InvokedMethod::compareMethodParameters(methodParameters) : Boolean

Compares the passed-in set of parameters with itself and returns true if the two method parameters are the same otherwise false.

InvokedMethod::compareMethods(method) : Boolean

Compares the passed-in method with itself and returns true if the two methods are the same otherwise false.

InvokedMethod::createMethodParameter(parameterType, parameterValue) :

ActualParameter

Creates an actual parameter with the specified type and value and add it to the method set of parameters.

InvokedMethod::getClassName() : String

Returns the method class name.

InvokedMethod::getName() : String

Returns the method name.

InvokedMethod::getType() : String

Returns the method return-type.

InvokedMethod::setClassName(className) : Void

Sets the method class name.

InvokedMethod::setName(name) : Void

Sets the method name.

InvokedMethod::setType(type) : Void

Sets the method type.

RegressionTestSuite::addTestCase(testCase) : Void

Adds the passed-in test case to the regression test suite set of test cases.

RegressionTestSuite::classifyTestCases() : Void

Classifies the different regression test cases in the test suite based on the their sequence of messages and the methods they contain.

RegressionTestSuite::createTestCase(name, type, testCase, version) : TestCase

Creates a test case object with the specified name, type, testCase, and version, the test case is also added to the regression test suite set of test cases.

RegressionTestSuite::getName() : String

Returns the regression test suite name.

RegressionTestSuite::getVersion() : Real

Returns the regression test suite version.

RegressionTestSuite::load(file) : Void

Loads the RegressionTestSuite from the specified text file.

RegressionTestSuite::mergeSets(useCaseMethods, classMethods) : Set of

InvokedMethod

Converts the classMethods into InvokedMethods and merges them with the useCaseMethods. The resulting set is returned.

RegressionTestSuite::regressionTestSelection() : Void

Classifies the different regression test cases in the test suite by calling classifyTestCases and creates the sets of obsolete, retestable and reusable test cases.

RegressionTestSuite::RegressionTestSuite(name, version) : Void

Creates a regression test suite with the specified name and version.

RegressionTestSuite::setName(name) : Void

Sets the regression test suite name.

RegressionTestSuite::setVersion(version) : Void

Sets the regression test suite version.

RegressionTestSuite::showTestCases(testCases) : Void

Displays the passed-in set of test cases.

RegressionTestSuite::testCaseMatcher(allMgSS, testCase) : Boolean

Tries to match the testCase against one of the MgSS's if the test case is a generated test case based on the sequence diagrams patterns. Returns true if the test case matches one of the MgSS's otherwise false.

Relationship::getRelationshipType() : Integer

Returns the relationship type {inheritance, aggregation, association}

Relationship::getTargetClassName() : String

Returns the target class name.

Relationship::Relationship(type, targetClassName) : Void

Creates a Relationship object with the specified type, and targetClassName.

Relationship::setRelationshipType(type) : Void

Sets the relationship type.

Relationship::setTargetClassName(targetClassName) : Void

Sets the target class name.

TestCase::addTestCaseMethod(InvokedMethod) : Void

Adds an invoked method to the test case set of methods.

**TestCase::createTestCaseMethod(methodName, methodReturnType,
methodClassName) : InvokedMethod**

Creates a test case method with the specified name, return-type and class name and returns it if it doesn't exist in the database otherwise the method is retrieved from the database and returned. The method is also added to the test case set of methods.

TestCase::getName() : String

Returns the test case name.

TestCase::getTestCase() : String

Returns the test case string of methods.

TestCase::getType() : String

Returns the test case type.

TestCase::getVersion() : Real

Returns the test case version.

TestCase::setName(name) : Void

Sets the test case name.

TestCase::setTestCase(testCase) : Void

Sets the test case string of methods.

TestCase::setType(type) : Void

Sets the test case type.

TestCase::setVersion(version) : Void

Sets the test case version.

TestCase::TestCase(name, type, testCase, version) : Void

Creates a test case with the specified name, type, methods “TestCase” and version.

TestCaseClassification::getClassification() : Integer

Returns the test case classification with respect to the associated regression test suite. The test case classification can be obsolete or retestable or reusable.

TestCaseClassification::setClassification(classification) : Void

Sets the test case classification with respect to the associated regression test suite. The test case classification can be set to obsolete or retestable or reusable.

TestCaseClassification::TestCaseClassification() : Void

Creates a TestCaseClassification object.

UseCase::addUseCaseMethod(useCaseMethod) : Void

Adds the passé-in method to the use case set of methods.

**UseCase::createUseCaseMethod(methodName, methodReturnType,
methodClassName) : InvokedMethod**

Creates an InvokedMethod with the specified name, return-type and class name. The method is then added to the use case set of methods.

UseCase::findUseCaseChanges(methods, allMethods) : Set of InvokedMethod

Compares the passed-in set of methods against the use case set of methods and detects the set of changed methods.

UseCase::getMgSS() : String

Returns the string that represents the sequence diagram “MgSS” associated with the use case. The MgSS is a regular expression representing the sequence diagram associated with the use case.

UseCase::getName() : String

Returns the use case name.

UseCase::setName(name) : Void

Sets the use case name.

UseCase::setMgSS(MgSS) : Void

Sets the MgSS string of methods.

UseCase::UseCase(name, MgSS) : Void

Creates a UseCase with the specified name and MgSS.

UseCaseDiagram::addUseCase(useCase) : Void

Adds the use case to the set of use cases associated with the class diagram.

UseCaseDiagram::createUseCase(name, MgSS) : UseCase

Creates a UseCase with the specified name and MgSS. The use case is then added to the use case diagram set of useCases.

UseCaseDiagram::findChangedMethods(changedUseCases, allMethods) :

Set of InvokedMethod

Compares the corresponding use cases from two use case diagrams and detects the set of changed methods.

UseCaseDiagram::findChangedUseCases(useCases) : Set of UseCase

Compares the corresponding use cases from two use case diagrams and detects the set of changed use cases.

UseCaseDiagram::getName() : String

Returns the use case diagram name.

UseCaseDiagram::getVersion() : Real

Returns the use case diagram version.

UseCaseDiagram::load(file) : Void

Creates a UseCaseDiagramParser and pass it the file name to load the use case diagram.

UseCaseDiagram::setAllMethodsAndMgSS() : Void

Creates the sets of allMethods and allMgSS's contained in the use case diagram.

UseCaseDiagram::setName(name) : Void

Sets the use case diagram name.

UseCaseDiagram::setVersion(version) : Void

Sets the use case diagram version.

UseCaseDiagram::UseCaseDiagram(name, version) : Void

Creates a UseCaseDiagram with the specified name and version.

**UseCaseDiagramChanges::associateUseCaseDiagrams(useCaseDiagram1,
useCaseDiagram2) : Void**

Associates two UseCaseDiagram's with this UseCaseDiagramChanges object.

UseCaseDiagramChanges::findUseCaseDiagram(nameVersion):UseCaseDiagram

Tries to find a UseCaseDiagram in the database with the specified name and version, if found it is returned otherwise the user is asked to specify the XMI file that contains it and it is loaded into the tool.

UseCaseDiagramChanges::findUseCaseDiagramChanges() : Void

Compares the different UseCaseDiagrams to find the sets of added, changed and deleted methods and use cases.

UseCaseDiagramChanges::showMethods(methods) : Void

Prints the passed-in set of methods.

UseCaseDiagramChanges::showParameters(parameters) : Void

Prints the passed-in set of parameters.

UseCaseDiagramChanges::showUseCases(useCases) : Void

Prints the passed-in set of use cases.

UseCaseDiagramChanges::UseCaseDiagramChanges() : Void

Creates a UseCaseDiagramChanges class to handle the comparison process between the two-associated UseCaseDiagram's.