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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

University of Ottawa- Université d'Ottawa

Traceability in Object-Oriented Quality Engineering

A Basis for Regression Analysis of Object-Oriented Software

by
Halim BEN HAJLA

A thesis submitted to the School of Graduate Studies and Research
in partial fulfillment for the requirement for the degree of

Master of Computer Science

School for Information Technology and Engineering
Department of Computer Science
Faculty of Engineering

October 1997

ABSTRACT

Object-Oriented (O-O) technology has grown over the past decades into a well defined and increasingly popular style of programming. It offers great benefits for code maintenance, software extensibility and reuse. It also introduces new concepts such as inheritance, encapsulation, polymorphism and dynamic binding. Moreover O-O software is often analyzed, designed and implemented in accordance with an iterative incremental development process. So far the current research in O-O software engineering focuses on problem analysis, software design and implementation. An important activity which has not been deeply investigated is regression analysis and testing which involves revalidating parts of a software system after it is modified. The modifications may be caused by changes in the specification or code. Such activity is commonly regarded as a major maintenance testing technique, although it can also involve the retesting of tested implementation in the system development phase. The objective of regression analysis and testing is to ensure that the modified parts of the software system still satisfy their original unmodified requirements.

In this thesis, we introduce a novel method for conducting and managing regression analysis and testing activities during an O-O iterative and incremental development process. The underlying process of the method is based on the following concepts, the Method Sequence Specification (MtSS) and the Message Sequence Specification (MgSS), the class firewall and traceability.

A *MtSS* of a class documents the causal order in which the methods can be invoked upon instantiation of the class. A *MgSS* for a method in a class documents the causal order in which messages can be sent to different instances of different classes. We use the MtSS and MgSS to detect the changes affecting the class specification and the interactions between classes. We introduce also a regression algorithm to solve the problem of reselection of test cases based on the class specification.

Traceability links together user requirements, analysis, design, implementation models and test cases. In our proposal we used it to manage the regression analysis and testing activities and to perform change analysis. We define a set of traceability links between the different artifacts produced during the O-O development process and we use them to trace and identify the models and software components affected by the changes.

The *class firewall* is computed based on the class implementation affected by changes in the code. Following Hsia [62], we define an Object Relation Graph (ORG) which captures the relationships between different classes and their objects. The relationship between classes modeled in the ORG are aggregation, inheritance and class association. Then we use the ORG to construct a class firewall which will describes the classes possibly (not necessarily) affected by changes introduced to a given class. This concept is used in our proposal to reconcile a modified class implementation against its specification described as an MtSS and MgSS.

The main contributions of this research can be summarized as follows: We provide a new and a complete definition of regression analysis and testing that takes in consideration not only the characteristics of O-O but the development process itself. We also propose a new method to reselect the test cases based on the class specification. We demonstrate that traceability is an essential framework component for O-O software development, especially in conducting and managing regression analysis and testing. We discuss also a possible integration of some of the concepts introduced in this research in an existing CASE tool.

Acknowledgments

I wish to thank the many individuals who have made this thesis possible. I am especially graceful to my supervisor Dr. R.L Probert who helped me choose the thesis topic and provided guidance and valuable support and encouragement during this research. I would like also to thank Dr. Jean-Pierre Corriveau my co-supervisor for his initial suggestion for the traceability as a promising research area. He also provided me with valuable information on the object-oriented technology and reviewed my work. I also would like to express my gratitude to Mr. Bran Selic and the *ObjecTime Ltd.* staff for their help and the valuable information about the ROOM methodology and the Case Tool ObjecTime.

The name TOOQE and the corresponding Canadian hat logo were conceived at the university of Ottawa by Halim Ben Hajla, Dr. Robert Probert and in collaboration with Dr. J-P Corriveau.

Many individual helped in the review of this document, but in particular, I wish to thank my friend and colleague Alain Williams for his thorough reviews and perceptive comments. Finally and most importantly I wish to thank my parents, my brothers and sisters especially my sister Saida for her support and encouragement during the whole period that went into conducting this research and writing the thesis.

To my mother Slaha and my sister Saida

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

1.1. Introduction and Definitions

There is an increasing demand for innovative software that satisfies high quality and reliability requirements imposed by users. In recent years, the Object-Oriented paradigm is gaining acceptance for developing robust and complex software. Powerful features such as encapsulation, inheritance and dynamic binding introduce new problems into the area of software validation and verification as recognized in [6,14,62]. To date, much of the effort spent in research on O-O software has been in problem analysis, software design and implementation. An important activity which has not been deeply studied is O-O testing. In particular *regression analysis and testing* which involves retesting parts of software system after it has been modified. The modification may be caused by specification or code change. Such activity is commonly thought of as a major maintenance technique, although it can also involve retesting an implementation in the system development phase.

According to [64] regression analysis and testing begins in the development phase after the detection and correction of errors in a *tested program*. A tested program is a program which has been tested with a *high quality test plan*, which means that at the last stages of the program development and after it has been reasonably tested, a well developed test plan should be available. It makes sense to reuse existing test cases rather than to create all new cases, in retesting the program after it is corrected. In the maintenance phase the software system may be corrected, adapted to an new environment or enhanced to improve its performance. Therefore we note that regression analysis and testing is required in both the development phase (before the release of a new system) and the maintenance phase (after release or revision of the system); where either requirements or design details are changed or when faults are diagnosed and repaired. In some cases, both types of changes are performed.

Modifying a program involves creating new logic to fix an error or to implement a change and incorporate that logic into an existing program. The new logic may involve minor modifications such as adding, deleting or modifying few lines of code, or major modifications such as deleting adding or modifying modules[1] or subsystems. In both cases, the specification of the program may or may not be changed.

The main objectives of regression analysis and testing are to insure that the modified parts of the software system still satisfy their original unmodified requirements and that the previous functionality of the software, which should not be affected by the modifications, has indeed not been affected. That is no new errors or undesirable side effects are caused by the changes. To save effort and time. Only those parts that are affected by the modification need to be retested.

Regression analysis and testing has to address the following fundamental problems as mentioned in [62]:

[1] The term module is used to describe a logical division of the functionality of the system which contains a set of strongly coupled functions and procedures

- To identify automatically the *software components*[2] affected by a change in other software components. The changes may affect the user requirements; analysis, design, or implementation models, and test cases.
- To determine a cost effective strategy for retesting the affected components.
- To select the appropriate coverage criteria for retesting these components.
- To select, reuse or modify the existing test suites (and generate new a one) in order to support an economical (cost-effective) revalidation process.

Solutions to these problems for traditional systems have been proposed during the last two decades [7]. However, as mentioned earlier testing, of object-oriented programs have received little attention, and regression analysis and testing have received almost none. In this research we propose a novel method for conducting and managing O-O regression analysis and testing activities during the O-O software development and maintenance phases.

1.1.1. Types of modifications and regression analysis and testing:

Many modifications may occur during the development and maintenance phases where the software system is corrected, updated, or fine-tuned. In [64], White classifies the modifications in three types according to the following maintenance activities performed on the software system:

- *Corrective maintenance:* Commonly called 'fixes', and which involves correcting the software failures in order to keep software working properly. The specification is not likely to be changed and no new modules are likely to be introduced. Many program modifications occurring during the development phase are similar to corrective maintenance because the specification usually will not be changed based on the identified code errors.
- *Adaptive maintenance:* Involves adapting the system in response to new user requirements or processing environments.
- *Perfective maintenance:* Covers any enhancement to the system, where the objective may be to provide additional functionality, increased processing efficiency, or improved maintainability.

During adaptive and perfective maintenance, new functions and modules are usually introduced to reflect a change in the user requirements and/or the specification of system. We combine adaptive and perfective maintenance into *progressive maintenance*, and only two types of regression analysis and testing need to be considered:

- *Progressive regression analysis and testing:* The specification or a set of user requirements is modified. Whenever new enhancements, new data or functions are incorporated in the system, the specification is modified to reflect these changes. In most cases, new modules or subsystems will be added to the system with the consequence that the regression analysis and testing process involves testing a modified design or implementation against a modified specification.
- *Corrective regression analysis and testing:* The specification does not change. Only some instructions of the program and possibly some design decisions are modified. This has important implications because most of the test cases in the previous plan are likely to be valid. The corrective regression analysis and testing is often performed after the corrective maintenance is performed.

[2] The definition of software components is provided in Chapter 2 , section 2.2.3.

1.2. Statement of the research problem and motivation

O-O regression analysis and testing differs from traditional approaches due to the fact that many of the O-O development methodologies involve the use of an Iterative Incremental Development Process (IIDP) in order to build and deliver reusable, extensible and robust software [18,50]. In such a process the different artifacts (analysis, design, implementation models) are developed through successive refinements, increasing the level of detail over a number of iterations. System functionality is added, modified or deleted based on an increasing understanding of the problem domain and the interactions between users and the development team. The benefits of such a development strategy are numerous as described in [50]. New problems and challenges are introduced for O-O regression analysis and testing activity, namely how to handle the modifications and changes during the development process itself. Moreover, the O-O paradigm for software development introduces a number of new concepts, such as a class inheritance, encapsulation, dynamic binding and polymorphism. These concepts result in complex relations between classes and their attributes. They not only introduce new testing problems, but raise new challenging questions about how to conduct and manage regression analysis and testing for O-O software.

The main motivating factors behind this research can be summarized as follows (a detailed description is provided in Chapter 2, section 2.4.4):

- ***The absence of methods to address the O-O regression analysis and testing problem:*** Few of the existing regression analysis and testing methodologies address O-O software development. They are code based and designed for procedural languages [62]. Although many of the existing results can be applied to regression analysis and testing of methods, or member functions of a class at the unit and integration level, they are not suitable for components at higher levels, such as a class, a group of classes, or class libraries.

- ***The maintenance problem of test Data :*** Regression analysis and testing is concerned with the maintenance of test data. Many of the existing code-based methods focus on the changes introduced to the code and their effects on the relevance of test cases. However, new research results have established that focussing only on the code is inadequate [46,51]; modifications should also involve the other life cycle work products such as analysis and design models (e.g., message sequence charts, use cases, finite state machines, etc...). The traceability between the analysis and design models, and the implementation is recognized as a key life cycle issue

- ***Reusability and class evolution :*** There are few strategies for retesting reusable classes based on their specifications [47]. Such methods would facilitate the retesting process by selecting the appropriate regression test cases from existing test suites and therefore reinforce the concept of reusability of both test cases and classes.

- ***The O-O test case perspective:*** There are three common methods used to generate test cases respectively known as, black-box, white-box, and grey-box test design techniques [65]. In the first method test, cases are generated based on the specification of the system. The second method is to apply structural testing which involves test cases based on control and data flow structures of a program. The third method uses scenarios and use cases developed by the user to generate test cases. In an O-O system, classes encapsulate data and functions. At run time, each instance object has a state, hence the need of state based testing strategy [6]. More research is needed to determine how state-based test cases can be included in the regression analysis and testing methods.

1.3. Scope of the thesis and general contributions

In this research, we describe a novel methodology for conducting and managing the regression analysis and testing activities of O-O software during the IIDP, called the TOOQE methodology (Traceability in Object Oriented Quality Engineering). The method can be classified as a progressive regression analysis and testing method. All the artifacts produced during the development process are considered when the regression analysis is performed. The underlying process of the method is based on three concepts:

- *Traceability:* The user requirements; analysis, design, and implementation models; and test cases are linked together [52]. This is used as a framework for managing the regression analysis and testing activities and performing change impact analysis. We define a set of traceability links between the different artifacts produced during the O-O development process and use them to trace and identify the software components (models artifacts and code) affected by the changes.

- *The regular expression formalism:* We model the class specification behavior and the interactions between classes using the regular expression formalism [2,39]. We introduce two concepts the Method Sequence Specification (MtSS) and the Message Sequence Specification (MgSS) [1]. The MtSS and MgSS abstract the behavior of classes. These concepts are used to detect the changes affecting the class specification of a class and the interactions between classes. We also introduce a regression algorithm to solve the reselection problem of test cases based on the class specification , MtSS and MgSS.

- *The Class firewall:* The relationship between classes such as aggregation, inheritance and class association [18] are modeled in a graph used to construct a class firewall [62] which describes the classes possibly affected by the code changes. We use this class firewall concept in our proposal to reconcile a modified class implementation and its specification modeled as MtSS and MgSS.

The contributions of this thesis can be summarized as follows:

- We provide a new and a complete definition of regression testing that takes in consideration not only the characteristics of O-O software, but also the development process itself. It fits well with the O-O paradigm, where the primary focus in the different O-O development methodologies is the development process, with its analysis, design and implementation models.

- We promote the use of the iterative incremental development process in order to build reusable, extendible and robust software. We show how regression analysis and testing can be performed at the end of each iteration of software development. Such an approach allows developers to begin testing activities at the early stages of development and therefore reduces the cost of future maintenance activities by improving the quality of software.

- We have developed, and now propose, a practical and concrete solution to the O-O regression analysis and testing problem. Depending on the formalism used to express a view of the system at a certain development stage, we provide techniques for abstracting the class specification and the interaction between classes. The method is flexible and can be easily integrated into many of the existing, CASE supported O-O methodologies to reinforce the validation and verification activities.

- We demonstrate that traceability is an essential framework component for O-O software development, especially in conducting and managing O-O regression analysis and testing. We define a set of traceability links between the analysis and design models that can be used to perform change analysis. The links are not simple cross references between the

models, but carry semantics that allow the developers to control the evolution of the software, detect the changes and trace their ripple effects.

- We propose a new method to reselect the test cases to be used in the regression analysis and testing activities. Unlike previous strategies, our method does not focus solely on the code, but takes into consideration the changes introduced to the class specification.

- We discuss how to integrate some of the TOOQE concepts in a commercial CASE tool called [3]ObjecTime™. The tool provides users with an integrated development environment, including analysis, design, and code generation capabilities. We propose to extend the validation and verification capabilities by including traceability and regression analysis, and testing features.

1.4. Outline of the thesis

The thesis is organized as follows:

- Chapter 2: Describes background information on O-O software, and the basic terminology used throughout this document. We define the regression analysis and testing problem for O-O software, review the relevant literature and give detailed descriptions of the motivations behind this research. The chapter concludes with an overview of various methods and the contribution of this research.

- Chapter 3: Describes a method for capturing the design and characterizing the changes introduced to the analysis and design models, based on the concept of method and message sequence specification.

- Chapter 4: In this chapter, a list of the possible changes that may affect classes during an IIDP is presented. The regression analysis problem is discussed and a set of rules is defined in order to detect changes.

- Chapter 5: Discusses the problem of validating the changes introduced to the implementation. A reconciliation method between the classes affected by the code changes and their specification is described.

[3] ObjecTime is a commercial real time O-O CASE tool, and is a registered trade mark of ObjecTime Ltd.

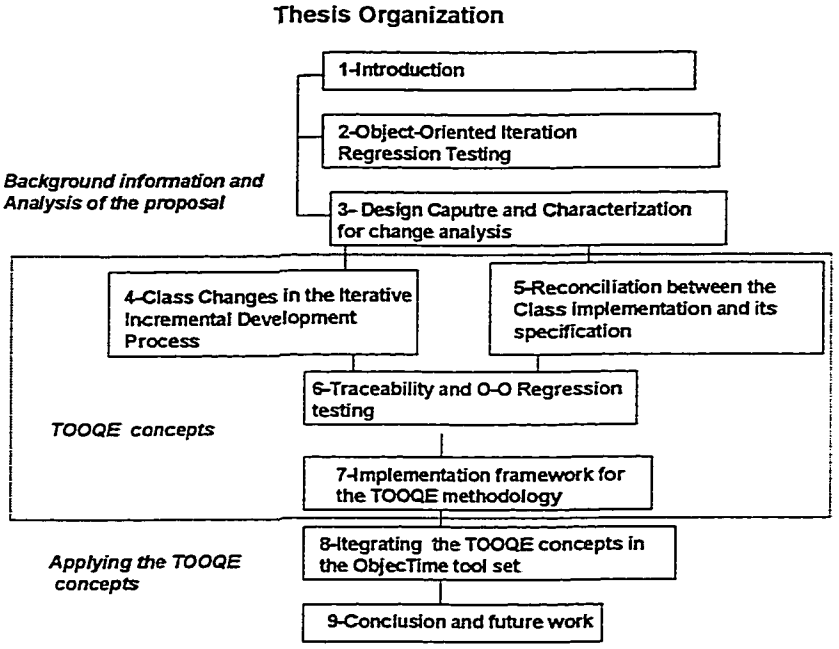


Figure 1-1: Outlines of the thesis

- Chapter 6: We stress the importance of traceability as a framework that can be used to integrate all the regression analysis and testing activities of O-O software development.
- Chapter 7: A high-level supporting tool is presented. The implementation issues are discussed.
- Chapter 8: A possible integration of the TOOQE concepts with an existing commercial, real-time CASE tool called ObjecTime is described.
- Chapter 9: Concludes the work and proposes future research topics in the area of O-O regression analysis and testing.

2. Object Oriented iteration regression analysis

2.1. Introduction

Regression analysis and testing involve retesting parts of a software system after they are modified. The modifications may be caused by changes in the specification or in the code. The objective of regression analysis and testing is to ensure that the modified program still satisfies its original unmodified requirements. To save time and effort, regression analysis and testing needs only to retest those parts affected by the modifications.

The O-O paradigm for software development introduces a number of new concepts such as *inheritance, encapsulation, polymorphism and dynamic binding.* Moreover, O-O software is most often analyzed, designed, implemented and tested in accordance with an *iterative* and *incremental* development life cycle. These new concepts result in complex relationships between classes, and their attributes. They not only introduce new testing problems as recognized in [62], but also raise new problems of managing and conducting regression analysis and testing.

In the following sections, we define what is an iterative and incremental development process from our perspective. Then, we present background information related to O-O software testing. We define and analyze the regression analysis and testing problem in general, then present an extensive survey of the existing literature in regression analysis and testing. Next, we point out some of the limitations, and conclude with an overview of the novel method to conduct and manage regression analysis and testing for O-O software design development.

2.2. The Iterative Incremental Development Process (IIDP)

The IIDP is a framework for laying out activities for different time periods in the analysis, design, implementation and test development phases. It is an approach that manages complexity and allows for changes during software development. It is a framework for methodology steps used in developing and documenting the software. Iterative development refers to the fact that a software system is repeatedly improved over a number of iterations, based on an increasing understanding of the functionality of the system to be implemented, and the user requirements. Incremental development means that during the migration from one iteration to another, new functionality is added to the evolving system and integrated with existing subsystems (see figure 2-1).

The goals of the IIDP are:

- To ensure the flexibility to build the right system by systematically verifying and refining requirements with customers at planned intervals.

- To allow detailed plans to build complex systems that more closely match user requirements by staging a roll out of the functions as more of the system is understood.

O-O software is often developed and implemented using an IIDP [11,18,50]. The analysis, design and implementation models are developed and produced through successive refinement. The system capabilities are added, deleted and modified based on continuously increasing understanding of the user requirements. The development and release of the system takes place over a certain number of iterations, depending on the type of the project[4], and the available development resources[5]. The completion of one iteration and the entire development process itself is based on the satisfaction of the user requirements. At the end of each iteration, a version of the system is released. The development team, in collaboration with users, assess the version and certify that the implemented system functionality satisfies the original requirements.

2.2.1. An iteration

2.2.1.1. Definition

According to [50], a single iteration consists of planning, production, and assessment periods. The iterations discussed in our case are major iterations of a system or subsystem. There are numerous builds in iterations that occur during each major iteration. These build cycle iterations are discussed in more detail in the following sections.

During one iteration, functions are added incrementally to the evolving system while we iterate on complex and ill-defined portions of the system a number of times. This means that the same portion of the system is worked on a number of times based on increased understanding of the tasks and related requirements (see Figure 2-1):

- Planning : It starts an iteration. It is used to adapt to changing requirements, resources, schedules and prerelease "drivers" code contents. The activities consists of:

1. Updating and prioritizing the system requirements.

2. Documenting external dependencies and deliverables.

3. Determining specific goals for each iteration.

4. Establishing a schedule.

- Production: In this period the analysis, design and implementation of the iteration line items occur following the O-O methodology chosen by the development team. During the design and test phase, the test team will be making builds to provide quick feedback to the developers and to iterate execution of system test cases.

- Assessment : It completes one iteration. It is used to evaluate the system according to a variety of criteria:

1.Conformance to customer requirements

2.Usability

3.Competitiveness

4.Performance

[4] Type of the project refer to the problem domain (e.g., real time distributed application), the complexity and the scale of the project (e.g.,small, medium, large).

[5] Available resources refer to the number of developers involved in the project, the availability of CASE tools, testing tools, etc.

5.Extensibility

6. Reusability

7. Reliability

8. Backward compatibility

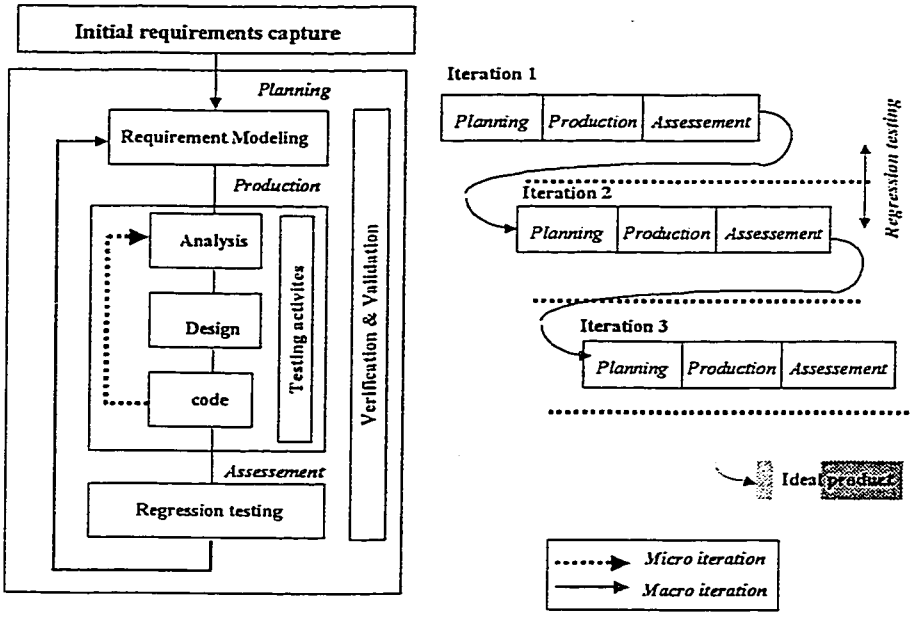


Figure 2-1 The Iterative Incremental Development Process

According to [48], there are two types of iterations: *micro* and *macro* iterations, the latter consisting of several micro iterations. After the user requirements are captured in the problem domain (e.g., by use of cases or scenarios), the planning period starts by modeling the requirements. Such modeling may be performed on the whole set of requirements or only on a predetermined subset, depending on the type of the project and the goals of the iterations. This marks the beginning of a macro iteration. In practice a new macro iteration addresses a new subset of the requirements of the system. Micro iterations iterate over these requirements selected during the macro iteration until the goals of the macro iteration are satisfied (see Figure 2-1).

2.2.1.2.Number of iterations

The number of macro iterations depends on the application being developed. Factors affecting the number of iterations include:

- The size of the project.
- The stability of the requirement.
- The complexity of the system.
- The productivity of the people and environment.

2.2.1.3.The duration of iterations

The duration of one iteration depends on many factors such as:

1. The logical "parts" of the system and functions to be delivered.
2. The availability of the customers.
3. The volatility of the requirements.
4. The skills of the development team.

2.2.2. The management of changes during the IIDP

The IIDP is a flexible process where changes can be introduced to all the software artifacts, namely analysis, design, and implementation models. It creates new and challenging problems to the software testing community:

- How to identify changes;
- How to assess the *ripple effects*[5] of such changes on the interrelated models. For example, changing an analysis model may require a review of all the design models derived from it (e.g., cross-checking use cases and interaction diagrams [11]); and
- How to systematically conduct validation and verification activities during micro and macro iterations.

In the following, we give a brief description of common types of changes that can be performed during the development process:

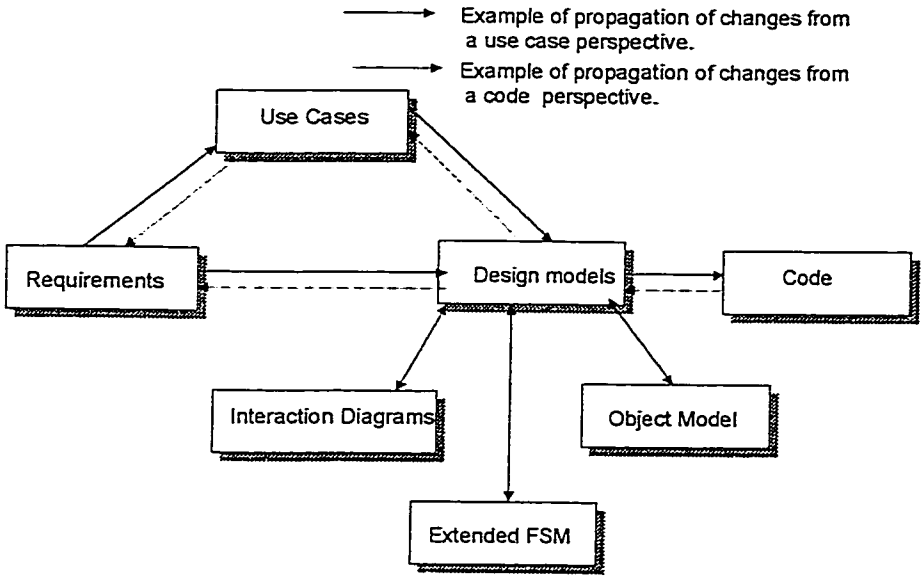


Figure 2-2: Propagation of changes in the IIDP

- *Changes introduced from a use case perspective:* This is a normal paradigm for software development. When a macro iteration begins, the development team extends the current software requirements to a new subset of requirements or modifies them to accommodate a user request or an implementation design constraint. The modification can be introduced to

[5] The term "ripple effects" in our context refers to the impact of changing one model on other related models, and the propagation of such modifications.

both the functional and non-functional requirements. They will be propagated to the already existing models including the code, and are identified as *ripple effects*. A simple example of such a change is adding new use cases. Such modification will affect, of course, the existing interaction diagrams. It may also lead to the creation of new classes, and establishment of new interactions between classes.

- *Changes introduced directly to the code:* Frequently, during the migration from one iteration to another, the development team may directly modify the code, without updating the corresponding analysis and design models. Such changes often lead to inconsistencies between the software documentation and the code. Providing the development team with a method which allows them to trace such changes will help in building high quality software.

Changes introduced to the analysis and design models directly affect the class definitions because class definitions form the basic components of building O-O software. In [23], Kung and Hsia identify the following types of changes that may be introduced to a class at the specification or the design or implementation levels :

1. Modification of the internal structure of class (e.g., adding/deleting methods).
2. Modification of the class hierarchy (e.g., add/delete a superclass, subclass).
3. Modification of the class interactions (e.g., add/delete associations).

In [14] and [48], the authors propose new solutions to address such problems by integrating the V & V activities with the IIDP itself, and proposing methodologies to conduct testing activities not only on the implementation and code but also on the analysis and design models. In this research we focus on the validation and verification activities that should be performed during the migration from one iteration to another; and identify these as *regression analysis and testing* activities. In section 2.4 we will define regression analysis and testing from our perspective, explain why it is needed in the IIDP, and provide a method for conducting and managing such activities. We assume that the IIDP will be the development methodology for our analysis of the iteration regression analysis and testing problem.

2.2.3. The IIDP and Software Components

During the analysis and design phases, *objects* (or *classes*) are identified. For example, for a medium size project, 30 to 100 objects can be specified [11]. It is seldom possible to get a clear overview of the number of objects, so that objects need to be placed in *groups*. This can be done at one or several levels, depending on the size of the project. Such groups of objects are called subsystems. The system thus consists of a number of subsystems which themselves contain subsystems. At the bottom of such a hierarchy are the objects (classes). The division into subsystems is usually based on the functionality of the system. All objects which have a strong mutual functional coupling [11] will be placed in the same subsystem. Another criterion for the division is that there should be as little communication as possible between subsystems.

The task of a subsystem is to package objects so that complexity is reduced. They work also as functional units in an organization for example marketing, sales, and delivery. Subsystems can be reused in similar projects and integrated with existing subsystems without performing major changes in their internal structure. They provide services to each others via contracts, where a contract defines the interface or the set of messages that the subsystem can send or receive [18] (Reusability of software components).

At the lowest level, subsystems are viewed as change units. We call these units *software components* [67]. These should be viewed as atomic. If the user so desires, he/she will get all of the software components. During the migration from one iteration to another, changes are introduced to one component or rather to the objects (classes) contained in it. This means that another important criterion to divide the system into subsystems or "software components" is the potential set of users of the components, since changes are usually caused or requested by users.

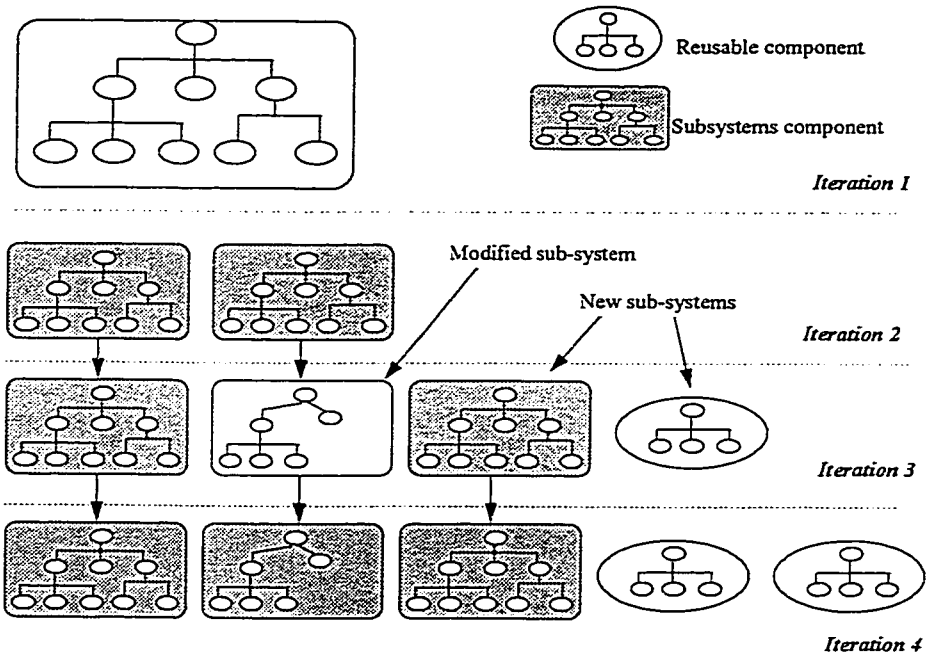


Figure 2-3: The IIDP and software components design

2.3. Testing Object-Oriented Software:

Testing is an important component of the Object-Oriented development process. In [14] the authors discuss three major categories of testing activities:

2.3.1. Testing the development process and the supporting documents:

There is a need to evaluate and to improve the development processes as much as the products. The process consists of a set of activities supported by a set of documents that describe the activities. The IIDP approach supports continuous improvement through feedback. Specifically *process testing* examines the goals of each iteration and evaluates the achievement of these goals at the end of the iteration. It also examines a cross section of successive iterations to identify problems and improvements in a particular activity across the iterations. Process and configuration management tools [16] are designed to address such issues.

2.3.2. Testing the analysis and design models:

The Object-Oriented approach places significant importance on the analysis and design models and the identification of abstractions. Since these directly shape the code and provide key customer and domain information, testing the analysis and design models is important. Regardless of the methodology adopted to perform such activity, models are usually checked for correctness, consistency, and completeness. Formal methods promote formal systematic testing (e.g., LOTOS, SDL).

2.3.3. Static and dynamic testing of the implementation

Testing the implementation has been addressed under a variety of aspects such as class based [9] and state based testing [6]. The goal of this type of activity is to locate as many errors as possible in the code. It includes the generation of test cases based on the specifications or on

the code itself. Static testing includes techniques like code inspection however dynamic testing focus on finding bugs in the implementation at runtime.

We mentioned earlier that the O-O paradigm introduces new concepts such as *inheritance, explicit encapsulation, polymorphism* and *dynamic binding*. These new concepts result in complex relationship between classes, and their attributes; and introduce new testing problems. Objects encapsulate data and functions; thus at run time they have different states. In [6], authors show the need for state based methodologies in O-O software and provide a method to generate such test cases based on the class specification. They also study the implication of inheritance on generating and reusing the test cases in a class hierarchy. In [10], authors focus on solving the testing problems introduced by polymorphic messages and dynamic binding. They propose a method to generate a minimum number of test cases by reducing the number of all the possible combinations of polymorphic messages. although the problem is still under investigation.

Testing of the implementation is organized around the recognized units of complexity of O-O design. Four level of complexity for testing are discussed here [14]:

1. Class testing: A class is basic unit for building O-O software. It has its own instance variables and methods. The external behavior (black-box) and the internal structure (grey-box, white-box) of the class should be tested.

2. Cluster testing: A cluster of classes is a grouping of cooperating classes. The focus of cluster testing is interaction among the instances of the classes in the cluster. It is assumed that each class has been tested individually. This means that the internal structural and the external behavior is tested and certain coverage criteria are satisfied. To aid in recognizing a cluster, a specification should be developed for each cluster that is similar to the specification for a class. This will include specifying those methods from each class that will be accessed from outside the cluster and the pre-conditions and the post-conditions for each one (cluster contracts).

3. Subsystem testing: A subsystem is a grouping of clusters. During this activity interaction between clusters are tested based on the use cases and the cluster contracts defined previously.

4. System testing: The system functionality is tested using test cases derived from the use cases and other system requirements.

2.3.4. Test design perspective

In the following section we provide an overview of the test case design perspective in O-O software.

2.3.4.1.Functional test design (Black-Box)

The functional perspective is the external view that considers the behavior promised by the class to those interacting with it. Functional test cases are constructed based on the specification of the component. The specification of a class includes a specification for each method plus a class invariant. Typically, every method is executed at least once to verify conformance to stated pre-conditions and post-conditions.

2.3.4.2. Structural test design (White-Box)

The structural view is an internal view that is guided by the relationships among individual lines of code. Structural test cases are constructed by identifying individual paths through the code. The coverage will be usually stated in terms of percentage of the *branches* of code that have been executed based on the test cases. Every line of code is typically executed at least once. However, it is not possible in general to execute all the paths through the code[64]. A

larger percentage of the paths through the individual methods can be executed, rather than paths through the complete program [14].

Functional testing should receive more emphasis than structural testing in the O-O development process. There are several reasons for this:

8. Methods in classes are typically sufficiently small that an aggressive code inspection can catch many errors at early stage of the development process.

9. The information hiding property of objects makes the internals of individual objects less likely to cause problems for other objects.

10. The presence of polymorphic messages that are not bound until execution time (dynamic binding), makes static control and data flow analysis difficult.

2.3.4.3. Interactions test design (Grey-Box)

There are two levels of interactions that are of interest to us: interactions among methods within a class, and method interactions among different classes. Even at the simplest level, an interaction between methods may directly or indirectly produce incorrect results. For example a Stack object should be tested if its state is empty or not before sending any 'pop' message to retrieve an element. Interaction test cases are constructed by identifying values that are set or used by several methods, including cases of parameters being passed between two methods. The two methods may both be in the same class (intraclass) or in different classes (interclass). This means that not only the state of each object included in the test case should be clearly identified and properly set, but also that the state of the object parameters passed between the methods including the expected objects result should be known.

O-O technology complicates interaction testing by using *dynamic binding.* Certain interclass relationships cannot be uniquely defined at compile time, and thus a generated interaction cannot be identified. Strongly typed languages do restrict the associations that can be formed at run time, and thus make the identification of a set of potential associations difficult. Authors in [10] analyze the problem of completely covering all pairwise interactions between objects. They use an orthogonal Latin squares algorithm to provide a technique for constructing appropriate set of test cases.

2.4. Object-Oriented Regression analysis and testing

2.4.1. Definitions of specification oriented and code based regression analysis and testing

Regression analysis and testing is a process applied after a change to the implementation of a software system is made [13]. The specification of the system may or may not be changed. It is commonly thought of as a major software maintenance activity, although it can also involve retesting a previously tested program during the system development phase. The objective of regression analysis and testing is to ensure that the modified program still satisfies its unmodified requirements, and to ensure that the previous functionality of the software which should not been affected by the modifications has not, in fact, been affected (i.e., no new errors are introduced and no unintended side effects are caused by the change).

We have mentioned earlier that many of the existing regression analysis and testing methods focus only on the changes introduced to the code. In the following we provide a new and complete definition of regression analysis and testing from our perspective. We take under consideration not only the characteristics of the O-O software, but also the nature of the development process itself (IIDP). Regression analysis and testing is defined from two

perspectives. The first view is related to the code. It is inspired from the regression analysis and testing methods designed for C++ program and described in [12]. The second view is related to the O-O analysis and design models generated during the development process. It is important to point out that the two definitions complement each other.

1. Code oriented definition:

Given a program P, a modified version P', and a set T used previously to test P, regression analysis and testing techniques attempt to make use of T to gain sufficient confidence in the correctness of P'. These techniques consists of the following steps:

1. Identify the modifications that were made to P.
2. Select $T' \subseteq T$, the set of tests to re-execute P'.
3. Retest P' with T' establishing P' correctness with respect to T'.
4. If necessary to satisfy some coverage criteria, create new tests for P'.
5. Create T'', a new test set for P, and gather test information for T''.

6. Specification oriented definition:

Given a set of analysis and design models $M=\{M_1, M_2,..,M_k\}$ and a set of use (test) cases T generated based on these models (functional and state based test cases) to test a program P.

Specification regression analysis and testing attempts to reselect a subset $T' \subseteq T$ of test cases that are generated from the models affected by the modification, and that can be used to re-validate the modified version of the program P.

Regular expressions are used to represent the set of paths that can be executed by the test cases in the models before and after modification. By comparing these expressions using elementary algebraic operations, it is possible to classify the paths affected by the modification into new, deleted and modified paths. The technique consists of two steps:

1. Identify those paths that have been added, deleted or modified.
2. Update and rerun the test cases which exercise the modified and new paths.

Examples of models: Message Sequence Charts (MSCs), State Transition Diagrams (STDs), ROOM state charts.

2.4.2. Regression analysis and testing strategies (Discussion)

It has been realized [34] that completely retesting the whole software system which has few changes is very expensive in terms of time and computational resources. Hence regression analysis and testing needs to retest only the parts affected by the modification. As we mentioned in Chapter 1, regression analysis and testing has to address the following fundamental questions:

1. What are the affected components due to change of some software components? The changes may affect the user requirements; analysis, design, implementation models, and test cases.
2. What strategy should be used to retest the affected components?
3. What are the coverage criteria for retesting these components?
4. How should one select, reuse and modify the existing test suites (and generate new ones) in order to support an economical (cost-effective) revalidation process? Such activity is known as *selective regression analysis and testing strategy* [13].

To address the above issues, any regression analysis and testing strategy should involve five essential activities [62](see Figure 2-4):

5. Identify the affected components and characterize the ripple effects of the modification.

6. Select the test cases to retest the affected implementation component(s).

7. Execute the modified program based on the selected test cases.

8. Ensure that the modified program still performs the intended behavior specified in the (possibly modified) specification.

9. Update the test plan for the next regression analysis and testing session.

The first activity in regression analysis and testing is fundamental; it should be performed so the tester understands what changes have been made, and analyzes the impact of changes to recognize the regions affected by the changes. Most previous proposals [13] in supporting this activity focus on studying the impact of changes within the code. However, the global effects of change, particularly when the specification of the system is changed, would apply to not only the program but also to the specifications and test cases. Therefore we propose that after the affected components (in the specifications or the program) are determined, the test cases associated with these affected regions should be identified as affected by changes.

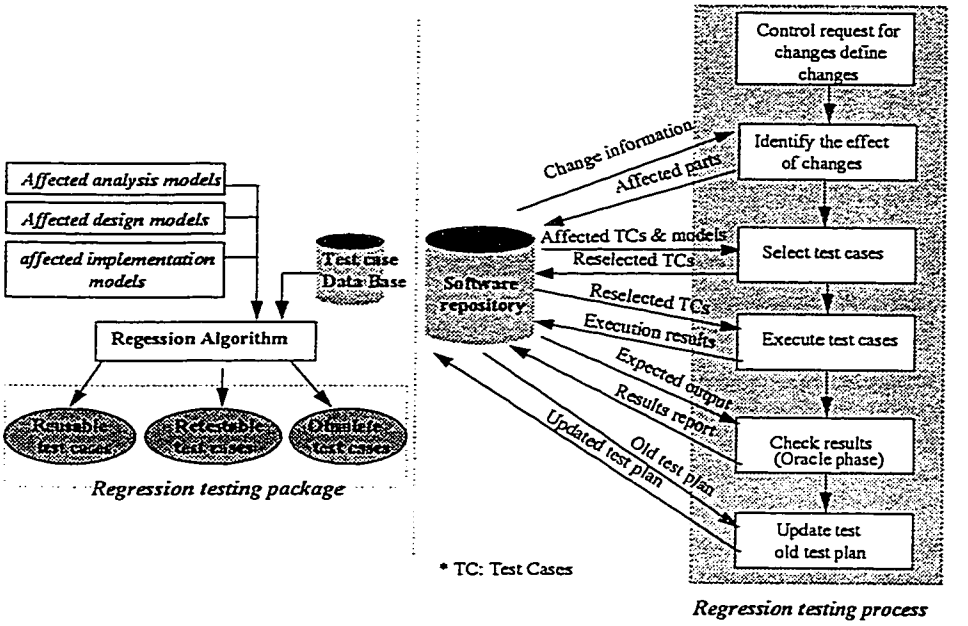


Figure 2-4: Test cases reselection process

Activity (2) is concerned with the selection of test cases to run or rerun. It involves the generation of new test cases and reselection of appropriate old ones (see Figure 2-4 and Figure 2-5) . In [13], the old test cases are divided into two classes:

10. *Reusable test cases* are those which tested unmodified parts of the specification and their corresponding unmodified parts of the implementation. They remain valid after the modification to the specification and implementation, and need not be rerun.

11. *Affected test cases:* The test cases relevant to the modified parts of the specification and the implementation have two categories: *retestable test cases* which are still valid and should be rerun; and *obsolete test cases* which have become irrelevant or out of date due to the changed specification or programs.

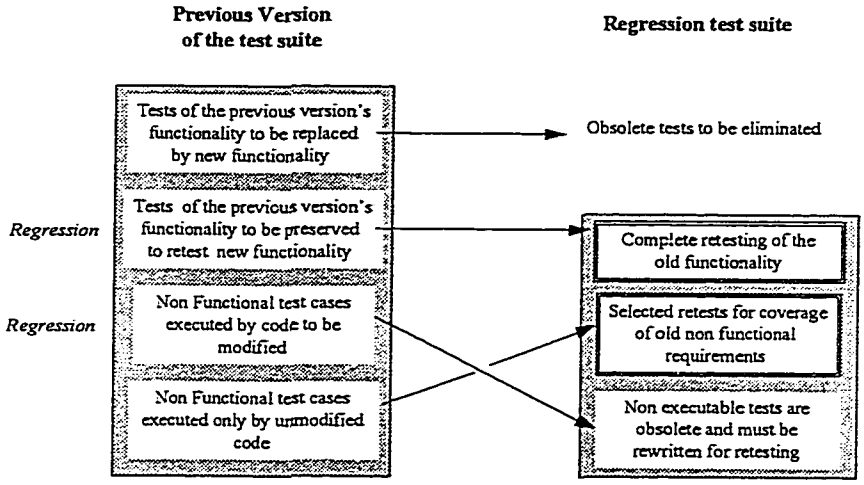


Figure 2-5: Relation between Regression tests and the previous test cases

2.4.3. Related Work in regression analysis and testing

In the following section we briefly review existing work on regression analysis and testing, discuss problems and relate our work to the existing work.

Lui and Roberson [13] proposed a model of a regression analysis and testing database (RTD) which emphasizes configuration management, traceability and change impact analysis of data used in regression analysis and testing. Then they describe the SEMST, a prototype system that manages all versions of specifications, test cases, and programs; as well as control relationships between these components.

Hartman and Robson, examine several regression analysis and testing strategies, including methods for capturing the program portions which may be affected by maintenance modifications to a conventional program [27]. A similar study was conducted by Leung and White [28] using a formally defined cost model. They use data flow analysis and program understanding models. Laski and Szermer [29] describe an algorithm for identifying the affected parts (which we call ripple effects) in conventional program maintenance. The algorithm is based on differentials between the control flow graphs [12] of the original and the modified program.

Some conventional program maintenance systems have been reported in the literature. VIFOR (Visual Interactive FORtran) [31] were developed for FORTRAN programs, MasterScope [32] for Interlisp, and CIA (C Information Abstractors) [33] for C. These systems provide editing, browsing and database supports to maintainers.

Wilde and Huit [34] analyzed problems of dynamic binding, object dependencies, dispersed program structure, control of polymorphism, high level understanding and detailed code understanding. They provide a general list of recommendations, including the use of dependency analysis [35] and clustering methods [36] [37], for possible tool support.

Croker and Mayhauser [38] addressed problems relating to class hierarchy changes, class signature changes, polymorphism, high level understanding and detailed code understanding.

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

Lejeter, Meyers and Reiss [39] discuss the difficulty of maintaining an OO software system due to the presence of inheritance and dynamic binding. They describe the XREF/XREFDB prototype system that provide text editing and relational data base querying support to facilitate OO software maintenance. A similar system was described in [16].

Kung, Hsia and Toyoshima [23] address the regression analysis and testing problem for C++ programs. They introduce the concepts of class firewall, and the block-branch diagram. Then, they describe a strategy for detecting the changes introduced to the class implementation, interaction between classes and the class hierarchy based on the above concepts. A maintenance environment called OOTME which implement their strategy is also presented.

Ernst and Newton [43] describe a test case browser called TOBAC built in the Smalltalk-80 environment to define, manage and execute test cases. The tool performs code analysis to detect changes introduced to the implementation and allows the user to built test cases interactively.

Most of the methodologies described above (except the RTD model) focus on the changes introduced to the code, how to detect them, assessing their ripple effects on the program components and how to solve the reselection of test cases problem based on the previous analysis. To do so, they use different underlying philosophies. In [7] the authors classify code based regression analysis and testing strategies in three categories:

1. *Minimization approaches*: These seek to reestablish satisfaction of some coverage criterion, by identifying a minimal set of test cases that must be rerun to meet that criterion. For example in [40] the authors describe a minimization approach that selects one test through every modified basic block, providing statement coverage of the changed code.

2. *Coverage approaches*: These seek to select all tests that exercise changed or affected program components. For example, data and control flow based algorithms select all tests that exercise changed use-pair definitions. In [41], a method is described which uses this approach.

3. *Safe approaches*: these place less emphasis on coverage criteria and attempt instead to select every test that will cause the modified program to produce different output than the original program. The retest-all strategy is a *safe* approach. A system which uses this strategy is TestTube [42].

In [7], the authors also propose a framework for assessing the reselection of test cases strategies based on five criteria:

4. *Inclusiveness*: Measures the extent to which a method chooses tests that will cause the program to produce different output than the original program.

5. *Precision:* Measures the ability of a strategy to avoid choosing tests that will not cause the program to produce different output than the original program.

6. *Efficient:* Measures the computational cost and automatability, and thus the practicality of a selective retest approach.

7. *Generality:* Measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications and realistic testing applications.

8. *Accountability:* Measures a method's support for a set of coverage criteria; that is, the extent to which the method can aid in the evaluation of test suite adequacy.

2.4.4. Limitation of the previous proposals and motivations of the research with respect to regression analysis and testing

2.4.4.1. The Lack of methods to address the O-O regression analysis and testing problem

Most of the regression analysis and testing strategies are code based and language dependent; they are designed for procedural languages. Few of these methodologies address O-O software [62]. Although many of the exiting results can be applied to the regression analysis of methods (or member functions) of a class at the unit level, they are not suitable for components at higher levels, such as a class, a group of classes, or a class libraries.

1. Traditional approaches do not address the complex relationships and dependencies, such as inheritance, aggregation, and association, that exist between classes.

2. Most traditional approaches are based on the control flow model, but classes have a state-dependent behavior that can be changed in various ways (see section 2.4.4.4). Hence, traditional approaches can not be applied to class testing.

3. Traditional approaches use test stubs to simulate the modules that are invoked. But in O-O programs this is difficult and costly because it requires understanding many related classes, their methods and how the methods are invoked [10] and executed. This is an implication of the problem of dynamic binding and polymorphic messages.

2.4.4.2. The maintenance problem of Test Data

Regression analysis and testing involves the maintenance and reuse of test data. Regression test data are of four kinds as described in [13]: analysis, design models, code or class implementation, and test cases. While code based methodologies focus on the changes introduced to the code and their effects on the relevance of test cases, research has already established that the focus on code is inadequate [46,51] and modifications should also involve the other lifecycle work products such as analysis and design models.

In an IIDP, software components are usually large, and stored in a number of different files, data formats, and storage media. There are few methods provided to record the history of these data, so testers generally have no knowledge about what changes have been made to these components and how these components have evolved in the software life cycle (*change history*). Furthermore, relationships between these components are usually poorly controlled so that the traceability over them and information about the effects of changes cannot be obtained. There are in fact, very close and complex relationships (which are of type many-to-many) existing between these components. The change to a part of one component would affect the validity of other components. Understanding the changes introduced during the development process and locating with precision what parts are involved in a modification requires the explicit representation of the links among programming concepts in the implementation domain, solution concepts in the design domain; and problem concepts in the analysis domain.

2.4.4.3. Reusability and class evolution

The O-O paradigm promotes the concept of reusability. Classes designed for reuse are selected through an iterative incremental process. Changes are introduced during the development process and in particular to accommodate a new implementation environment. While many proposals address the class evolution from a code perspective such as class tailoring, class versioning, class surgery, and class reorganization [49], there is a lack of a strategies to retest such classes based on their specifications. Such strategies would facilitate the retesting process by selecting the appropriate regression test cases from existing test suites. Retest strategies reinforce the reusability of both test cases and classes.

2.4.4.4. The test case perspective and O-O regression analysis and testing

Test case design principles are fundamental in any regression analysis and testing methods. Selective retest strategies focus on the relationship between the components affected by the changes and the test cases to decide its relevance and if it should be used to retest the program.

The reselection of test cases is an important step in any regression analysis and testing approach. In the following section we identify problems related to the relevance of test cases when changes are introduced to the analysis, design and implementation models. Since the goal of regression analysis and testing is to reselect a subset of the test cases to rerun in order to revalidate the unaffected software components, it is important to establish a framework to assess when a test case becomes obsolete, reusable and retestable in the context of O-O programs.

Traditional approaches for regression analysis and testing (designed for procedural languages) assume the following test case harness specifications or components are available [43]:

1. The initialization routine, if necessary.
2. The procedure to be tested with arguments and/or global values.
3. The expected result value.
4. The expected exception, if any.

Such specifications are simple and only defined for unit testing. The test case becomes obsolete if the path that it executes is modified or the specified relationship between the input and output becomes irrelevant (input changed/output deleted, etc.). In O-O software the distinction between unit (intraclass) and integration testing (interclass) is blurred. Testing O-O software basically means sending messages to instances of a group of interacting classes related by an association, aggregation or inheritance relationship. In this context, the following test case components are usually considered as required [43]:

5. The value of global variables before and after the test execution.
6. The class.
7. The creation method (constructor) to produce an object of that class with a certain state.
8. The message or sequence of messages to be tested.
9. The messages' arguments.
10. The expected result values.
11. The expected exception, if any, including parameters.

The following example (see Figure 2-6) illustrates the components of test cases designed for testing a set of interacting objects:

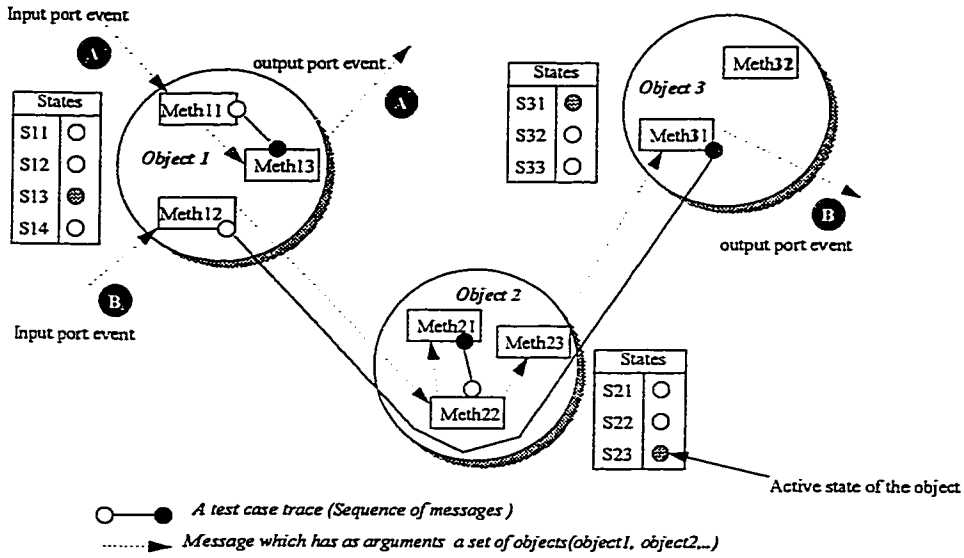


Figure 2-6: Example of the specification of a test case

Changes introduced to analysis, design and implementation models affect the following components of test cases:

12. The original state of the sender or the expected state of the receiver.
13. The objects participating in the test case trace (e.g., add new objects, delete).
14. The sequence of messages described in the test case.
15. The structure of the test case trace (e.g., add, delete messages).
16. The arguments of a method.
    - Add, delete a parameter (object); change the sequence.
    - Modify the type of a parameter (object).

Deciding when a test case becomes obsolete, reusable or retestable is a complex problem. In [12] the authors present a regression analysis and testing safe approach that uses the flow graph dependency analysis of the code to reselect test cases designed for C++ programs. The limitation of their proposal is that it just considers the execution path that the test case is supposed to validate in one class or in a class hierarchy. They do not consider the object state problem, the possible interaction between classes and polymorphic methods.

2.5. Proposal and methodology concepts

Having described the limitations and shortcomings of the previous proposals, in this section, we define and describe our methodology for conducting and managing the regression analysis and testing activities of O-O software during an IIDP. We call it the TOOQE methodology (Traceability in Object Oriented Quality Engineering). The underlying process of the method is based on the following three concepts (see Figure 2-7):

1 Traceability.

2. Regular expression formalism for method and message sequence specifications.

3. The class firewall concept.

2.5.1. Traceability

This concept links together the user requirements, analysis, design, implementation models and test cases [52]. In our proposal, we use it to conduct and manage the regression analysis and testing activities and to perform change analysis. We define a set of *traceability* links between the different artifacts produced during the O-O development process, and we use them to trace and identify the components affected by the changes. Traceability will allow the development team to navigate between the models easily, and therefore helps them to address the first problem of regression analysis and testing (see section 2.4.1). We consider TOOQE as a traceability framework which can be included in a maintenance environment and supported by a set of tools such as code analyzers, testing tools, and CASE tools.

The requirement of traceability is motivated first by the fact that one of the major difficulties in software regression analysis and testing is to identify automatically changes and their impact. It is very difficult to keep track of changes when a software system is modified extensively by several persons. This capability becomes even more crucial when the modifications are performed by one team, and regression analysis and testing is performed by another team. Traceability will help in the control, and the configuration management, of the software artifacts produced during the development process, and therefore help in solving the problem of maintenance of test data.

2.5.2. The regular expression formalism

We use the regular expression formalism as presented in [1,2] to model the class specification behavior and the interactions between classes. We introduce two new concepts the Method Sequence Specification (MtSS), and the Message Sequence Specification (MgSS) (defined in the next chapter). The MtSS of a class documents the causal order in which the methods can be invoked at the instance of the class. The MgSS of a set of classes documents the causal order in which messages can be sent to different instances of different classes.

Expressing the class specification using MtSSs and MgSSs is an important concept in our methodology. MtSSs and MgSSs can be derived from the analysis and design models, and can be used in conjunction with the traceability links to assess the ripple effects of the changes introduced to the class specification. Moreover, we use them to solve the test case reselection problem, not from a code perspective as do many of the existing regression analysis and testing methodologies [23-43], but from a use case perspective.

The motivations behind using the regular expression formalism are the following:

1. It is easy to learn and understand, MtSSs and MgSSs are simple sequences of methods. They are equivalent to finite automata [3]. Manipulating them can be automated easily based on existing algorithms for regular expressions.

2. MtSSs and MgSSs can be generated easily from the analysis and design models. The process of generation can be automated by retrieving the necessary information from CASE tools repositories.

2.5.3. The class firewall

Based on the class implementation we define an Object Relation Graph (ORG) to capture the relationships between different classes such as aggregation, inheritance and class association. We use the ORG to identify the affected classes when one or more classes are changed. The ORG concept was first introduced in [62], and it is used to construct a class firewall which describes the classes that may be affected by the changes introduced to a given class.

Many existing regression analysis and testing strategies are code based. They address the test case reselection problem by comparing the old and modified versions to detect the changes [7]. Here we go a step further; we compare a modified version of the class implementation with its specifications by comparing the MtSSs and MgSSs of a class and the use sequences[6] generated from the code. The method will be used in conjunction with the traceability links to reinforce the validation and verification process, and helps the developer to keep the software documentation up to date.

The class firewall concept is used because it is a well defined method and supporting tools already exist [62].

2.6. Overview of the methodology and assumptions

The TOOQE method consists of several steps and strategies. The concepts and terminology will be discussed in chapters 3 through 6. We will provide a brief overview at this point to clarify the method, our assumptions and what we aim to achieve. Much of our work can be summarized by the following illustration (see **Error! Reference source not found.**):

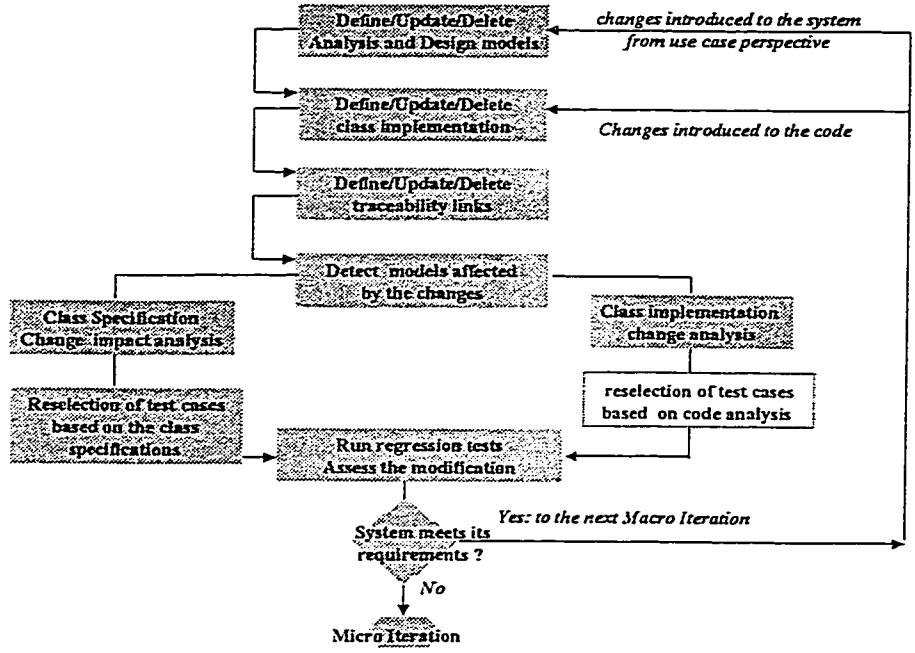


Figure 2-7: The TOOQE methodology steps

We emphasize in our regression analysis four main activities :

1. Defining and manipulating the traceability links between the analysis, design, implementation models and test cases.
2. Analyzing the impact of a class specification change.
3. Analyzing the impact of a class implementation change.
4. Reselecting test cases based on the class specification.

[6] Use sequence refers to the invocation sequence of methods sent from, or received by an object at execution time

In our framework we do not include a reselection method for test cases based on code analysis because several proposals have already solved this problem for O-O software [7,23,43,62]. They can be included easily in our framework. Also, the authors in [7] present a strategy for assessing test case reselection methods. We will use some of these concepts to assess and report the limitations and pitfalls of our testing reselection strategy.

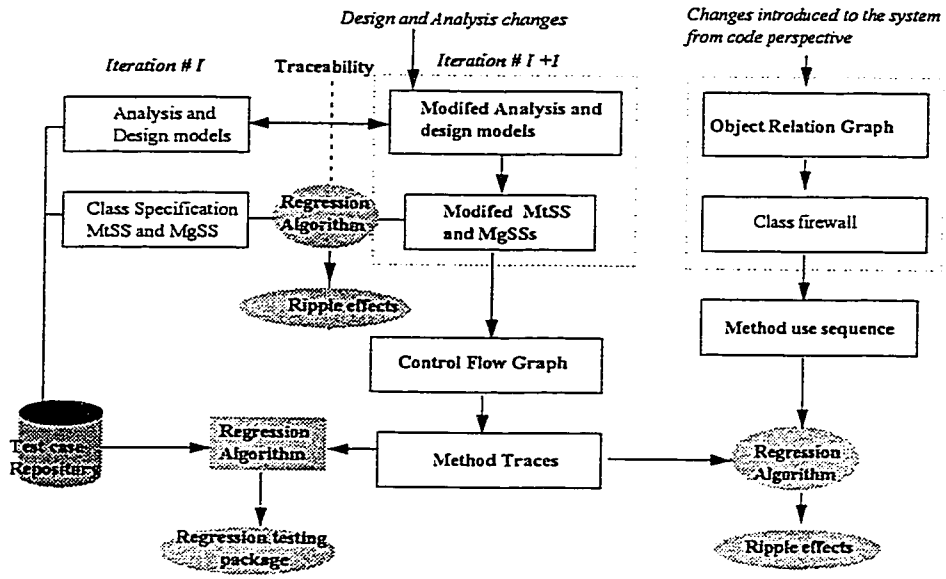


Figure 2-8: The TOOQE methodology concepts

In order to build a supporting tool for our methodology we assume the existence of :

5. **A Repository:** The different models produced during the O-O software development process are stored online in a database system. This will provide efficient management of the traceability links and the models associated with regression analysis and testing.

6. **CASE Tools:** One or more CASE tools should provide support to the development team. They will help in the process of generating and documenting the different analysis and design models and the code.

7. **Support tools:** Regression analysis and testing activities should be supported by a set of tools such as a change impact analyzer, test executor, test reporter, etc. A traceability tool that implements our methodology should be a central common area that allows these tools to communicate with each other. It should also be integrated into a maintenance environment.

2.7. Contributions of the research

The primary contribution of this thesis is to propose, for further assessment and development, a concrete and practical methodology for conducting and managing the O-O regression analysis and testing activities during an iterative and incremental software development process. We demonstrate that traceability is an important concept that can be used as a framework for detecting and assessing the changes introduced to the analysis, design and implementation models.

TOOQE is designed to help address the fundamental problems stated in the beginning of this section (see section 2.4). In our proposal we identify and conduct the regression analysis and testing activities from three perspectives (see Figure 2-9):

1. Verification and validation of the analysis and design models affected by the changes. This is done by comparing the old and new MtSSs and MgSSs of the class specification between successive iterations. The goal of this verification is to check correctness and consistency between the models, and to analyze the ripple effects of the system specification.

2. Reselection of the test cases based on the class specification. The class firewall and the traceability links will determine the potentially affected classes. For each modified class, we use the MtSSs and MgSSs to solve the reselection problem.

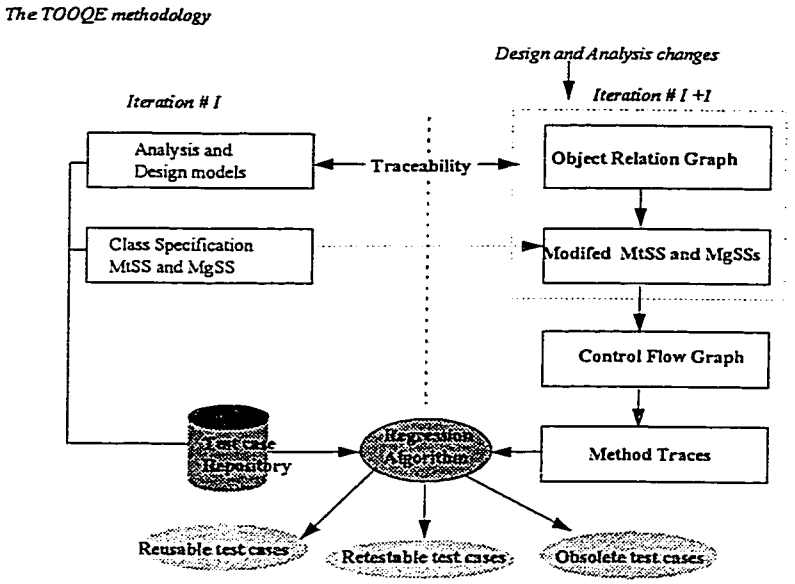


Figure 2-9: Class specification change analysis

3.Reconciliation between the class implementation affected by code changes and its specification. This is done via the mapping between the class firewall and the corresponding MtSSs and MgSSs. The motivation behind this activity is:

- checking the consistency and the correctness of the modifications against the user requirements.
- helping the development team to keep the software documentation up to date.

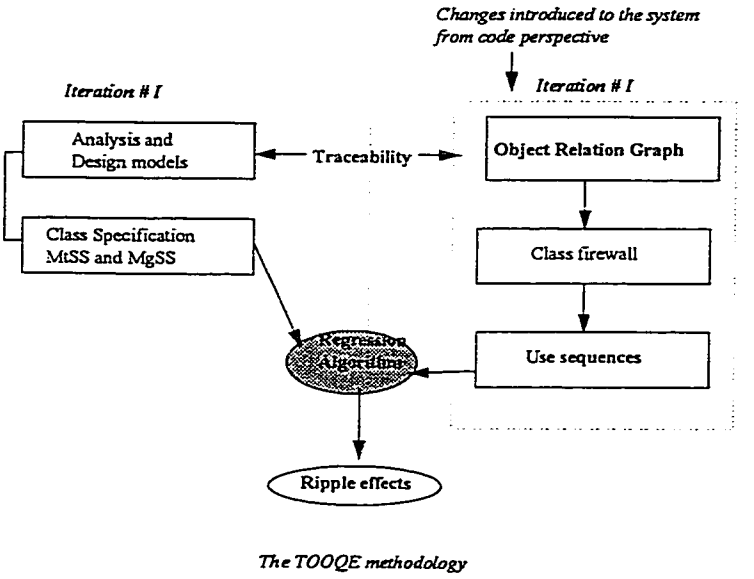


Figure 2-10: Reconciliation between the class implementation and its specification

We also conduct an extensive survey of the different existing regression analysis and testing methodologies. We analyze the problem in general, and we point out particularly the shortcoming and drawbacks of the previous proposals. We show how the existing strategies focus only on code and do not address the maintenance problem of the test data namely the analysis and design models. Many of the latter strategies are also inappropriate for O-O software.

3. Design Capture and Characterization for change Impact Analysis

3.1. Introduction and motivation

The software models produced during the IIDP represent different views of the systems (e.g., structural, dynamic, functional) at different development phases. A single model is expressed using a certain formalism and notations (e.g., the behavior of a class can be modeled as a state transition diagram) depending on the O-O methodology adopted during the development process [11,18,19]. Managing the evolution of these models is a complex problem as was recognized in [13]. In this chapter we show how to generate, from the analysis and design models, the *class specifications,* and express them using the regular expression formalism [2,39]. We discuss how such a formalism can provide a common framework to abstract and express the different notations used in O-O methodologies and therefore help to reduce the complexity of *change impact analysis* [29].

We use two concepts the Method and Message Sequence Specifications (MtSS & MgSS) introduced in [2] to model the class behavior. The MtSS specifies the causal order in which the methods of an instance object can be invoked. The MgSS specifies the causal order in which messages are sent out from an object to client objects. For each class the MtSS and MgSS define the intra and inter class interactions.

The chapter is organized as follows. First, we give an overview of the regular expression formalism. We present background information, definitions and theorems that we will use later in our method. Second, we briefly describe the process of generating the MtSS and MgSS from existing design and analysis models using different examples. The process itself depends on the formalism and notations used by the designer. For further information please refer to [2].

3.2. Class Behavior Characterization (Method & Message Sequence Specification)

3.2.1. Existing work

Many O-O analysis and design methods provide models and formalisms to specify the dynamic behavior of objects and their interactions. For example Jacobson [11] prescribes the user cases, Interactions diagrams and state transition diagrams. Rumbaugh [18] proposes in he OMT method event traces and state charts. Similar notations are also provided by other O-O development methodologies [21].

In the following sections we define the Method and the Message Sequence Specification for classes and we describe how we can generate them from existing O-O analysis and design models. The MtSS and MgSS concepts were introduced first in [1,61]. The Kirani and Tsai used them to perform consistency and correctness checks of a single hierarchy of classes. Their objective is to improve the validation and verification activities during the development phase. In this proposal we use the same basic definitions of the MtSS and MgSS; however we extend and add new concepts to their work that can be summarized as follow:

- The MtSS and MgSS are used to perform regression analysis during an iterative incremental development process. Rules to detect the changes introduced the interclass and intraclass interaction are listed and demonstrated.

- In [61] Kung and Hsia provide rules to check the consistency and correctness of MtSSs and MgSSs for given class and its subclasses. The only class relationship considered in their proposal is a single inheritance hierarchy. In this research, we extend the set of rules to detect changes introduced to the same version of a class during different iterations.

- We introduce a regression algorithm to detect the obsolete set of test cases based on the MtSS and MgSS defined for each class.

3.2.2. Limitations of the MtSS and MgSS

The MtSS and MgSS are generated from the design and analysis models, as will be explained in the following sections. Here we list briefly their limitations:

- The MtSS and MgSS of a class represent an incomplete abstraction of interclass and intraclass interactions because they may have been generated from an incomplete design model at that stage of development. This is expected in an IIDP, where models are built incrementally.

- The method assumes a certain detailed level of design, which means that class, method and variable names are significant and are used in the implementation models. This is justified by the following reasons. First, it reconciles the class implementation and its specification (see Chapter 5). Second, the MtSSs are used to reselect test cases to rerun during regression analysis and testing activities. One component of a test case is the sequence (Chapter 2,section 2.4.4.4) of the exchanged messages between the interacting objects. Names of methods correspond to the actual implementation.

- The MtSS and MgSS do not incorporate data in their structure. The contents of messages sent from one class to another, and the class data, is irrelevant to the regression analysis.

- Issues related to real time constraints and concurrency are not addressed in this proposal This may be considered as an area for future investigation.

- The method for generating the MtSSs and the MgSSs is semi-formal, and depends on the formalism and notations used in the design.

- The MtSS and MgSS are a text based notation. Thus it is not a user friendly interface. It is highly recommended to generate, manipulate and store them in a completely transparent manner way to the end user (designer /developer).

- In case of a large software component (class) and because of the inherent complexity of the algorithm of matching the regular expressions, our regression analysis becomes unfeasible. The time and space needed to perform such analysis grow exponentially.

- The MtSS and MgSS do not describe the complete functionality of a given class. They simply describe some proper sequences of the class interactions with other classes and the order and sequence of the its method calls. This constraint restricts the completeness of our regression analysis.

We will point out any other important limitations as we describe the different concepts of the TOOQE methodology.

3.2.3. MtSS (Method Sequence Specification) for Intra-Class behaviors

3.2.3.1. Regular expression, regular languages and regular grammar

3.2.3.1.1. Basic definitions

In the following we define the regular expression formalism that we are going to use for expressing the class specification. For further discussion and details please refer to [1,39].

Definition 3.1:
Regular expression: We construct a regular expression from primitive constituents by repeatedly applying certain recursive rules.

Let Σ be a given alphabet. Then

1. $\varnothing$, λ and $a \in \Sigma$ are all regular expressions. These are called primitive regular expressions.

2. If r_1 and r_2 are regular expressions, so $r_1 + r_2$, $r_1.r_2$, r_1^*, and (r_1),

3. A string is regular expression if and only if it can be derived from the primitive regular expressions by a finite number of application of the rules in (2).

Definition 3.2:
Language: The language L(r) denoted by any regular expression r is defined by the following rules:

1. $\varnothing$ is regular expression denoting the empty set.

2. λ is a regular expression denoting $\{\lambda\}$.

3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$.

5. $L(r_1.r_2) = L(r_1)\, L(r_2)$.

6. $L(r_1^*) = (L(r_1))^*$.

7. $L((r_1)) = L(r_1)$.

Theorem 3.1:
Regular Language: Let r be a regular expression. Then, there exists some non deterministic finite acceptor that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

For the proof of the theorem, and further discussion, please refer to [39].

3.2.3.1.2. Regular Grammar and the class specification

Instances of classes support a set of methods that can be invoked by sending messages from client objects. Each object maintains a state. The state of an object is modified when an instance method of the class changes the attribute value of the object. When an object receives a message, the result of the execution of a method in response to the received message depends on the state of the object. The state of an object depends on the sequence of methods the object has already executed, which depends on the messages the object has received. Thus both the object state and behavior depend on the sequence in which the methods are invoked. The correct object behavior is possible only if the methods are invoked in a correct well-defined sequence. In this section, we describe the MtSS for specifying the relationship between the methods of a class.

Methods of a class can have different relationships among themselves. The causal relationship between the methods specify the sequence in which the methods should be invoked such as method m_1 should not be invoked before method m_2. We use the *MtSS* defined below for specifying this causal relationship.

The strict sequence among the methods of a class depends on the functionality of the class. For example, the instance of a Stack class should receive a first *pop()* message only after receiving a least one *push()* message. Some of the causal ordering rules can be due to implementation issues. For example, C++ objects must receive a constructor before receiving any other message, and the delete message must be the last message. All other messages must be received between these two types of messages.

Note that the set of the strict method sequences is not sufficient to describe the complete functionality of a given class. By using regular expression or a finite state machine, there is no way to formally express the number of pop messages sent to a Stack is equal to the number of preceding push messages received by the Stack.

To represent *the method sequence specification of a class C* denoted as ($MtSS_C$), we use a regular grammar [39] over an alphabet (referred as Σ) consisting of methods from *Methods(C).* where *Methods(C)* is defined as follows:

Definition 3.3:

Methods(C): For a Class *C* we define *Methods (C)* as the set of all instances of methods defined in *C* that are *publicly* available.

Example *:* For a stack the method set can be Methods(stack) = {push, pop, isempty?, isfull?, top}

Definition 3.4:

The regular grammar is of the form [39]:

$l_1 \rightarrow r_1$
$l_2 \rightarrow r_2$
$l_n \rightarrow r_n$

where, each l_i is a distinct label, and each r_i is a regular expression over the methods in $Methods(C) \cup \{l_1, l_2, .., l_{i-1}\}$.

We use the following rules to define the class specification based on the regular expression formalism:

- Two methods m_1 and m_2 related as $m_1 . m_2$ implies that one or the other is a sequential order between m_1 and m_2 namely the method m_1 is invoked before m_2 is invoked.
- Two methods m_1 and m_2 are related as $m_1 | m_2$ or $(m_1 + m_2)$ implies that either of the method can be invoked, but not both. The operator '|' is an exclusive or operator.
- The regular expression symbol '*' specifies zero or more repeated instances, and '+' denotes one or more repeated instances of a method or group of methods.

3.2.3.2. Class Method Sequences, Sequence Specifications, Safe Sequences, MtSS and Properties

In the following we give the formal definition of the method sequence specification of a class and other related concepts. For further details please refer to [2].

Definition 3.5:

Method sequence: A method sequence *S* is a finite sequence over a method set *M* of *C*, $(m_0, m_1, ...m_n)$, where each $m_i \in M$. A method sequence need not contain all the methods in the

method set. The entries in a method sequence correspond to a causal order in which the methods can be invoked.

Example: In the Stack example, a possible method sequence is (push,top,pop,isempty?). In this method sequence, the method 'push' must be invoked before the method 'top' is invoked for all instances of class Stack.

Definition 3.6:
SeqSpec(C): A *SeqSpec(C)* is a specification of *C* that defines a sequence relationship between all the methods of C. We use a regular grammar for specifying the *SeqSpec(C)* [2,39].

Example : Let us consider a specification of a simple bank account class. The interface methods defined in class Account are:
Σ={create,deposit,open,withdraw,close,delete}.

'Create' is a constructor method and delete is the destructor method. The 'open' method is used for opening an account, 'deposit' method is used for depositing positive sums of money into the account, 'withdraw' method is used for withdrawing money from the account and close method is for closing the account. In the following we describe a MtSS for the class Account. In the sequence specification, the regular grammar labels such as Account Methods are present on the left hand side of the arrow [2,39].

The SeqSpec(Account) is:

SeqSpec -> create.AccountMethods.delete (MTSS description)
AccountMethods -> open.TransactionMethods.close

*TransactionMethods ->(deposit.(deposit | withdraw) *)*

In the above sequence specification, 'create' is the first method the instance of class Account executes before any other methods. Once the method 'open' is executed, the instance can invoke either 'deposit' or 'withdraw' methods any number of times before executing the 'close' and 'delete' methods. Thus the MtSS of class Account indicates the causal order in which methods can invoked for the correct behavior of instances of the class.

Definition 3.7:
SafeSeq(C) : A *SafeSeq(C)* defines a set of all sequences S_i that can be derived from *SeqSpec(C)* of the class *C*. A sequence in *SafeSeq(C)* is a valid sequence of messages accepted by any instances of the class C. *SafeSeq(C)* is the regular language defined by *SeqSpec(C)* [2,39].

Example : The sequence specification of class Account given in the previous example defines the set of all possible valid sequences of messages that an object of class Account can receive. This set of sequences is *SafeSeq(Account)*:

{(create. open.deposit.withdraw.close.delete) ,
(create. open.deposit.deposit.withdraw.close.delete) ...}

3.2.3.2.1.Properties satisfied by the MtSS:

In this section we give two properties satisfied by any MtSS. These properties are derived from regular definitions and therefore, these properties will not be discussed in depth

Property 1: *Associativity over '.'* : for methods m_1, m_2, and m_3 of a class, then

$$m_1.(m_2.m_3) = (m_1.m_2).m_3 = m_1.m_2.m_3$$

The brackets do not change the causal order in which the methods are invoked.

Property 2: *Distributivity of '.' over '|'*, for methods m1, m2 and m3 used in a MtSS of class

$$m_1.(m_2 \mid m_3) = (m_1.m_2) \mid (m_1.m_3)$$

The proof is given in [1,2,39].

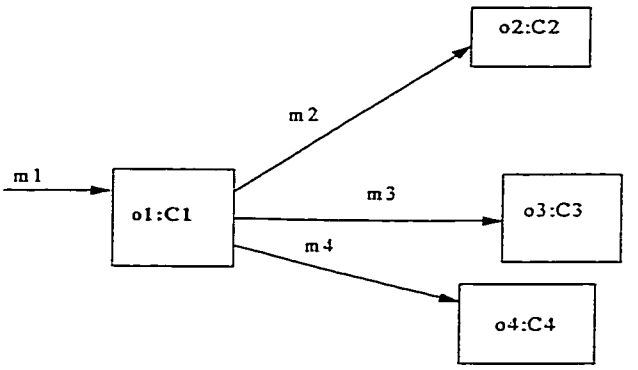
3.2.3.3. MgSS (Message sequence Specification) for inter-class behaviors:

In this section, we discuss the MgSS that can be used to describe the causal order among methods supported by different objects.

Definition 3.8:

The *MgSS* is specified for each method that sends messages. MgSSs and MtSSs together can identify the causal order between any two methods defined in different classes. Although the MgSS for a method that sends no messages to other objects is null, the MgSS is useful for those objects that act as clients to other objects [2].

A MgSS of a method identifies all the messages it sends to other domain objects. It also identifies the causal order in which the message is sent. For describing MgSSs, we use the same regular grammar formalism as with MtSS. In addition to the operators '.' , '|', '+' and '*' we introduce a new operator '?' that indicates that a message prefixed to the operator '?' is optional.



Upon receiving the message m1, The object O1 sends respectively three messages (m2,m3 and m3) to the interacting objects O2,O3 and O3

Figure 3-1: Diagram for illustrating MgSS

In the following we describe the MgSS of a method of a class. Figure 3-10 contains four objects O1, O2, O3 and O4. The method *m*1 of O1 sends messages to O2, O3 and O4. The order in which the messages *m*2, m3 and *m*4 are sent to objects by message m1 forms the MgSS for method m1. If the messages are sent in the order of *m*2, *m*3, and *m*4, then the MgSS of *m*1 is:

$$m1_{O1} \Rightarrow (m2_{O2}.m3_{O3}.m4_{O4})$$

If either m2 or m3 are sent before m4, then the MgSS of m1 is, $m1_{O1} \Rightarrow ((m2_{O2} | m3_{O3}).m4_{O4})$

For a given method, its MgSS can be determined by how the method interacts with other objects. In the following we discuss briefly how different scenarios lead to different MgSS.

- If a method sends messages of objects one after the other, then all of the messages represented in the order in which they are sent. For example, if m1, m2 and m3 are sent in sequence, then the corresponding MgSS will be *m1.m2.m3*.

- Optional messages are represented in MgSS using the operator '?'. For example, if *m1* is sent out optionally and then *m2* and *m3* are sent, the MgSS will be *m1?.m2.m3*.

- If a method sends out messages alternatively (for example , a true condition results in one message and false condition results in another message,) the MgSS will contain both messages as alternatives. For example , if *m1* or *m2* is sent out, but not both, then the corresponding MgSS is *m1| m2*.

- If a method sends out messages repeatedly, such as in a loop, the operator '*' or '+' is used. The operator '*' is used when messages are sent out zero or more times. The operator '+' is used when messages are sent out one or more times. For example if a message m1 is sent one or more times, followed by *m2*, the corresponding MgSS is: $(m1)^{+}.m2$.

3.2.4. Derivation of MtSS & MgSS: Principles and examples

In the following sections, we describe the method of generating the method and message sequence specification from the analysis and design models. Throughout the discussion, we use the following examples: a simplified ATM system, a photocopy machine and object model of a simple graph. We do not focus in these examples on the analysis and design aspects of the different systems. Instead, we just focus on the process of abstracting those models and generating the MtSS and MgSSs of a class.

3.2.4.1. Derivation of MtSS: Principals and examples:

3.2.4.1.1. From State Transition diagrams:

Many OO analysis and design techniques propose State Transition Diagrams (STD) for modeling the dynamic behavior of classes [17,18,19]. They can be used to model both inter-class and intra-class dynamic behavior [19]. For deriving the MtSS of a class we use its intra-class STD. The STD model represents all possible states of the class, the events that can cause state transitions, and the actions that result state changes. The STD for a class describes how the sequencing of external events can affect the state of an object of the class.

Each state of an object captures a combination of values of the attributes. The states are linked to one another through state transitions. State transitions are caused by the occurrence of events which are stimuli received from the environment. Thus, the combination of the attribute values define what the state of an object is, while the messages received from the environment are the events.

The MtSS for a class can be derived automatically from the STD specification of a class. For deriving the MtSS, we use only the events. We do not consider states and actions associated with state transitions.

The set of sequences of messages accepted by an STD of a class forms the safeSeq(C). Construction of a regular grammar that accepts the same language as that of STD is well studied in the literature. Several efficient algorithms exist for deriving the regular expression from the STD [3,39]

Example

In following we illustrate the MtSS derivation from the STD of a photocopy machine [17]:

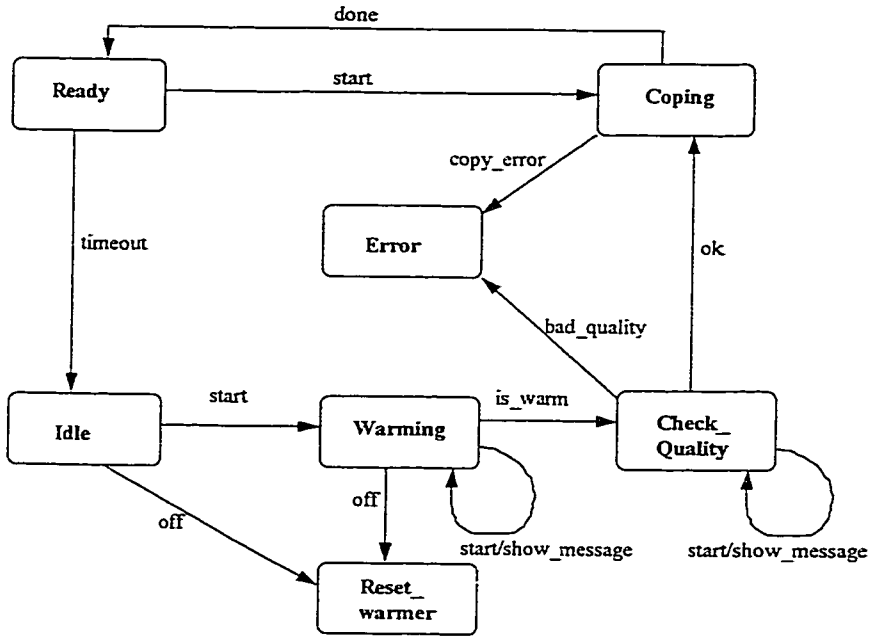


Figure 3-2: STD of the Photocopy machine class

The MtSS of the Photocopy class is defined by the following regular grammar:

SeqSep -> start.**CopyingMethods**| timeout.**IdleMethods**

CopyingMethods -> copy_error| done.**SeqSep**

IdleMethods -> start.**WarmingMethods**|off

WarmingMethods -> is_wram.**CheckqualityMethods**|startShowMessage^{+}|off

Checkquality -> startShowMessage^{+}| ok.**CopyingMethods.**

3.2.4.1.2.From State Charts (ObjecTime ROOM Charts):

For many applications, state transition diagrams grow large and cumbersome. State charts extend the STD to deal with those problems. They introduce new concepts such as state actions, conditional state transition, nested states (ROOM Charts) [66]. Existing O-O methodologies prescribes state charts for modeling the behavior of the system, however each one uses certain features of the formalism.

In the following, we give an example of generating the MtSS of a photocopy machine from a state chart with nested states. Many copiers will switch into power saving mode after they have not been used for a while. To make the model more readable and consistent, the original state diagram can be modified by adding a new state called PowerSave that encapsulates the Idle, Warming, and Ckeck_Quality states.

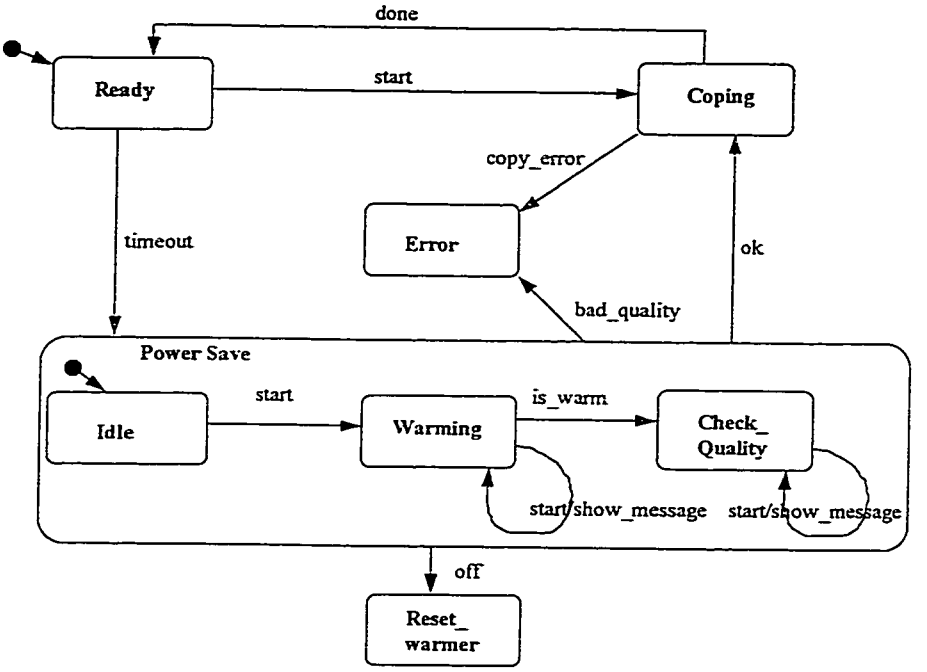


Figure 3-3: ROOM Chart for a copier machine

The MtSS generated for the class consists of the two regular grammars:

SeqSepc -> start.**CopyingMethods**|timeout.**PowerSaveMethods**

CopyingMethods -> copy_error| done.**SeqSep**

PowerSaveMethods -> SeqSpecStatePowersave(ok.**CopyingMethods**|
off|.bad_quality)

SeqSpecStatePowersave -> start.**WarmingMethods**

WarmingMethods -> is_wram.**CheckqualityMethods**|startShowMessage*|off

Checkquality -> startShowMessage*

3.2.4.1.3.From Interaction Diagrams:

Interactions diagrams (IDs) describe how each use case is presented by communicating objects. In [11] Jackobson provides a formal definition of an interaction diagram and how we can draw them. Here, we focus on the issue of generating the MtSS from such models. The ID shows a set of pairwise messages exchanged between entities (objects) in a system and the relative order in which they occur. There are similar notations in other O-O methodologies, such as event traces from the OMT method [18] and message sequence charts from the ROOM method (specified in ITU Z.120 Recommendations). The underlying mechanism is the same in each of the above models, however the formalism differs slightly.

To generate the MtSS of a class, we assume that the designer has already the set of IDs where the class is referenced, and that the names used for the classes and messages correspond to their actual implementation. To determine the MtSS for a given class, the messages sent to it from other interacting objects are considered. The sequence is determined based on the order described in the IDs. We note that we do not solve the problem of concurrent messages. The process of generating the MtSS can be easily automated based on the formalism used to describe the IDs.

Example: To determine the MtSS for ATMHardware Class (see Figure 3-4: Deriving MtSS and MgSS form interaction diagram), we consider the messages received from ATMHardware from other interacting objects. The incoming messages are ReadAmt and DispCash. The order is determined by the order in which the messages are generated, whether the messages are conditional or in a loop, etc. Using the above technique, MtSS for all the classes are given below:

$MtSS_{ATMHardware}$ -> *readAtm.DispCash*

$MtSS_{Bank}$ -> *withdrawReq*

$MtSS_{Account}$ -> *getAcctInfo.getAccSumm.balance.withdraw*

3.2.4.2. Derivation of MgSS: Principals and examples

3.2.4.2.1. From Interaction Diagrams

Interaction diagrams are used in O-O analysis and design techniques for representing the interaction between objects [19,11]. In [19], Booch uses interaction diagrams for representing use cases. Use cases are used predominantly for representing analysis and design information. Interaction diagrams are used to describe how each use case will be implemented by communicating objects. In this proposal, we use interaction diagrams as given in [11] and provide an outline for deriving MgSS and MtSS.

The MtSS corresponds to the causal order in which methods at an object are invoked. Figure 3-4, only one use case is shown and thus the MtSS generated from the above is not complete. However, once all use case descriptions are available, one can generate the complete MtSS.

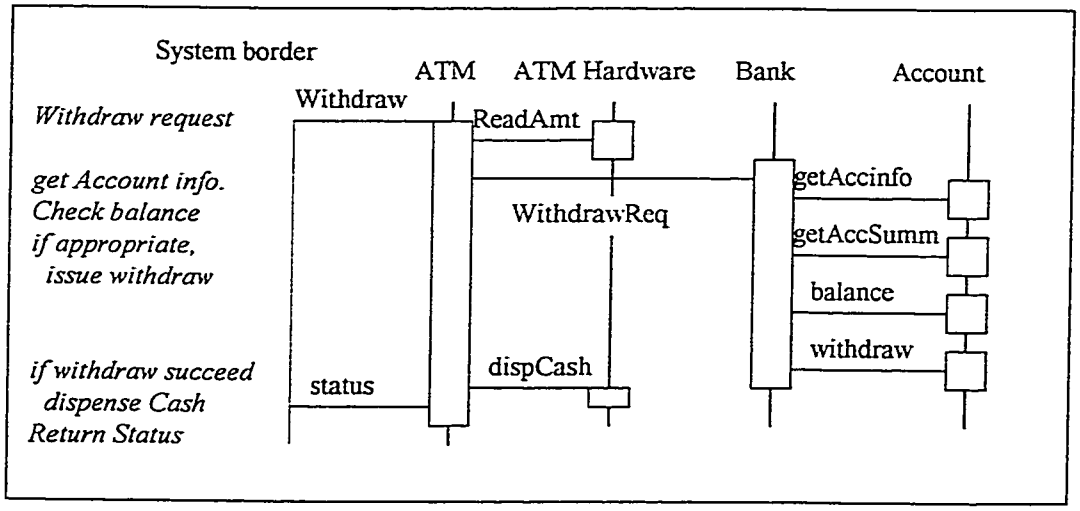


Figure 3-4: Deriving MtSS and MgSS form interaction diagram

To derive a MgSS from an interaction diagram, the messages that are sent from an object are considered. For example, in the object Bank, various messages are sent, such as AccInfo. Once all outgoing messages are known, the causal order among them is determined. The decision whether some of the outgoing messages are optional or whether they are repeated multiple times, is based on the description provided in the IDs. The MgSS derived for some methods (events) is shown below:

WithdrawATM -> $readAmt_{ATMHardware}.withdrawReq_{Bank}.dispCash_{ATMHardware}$

$withdrawReq_{Bank}$ -> $getAcctInfo_{Account}.getAccSumm_{Account}.balance_{Account}.withdraw_{Account}$

In the MgSS generated for the method *withdraw()*, three messages are to different classes and the causal order among them is identified.

3.2.5. Inheritance and the MtSS and MgSS

Inheritance is a concept introduced with the O-O paradigm. Objects identified during the analysis and design models and having common characteristics (behavior and information structure) are organized in inheritance hierarchies[18]. A child class inherits the methods and instance variables defined in parent classes. A child class, in addition to the inherited methods, can further enhance the class with additional methods or change the implementation of the inherited methods [11,18,19]. Inheritance can be used for specialization, refinement and reuse.

- Specialization: The child class satisfies the semantics of all the inherited methods from the parent class, and may be extended with additional methods and instances variables.

- Refinement: The child class may modify the semantics of the inherited methods. The latter methods are refined and modified. Therefore, the behavior of the child class's methods may be different form the behavior of the corresponding methods in the parent classes.

- Reuse: The child class may not inherit all methods from the parent class, and may modify the semantics of the inherited methods.

3.2.5.1. Discussion

The MtSS and MgSS characterize the behavior of a class. An object may send a message to another object which is not defined in the actual implementation of the receiver instance, but inherited from its parent classes. The MtSS description as defined earlier describes the sequence of the invoked methods at an instance of a class, including the inherited methods.

In the MgSS definition, only the methods declared in a class are used to describe the inter class interactions. This assumes that only refined methods are included in the MgSS definition. The methods invoked may be inherited in the receiver object (see figure 4-4).

3.2.6. Implications of polymorphism on the MtSS and MgSS definitions

In this section, the concept of polymorphism is introduced briefly. Then, its implications on the use and the generation of MtSS and MgSS of classes is discussed.

Instances of a class share the same methods. Each method has a target object as an implicit argument. The behavior of the method depends on the class of its target. An object "knows" its class and hence the right implementation of the method [18].

The same method may apply to many different classes. Such method is polymorphic, that is, the same method takes on different forms in different classes [11]. For example, the class File may have a method called print. Different methods could be used to print ASCII files, print binary files and print digitized pictures. All the methods perform the same taskprinting a file. However, each method may be implemented by different code. This common form of polymorphism is known as "method overriding". Another example is to use the same method with different types (operator + can be used to add an integer and a real) known as coercion of types. Because of the ability to refer to more than one class of object, a polymorphic reference has both a static and a dynamic type associated with it [11]. The static type is known at compile time and determines the set of valid types that can be accepted at run time. The dynamic type is not known at compile time and can change during the execution. For further details about the formal definition of polymorphism (ad hoc, universal) , and how is implemented refer to [18].

3.2.6.1. Discussion

Polymorphic methods are identified at the analysis and design level. Their implementation depend on how the programming language implements polymorphism. For example in a class hierarchy, superclass may contain an abstract declaration of a method, and each subclass may

provide its own implementation. The same name and signature are used to indicate that this method is polymorphic (see Figure 3-5). In a C++ environment, such method will be implemented as virtual in the superclass and subclasses must provide their own code.

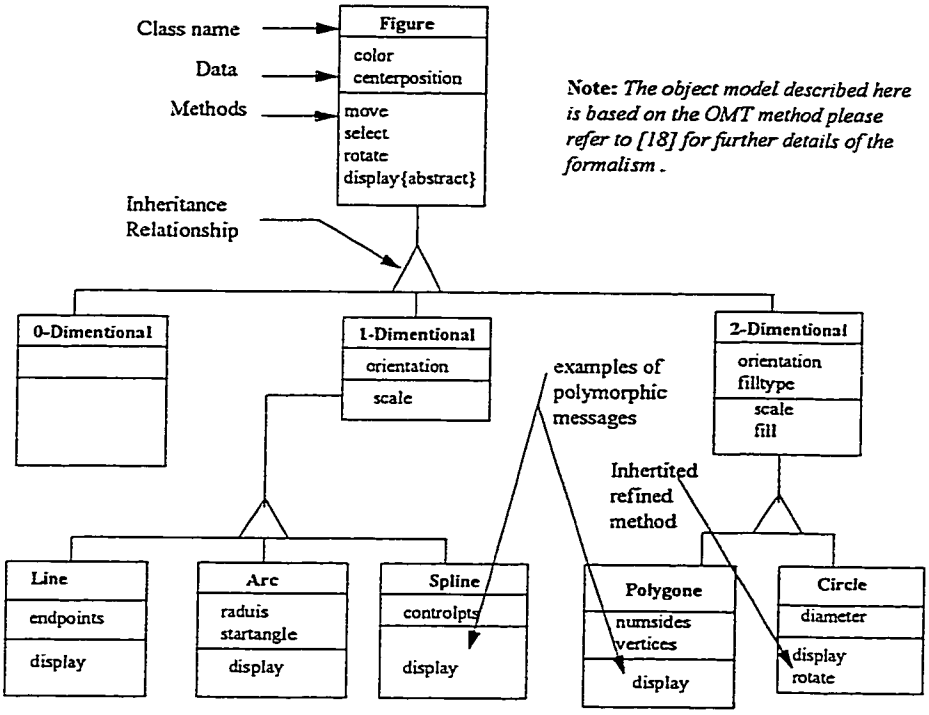


Figure 3-5: Inheritance hierarchy for a Graphic figure

Each class in the analysis and design model has its own MtSS, which describes the causal order in which its methods are invoked. At this point of abstraction, it is assumed that the receiver instance of the messages will be identified at compile time or run time. There are no restrictions on using the same name of a method in different MtSSs of different classes.

Similarly, the MgSS describes the causal order in which messages are sent by a method to different objects. The MgSS specifies which messages are sent, and to which object. This means that the same message can be sent to different objects with exactly the same meaning. Hence polymorphic methods do not affect the MtSS and MgSS. The analysis and design models shape the structure of the MtSSs and MgSSs. Depending on the level of abstractions or details expressed in the previously mentioned models, the polymorphic messages are identified and included in the MtSS and MgSS description.

Example: This example shows how we can use inherited and polymorphic methods in the definition of MtSSs (see Figure 3-5) :

The partial MtSS of the *Circle* and *Arc* classes are defined as follows:

$MtSS_{Circle}$ -> *display.select(move|rotate|scale|fill)**

$MtSS_{Arc}$ -> *display.select(move|rotate|scale)**

3.3. Chapter summary

In this chapter, we provided a method for capturing the design, and characterized the changes that may occur during an IIDP. We introduced two concepts: the Method and Message sequence specification to express class specifications based on the regular expression formalism. Then, we showed how to generate these sequences from existing analysis and design

models. The results discussed in this chapter represent the basis of our research and they will be used throughout the whole thesis.

4. Class changes in the Iterative Incremental Development Process

4.1. Introduction

Objects and classes are the basic units for building O-O systems. Thus, any changes introduced to the analysis and design models will be reflected in their structure and behavior. In this chapter a description of the types of changes that can be introduced to the class, from design and implementation perspectives, is provided. These common changes are considered as guidelines to help the development team in identifying and revalidating affected software components. We focus on two regression analysis and testing activities:

- *Identification of the class design alterations between macro-iterations.*

  This is done by comparing the old versions of the MtSS and MgSS with the new ones generated from the current iteration.

- *Reselection of test cases based on the class specification and change types:*

  It has been established that test cases for O-O software are divided into three categories: functional test cases (black-box), structural test cases (white-box, grey-box), and interaction test cases (grey-box). Code based regression analysis and testing strategies focus on the first two categories, since they perform control and data flow analysis to detect the modification introduced to the code [12]. For example, after comparing two versions of a data flow graph, a regression algorithm may detect a data item that is not used anymore (no def-use). This implies that all functional test cases using that data in their definition become obsolete.

In this research, we focus on interaction test cases, and we describe a regression analysis and testing algorithm that uses the definition of the MtSS and MgSS to detect and classify the design alterations and solve the reselection of test case problem. Hence, we stress the *intra-class and inter-class interactions.*

Note that our method does not cover all possible changes. We assume that the proposed strategy can be used in conjunction with other regression analysis and testing, and analysis code or specification based strategies to cover the entire test case data base.

4.2. Class Changes

Table 4.1 provides a classification of the types of changes that can be introduced to a class [23] during the IIDP. They are:

- *Data change:* Any datum (eg., a global variable, a local variable, or a class data member) can be changed by updating its definition, declaration, access scope, data access mode or initialization. In addition, adding new data or deleting existing data are also considered to be data changes.

- *Method Change:* A member function or a method can be changed in various ways. Here we classify them into three types : *Interface changes*, *control structure changes* and *component changes*.

  - The interface of a member function or a method consists of its signature, access scope and mode, and its interactions with other member functions (for example, a function call or message passing). Any change to the interface is called an interface change of a member function.

  - Structure changes include: 1) adding, deleting or modifying a branch or a loop structure, and 2) adding or deleting a sequential segment.

  - Component changes include: 1) adding, deleting, or changing a predicate; 2) adding or deleting a local variable; and 3) changing a sequential segment.

- *Class change:* Direct modification of a class can be on of the three types: component changes, interface changes and relation changes. Any change to a defined (redefined) member function or a defined data attribute is known as a component change. A change is said to be an interface change if it adds or deletes a defined (redefined) attribute, or changes its access mode or scope. A change is said to be relation change if it adds or deletes an inheritance, aggregation or association relationship between the class and another class.

- *Class interaction change:* This includes: 1) changing the defined members of a class; 2) adding , or deleting a class; 3) adding, or deleting a relationship between existing classes; 4) adding, or deleting an independent class.

Scope	Change types	Covered by TOOQE
Data changes	1. Change data definition/declaration 2. Change data access scope/mode/uses 3. Add delete data	
Method interface changes	4. add/delete external data usage 5. add/delete external data updates 6. add/delete change a method call 7. change its signature	√
Method structure changes	8. add/delete sequential segment 9. add/delete change a branch loop	
Method component changes	10. change a control sequence 11. add/delete local data 12. change a sequential segment	
Class component changes	13. change a defined/redefined method 14. add/delete a defined/redefined method 15. add/delete a defined datum 16. add/delete a virtual abstract method 17. change an attribute access mode/ scope	√ √ √

Class relationship changes	18. add/delete a superclass 19. add/delete a subclass 20. add/delete an object pointer 21. add/delete an aggregate object 22. add/delete an object message	
Class interaction changes	23. change a class (modify attributes) 24. add/delete a relation between classes 25. add/delete a class and its relations 26. add/delete an independent class	 √ √

Table 4-1: Changes that can be introduced to classes

Our method for detecting changes does not cover all of the above change types. In the following sections we will provide a set of rules for analyzing the ripple effects of changes, and we will cross reference the change types covered

4.3. Change detection and Classification

4.3.1. MtSS operations and languages

Based on our previous definition, the MtSS and MgSS describe the behavior of the system from different points of view or perspectives. Moreover, any kind of change (especially changing the specification and the internal structure of a class) will be reflected directly in their derivation. Hence, to detect the class changes, one can easily compare the MtSS of a class generated during one iteration with the new MtSS generated from the modified version during the next iteration. In a similar way, we detect the changes affecting the collaborations between the objects by simply comparing the original MgSS and the MgSS of the modified version. This allows the designer to have the necessary feedback to control the evolution of the system, especially from a use case perspective.

Let C_1, C_2,.. ,C_k be a set of interacting classes that form a system or a sub-system. We assume that they can be related to one another through inheritance, association or aggregation relationships [18]. We now define terms that are used in detecting class changes based on the analysis and design models.

Definition 3.9:
ε-replace(MtSS,S): ε-replace(MtSS,S) is defined as an operation on the *MtSS* where it substitutes all the methods of *S* in *MtSS* by *ε (null method).* The null method has no effect on how the MtSS is defined.

Example: If MtSS of a class C is $m_0.m_1.(m_2|m_3)^*$, and S ={ m_1,m_3}, then *ε-replace(MtSS,S)* $=m_0.(m_2)^*$

Definition 3.10:
Language(MtSS$_C$) : Language(MtSS$_C$) of class *C* defines a set of valid method sequences that can be generated from the sequence specification *MtSS* for *C.*

Example: If MtSS of a class C is $m_0.m_1.(m_2|m_3)^*$, then *language(MtSS$_C$)* = {$m_0.m_2$, $(m_0.m_1.m_3)$, ...}

4.3.2. Detection and classification rules:

We now give rules to compute sets which represent changes of different types discussed in section 4.2.

4.3.2.1. Adding, deletion, modification of methods

In the following section, we list a set of rules that characterize modifications introduced to the class specification. They cover of some the changes listed in the Table 4.1, namely the ones listed below, corresponding to each rule.

Given a class C_1 and the modified version of the same class denoted by C_2, let *Method(C1)* and *Method(C2)* be the set of methods defined in C1 and C2 respectively. We define $MtSS_{C1}$ and $MtSS_{C2}$ to be the Method Specification Sequences of C_1 and C_2 respectively. We define $L1 = Language(MtSS_{C1})$, and $L2 = Language(MtSS_{C2})$.

- **Rule 1 (Deleted methods) :**

if $(Method(C1) - Method(C2) \neq \emptyset)$ then any $m_i \in (Method(C1) - Method(C2))$ must be a deleted method. We define $Deletedmethod(C1) = (Method(C1) - Method(C2))$.

Discussion:

This rule is based on the set operation "-": Given any two sets A and B the set *(A- B)* is defined by the set $A\text{-}B=\{x;\ x \in A \text{ and } x \notin B\}$.

- **Rule 2 (New methods) :**

if $(Method(C2) - Method(C1) \neq \emptyset))$ then any $m_i \in (Method(C2) - Method(C1))$ is a new method, .we define $Newmethod(C2) = (Method(C2) - Method(C1))$.

Discussion:

This rule is based on the above discussion for the "-" set operation.

- **Rule 3 (Retained methods) :**

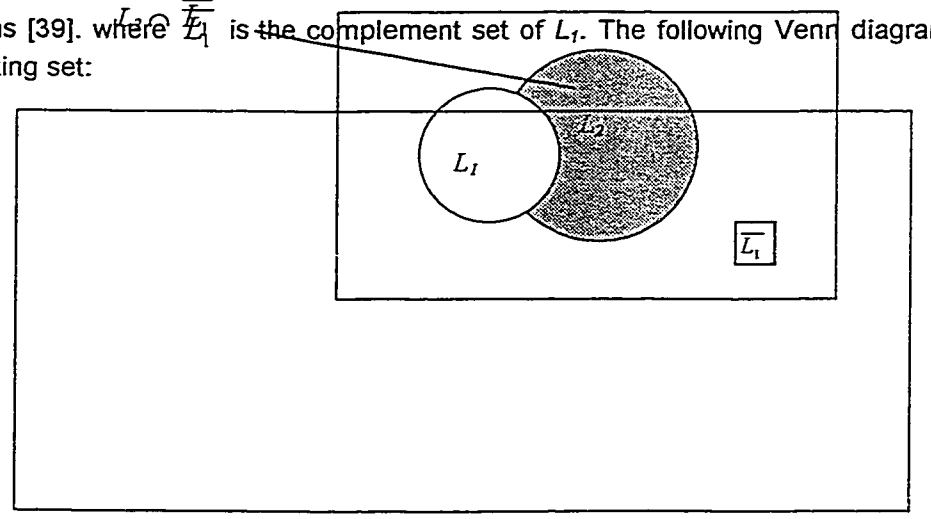
The retained methods are defined by the set $Retainedmethod(C2)=(Method(C1) \cap Method(C2))$. If a method $m_i \in Retainedmethod(C2)$ is changed, then a residual defined/redefined method is changed in *C2*.

- **Rule 4(New Sequences of methods):**

Let $L= (Language(MtSS_{C2})- Language(MtSS_{C1}))$. *L* defines the set of the new method sequences accepted by *C2* and not by the original version *C1*. This may occur because of one of the above types of changes (Add/Delete).

Discussion:

Based on the theorem of closure of regular expressions, $L=L_2\text{-}L_1= L_2 \cap \overline{L_1}$ under set operations [39]. where $L_2 \cap \overline{L_1}$ is the complement set of L_1. The following Venn diagram illustrates the resulting set:



$L = L_2 - L_1$ defines the set of all the sequences that are not accepted by the original version of the class. This is an important rule, since it allows us to identify the new sequences of added or modified methods that need to be considered in regression analysis and testing. These sequences correspond to the set of possible scenarios or test cases that should be added to the regression test suite.

- **Rule 5 (Deleted Sequences of methods)**

Let $L' = (Language(MtSS_{C1}) - Language(MtSS_{C2}))$. L' defines the set of the old method sequences accepted by *C1* and not by the modified version *C2*.

Discussion:

This rule is similar to the Rule 4. L' defines the set of all the sequences that are not accepted by the modified version of the class. This is also an important rule since it characterizes the sequences of methods which have become invalid or obsolete. This a major objective of regression analysis and testing. The invalid sequences correspond to a set of possible scenarios or test cases that have become obsolete and should be removed from the regression test suite.

Example:

Suppose we have the original MtSS for the Account class :

$MtSS_{C1}$-> *create.open.(deposit.(deposit|withdraw)*).close* .
The modified version of the above MtSS is defined as follows:
$MtSS_{C2}$-> *create.open.(deposit.(deposit|withdraw| balance| report)*).close.*

Two methods { *balance,report* } have been added. The regular expression may be represented by a DFA (Deterministic Finite Automaton) as described in [39] (see section 4.4.4.3). The result of applying the above rules is illustrated in Figure 4-1:

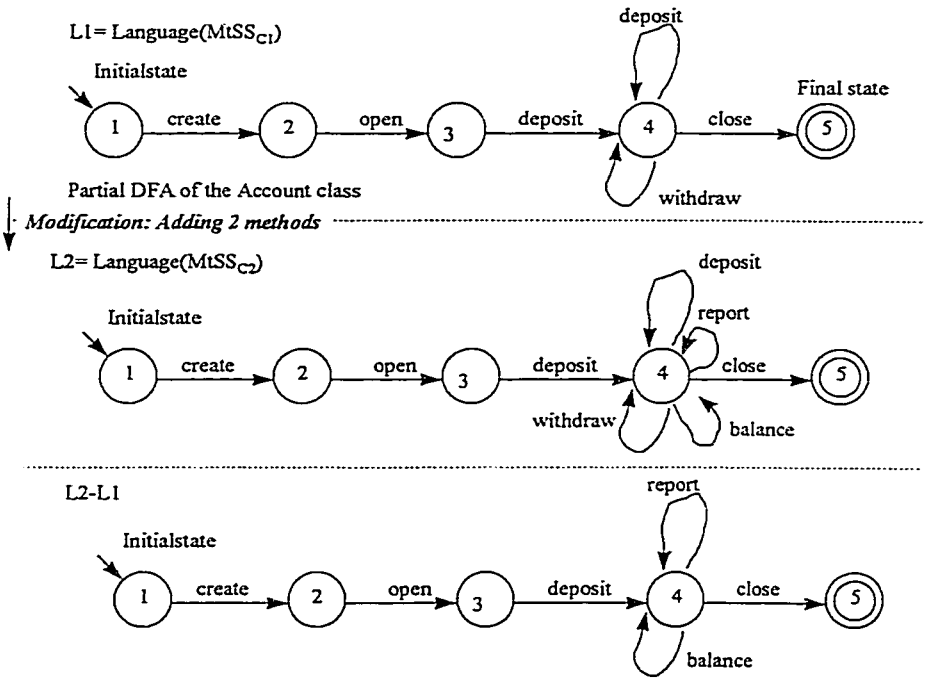


Figure 4-1: Example of deriving two MtSSs of the class Account as it evolves form C1 to C2.

4.3.2.2.Modification of the Class library Hierarchy changes

Given a class C_k and the modified version of the same class denoted by $C_{k'}$, we denote by $ParentC_k = \{MtSS_{Ci} \mid i=1..k-1\}$ the set of MtSS of the parent classes of the class C_k. Similarly, we denote by the $Parent C_{k'} = \{MtSS C_i \mid i=1..k'-1\}$ the set of MtSSs of the parent classes of the modified version of define $C_{k'}$.

Based on the rules discussed previously, and the set theory [39], we establish the following conditions:

- If ($ParentC_k$ - $Parent C_{k'} \neq \varnothing$) then any $MtSS C_i$ in ($ParentC_k$ - $Parent C_{k'}$) contains deleted scenarios or test cases from the super class.
- If ($Parent C_{k'}$ - $ParentC_k \neq \varnothing$) then any $MtSS C_i$ in ($Parent C_{k'}$ - $ParentC_k$) contains added scenarios or test cases in the super class.
- The set defined by ($ParentC_k \cap Parent C_{k'}$) represents the retained scenarios. The designer provides addtional information by inspection and walkthrough to determine the retained scenarios affected by the design changes.

4.3.2.3. Modification of the class interactions

In the following section, we list a set of rules that characterize the modification introduced to the interaction between classes based on the Message Sequence specification.These concepts are very similar to the above MtSS rules.

Let $Commonmethods(C_1, C_2) = (Method(C1) \cap Method(C2))$

Given a method $m_i \in Commonmethods(C_k, C_{k'})$, we define $MgSS_{C1}$ and the $MgSS_{C2}$ as the message specification sequences of C_1 and C_2 respectively for the method m_i:

- **Rule 6 (Deleted message interactions)**

For a method $m_i \in deletedmethod(C_1)$, the set of the deleted interactions is defined by $deletedInteractions(C_1) = MgSSC_1$ of the method m_i.

Discussion:

The deleted interactions are based on the specification of the methods deleted from the original version of the class. For each method $m_i \in deletedmethod(C_1)$, the $Language(MgSS_{C1})$ defines the set of all the deleted interactions. This rule allows us to detect the deleted interactions that needed to be removed from the regression test suite.

- **Rule 7 (New message interactions)**

For a method $m_i \in Newmethod(C2)$, the set of the new interactions is defined by $NewInteractions(C_1) = MgSSC_2$ of the method m_i.

Discussion:

The new interactions are defined based on the specification of the methods added to the modified version of the class. For each method $m_i \in Newmethod(C2)$, the $Language(MgSS_{C2})$ defines the set of all the new interactions. This rule allows us to define the new interactions to be tested and included in the regression test suite for each method added to the class.

- **Rule 8 Modification of the MgSS of an existing method (new sequence of message interactions):**

For every $m_i \in CommonMethods(C_1,C_2)$, Let $L'= (Language(MgSS_{C2})- Language(MgSS_{C1}))$, L' defines the set of the new message sequences send by the method m_i from C_2 and not by the original version of m_i in C_1.

Discussion:

The rationale behind this rule is similar to the one discussed in Rule 1.

Let $L_1 = Language(MgSS_{C1})$, and $L_2 = Language(MgSS_{C2})$ for a given method $m_i \in CommonMethods(C_1,C_2)$. Let $L=L_2-L_1= L_2 \cap \overline{L_1}$. L defines the new interactions that the modified version of m_i sends to the objects interacting with an instance class C_2. These new sequences result from adding new methods, deleting methods or modifying the original sequence of methods. The new sequences define a set of test cases to be added to the regression test suite.

- **Rule 9 (Modification of the MgSS of an existing method and invalid message interactions)**

For every $m_i \in CommonMethods(C_1,C_2)$, Let $L'_2= (Language(MgSS_{C1})- Language(MgSS_{C1}))$, L'_2 defines the set of the old message sequences sent by the method m_i from C_1 and not sent by the modified version of m_i in C_2.

Discussion:

The rationale for this rule is similar to the one discussed above (Rule 8).

Let $L_1 = Language(MgSS_{C1})$, and $L_2 = Language(MgSS_{C2})$ for a given method $m_i \in CommonMethods(C_1,C_2)$. Let $L' =L_1-L_2= L_1 \cap \overline{L_2}$. L' defines the deleted interactions that the original version of m_i defines, and that are no longer valid in the new version. These deleted sequences result from adding new methods, deleting methods or modifying the original sequence of methods. The deleted sequences can be used to detect the obsolete test cases from the regression test suite.

4.3.3. Change Consistency checking via Language Equality

4.3.3.1.Backward compatibility via ε-replace

Given a class C_1 and the modified version of the same class we denoted by C_2, we define $MtSS_{C1}$ and the $MtSS_{C2}$ the methods specification sequences of C_1 and C_2 respectively.:

Let $S_1= Deletedmethods(C_1)$ and $S_2= Newmethods(C_2)$. If we define :
$RetainedMtSS_{C1}= \varepsilon\text{-}replace(MtSS_{C1}, S_1)$ and $RetainedMtSS_{C2}= \varepsilon\text{-}replace(MtSS_{C2}, S_2)$ then
$Language(RetainedMtSS_{C1}) = Language(RetainedMtSS_{C2})$.
Discussion:

Informally in the above rule, the MtSS of the modified class with only the unmodified methods from the original version of the class must be equal to the MtSS of the original class without the deleted methods. The rationale behind the rule is based on the set theory and language operations:

$H =Methods(C_1) - Deletedmethods(C_1)= Method(C2)-Newmethods(C_2)$.

The causal relationship between the methods defined in S can be denoted $MtSS_H$. When we apply the *ε-replace* operator, we just replace the methods in the MtSS with the null method. The sequence between the methods is not affected. Hence, using the set methods defined by $Language(RetainedMtSS_{C1}) = Language(MtSS_H)$ we can similarly establish that, $Language(RetainedMtSS_{C2}) = .Language(MtSS_H)$

Example: Suppose we have the original MtSS for the Account class :

MtSS1= create.open.(deposit.(deposit|withdraw)).close* and the modified version

MtSS2= create.open.(deposit.(deposit|withdraw| balance| report)).close.*

To check the consistency of the new version of the class Account, one can replace the new added methods *balance* and *report* by the null method. This will yield the original MtSS.

4.4. Example: Analysis of change Impact on regression analysis of existing Test cases

4.4.1. Generating a subset of the language defined by a MtSS

4.4.1.1.Example: Deriving a Control Flow Graph from the MtSS

The MtSS uses the regular expression formalism to represent the causal relationship between methods. The regular expression operators that can be used in MtSS are "." , "*", and "|" described below. These operators are used to link the methods in causal relationships. In the following, we describe the method of converting regular expression operators to corresponding control flow graphs [5].

- *A.B:* The regular expression operator '.' represents a sequence between the methods A and B in sequence specification, (i.e. the method A is executed before the method B is executed). This regular expression is converted to a control flow graph consisting of nodes A and B with a direct edge between them.

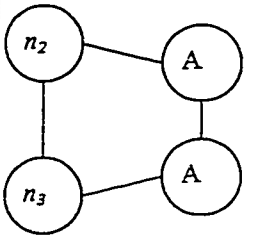
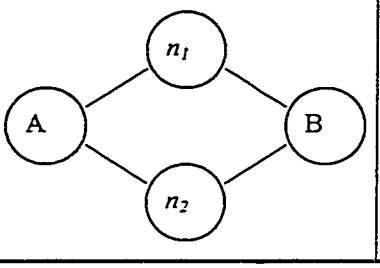
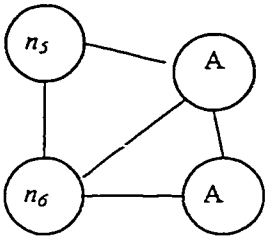
Regular Expression	Control flow graph	Regular Expression	Control flow graph
A.B		A*	
A|B		A+	

Table 4-2: Regular expression to Control Flow Graph

- *A|B:* The regular expression operator '|' represents an 'OR' operator between two methods in a sequence specification indicating that either method A is executed or B is executed but not both. This regular expression is converted to a control flow graph with 2 parallel direct paths between two dummy nodes n_1 and n_2 with nodes A and B in each path. The nodes n_1 and n_2 not read or modify any variables.

- *A*:* The regular expression operator '*' indicates that the method can be repeatedly invoked (possibly 0 times). In [22] it has been shown that for data flow analysis purposes for every 'A*' in a regular expression, one can substitute with '(1 | (A.A)) without affecting

data flow anomalies. While constructing the control flow graph, this theorem is used and the equivalent control flow graph for '*' is shown in Table 4-2 .

• A^+: The regular expression operators '+' indicates that a method can be executed repeatedly but at least one time. Extending the theorem from [22], the regular expression 'A+' can be converted to '(A | (A.A))' for data flow analysis purposes. The control flow graph equivalent to 'A+' is shown in the table 4.1 .

A sequence specification can contain several methods using multiple regular expression operators. The above technique of converting to a control flow graph is applied repeatedly. In a control flow graph, adjacent dummy nodes are reduced to one dummy node. In Table 4-2 the control flow graphs corresponding to different basic regular expressions are shown.

4.4.1.2. Generating traces from the CFG (potential test cases from the CFG)

It is useful to define another concept, namely the set of traces that is defined by the CFG. This set is defined to be the set of all sequences of methods that exist in the CFG. Consider the following regular expression : A.(B|C).D

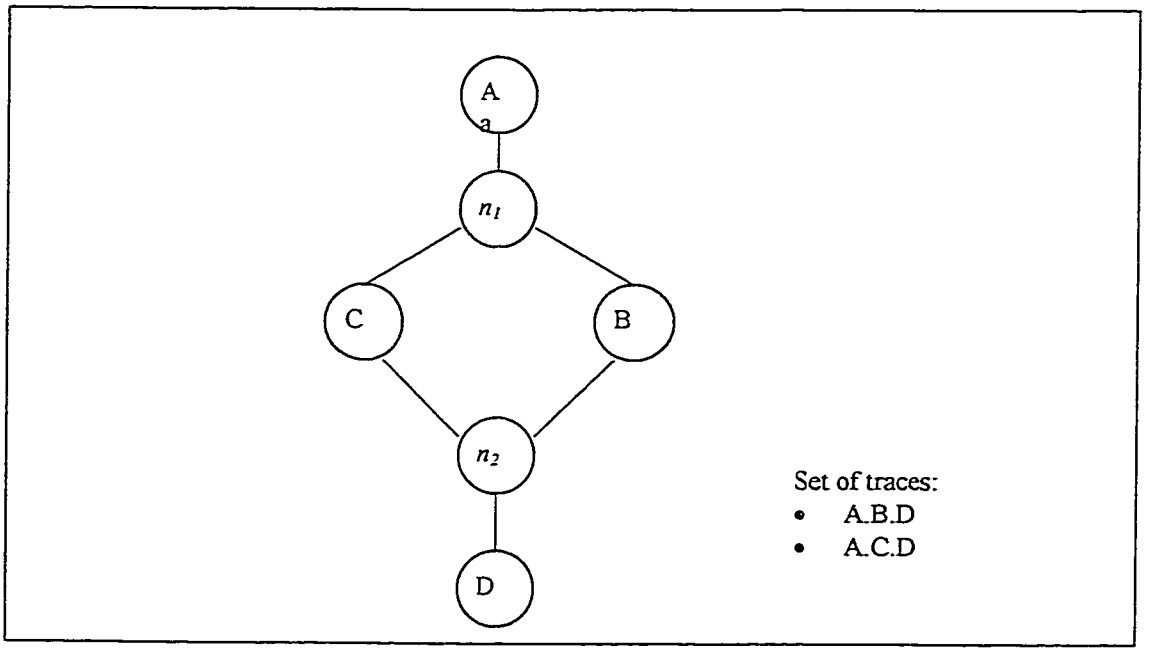


Figure 4-2: Example of generating traces from a CFG

The motivation behind introducing traces is that they represent the possible execution paths that the class or a group of interacting objects can perform and these paths can be mapped easliy in a test case or scenario.

• **Example:**

In this section, we illustrate the construction of a flow graph using the Account class described in the previous sections.

An Account consists of four methods open, close, deposit, withdraw. The expected behavior of the methods is that "open" must be called up before invoking either "deposit" or "withdraw" methods; and "close" method must be invoked last. The MtSS corresponding to this behavior of the Account class is:

$$Methods \Rightarrow open.deposit.(deposit|withdraw)^*.close \quad (1)$$

$$\Rightarrow open.deposit.(1|\ (deposit|withdraw))^2.close \quad (2)$$

$\Rightarrow$ *open.deposit.(1| (deposit|deposit)| (deposit|withdraw))|*

.(|withdraw|deposit).(withdrawt|withdraw).close **(3)**

The equation (1) corresponds to the sequence specification that can be derived from the design or a user specified one. Using Hung's theorem [22], in equation (2), the regular expression '*' operator is reduced. In equation (3) the exponentiation is expanded using the equation $(A\ |B)^2 = (A.A)|\ (A.B)|\ (B\ .\ A)|\ (B.B)$. We use equation (3) for conversion to a control flow graph.

The CFG for the Account class:

The CFG is constructed from the equation (3). We use the same technique as described in the table 4-2. The sequence specification in equation (3) contains only '.' and '|' operators . The '.' operators are converted to direct edges between their preceding and following method nodes. The expression within the huge bracket in equation (3) is concerted to five parallel paths between two dummy nodes n_1 and n_2. The control flow graph corresponding to the equation (3) is shown in the Figure 4-3: CFG of the class Account

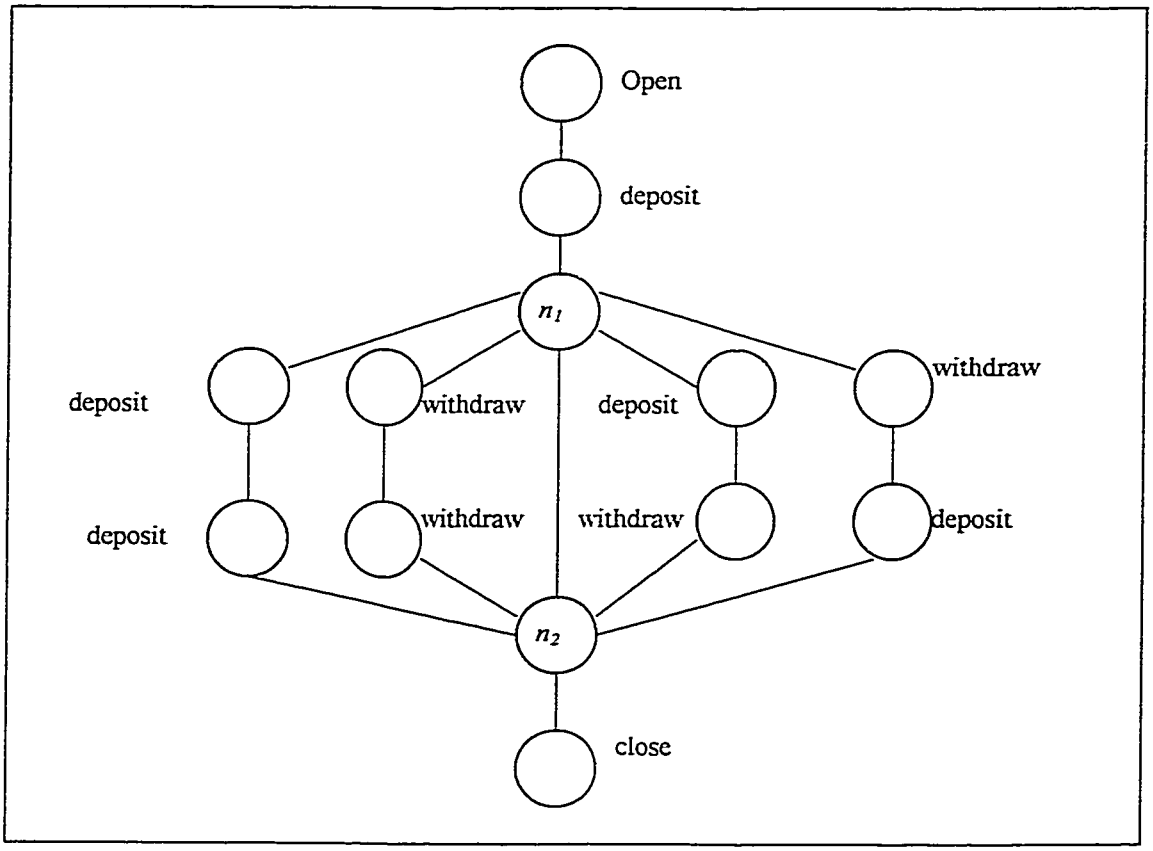


Figure 4-3: CFG of the class Account

The set of traces that can be generated from the above CFG are the following:

P1: open.deposit.close

P2: open.deposit. deposit. deposit .close

P3: open.deposit.deposit.withdraw.close

P4: open.deposit.withdraw.withdraw.close

P5:open.deposit.withdraw. deposit. close

4.4.2. Grey-Box Test case generation from State Transition Diagram (STD)

To generate test cases, it possible to adopt the following coverage criteria:

4.4.2.1. All States coverage

The STD contains many states, and at each state it contains many method invocations. The "all state" coverage method requires at least one method invocation at every state of the class. Test cases can be derived from the STD by selecting the paths that forces the object to go through all possible states. An example of test cases which cover all the states for class Account, is given below:

TestCase1 ⇒ open.deposit.close

TestCase2 ⇒ open.deposit.deposit.withdraw.close

TestCase3 ⇒ open.deposit.withdraw.balance.report.close

4.4.2.2. All methods coverage

An STD for a class contains several states. At each state, many methods can be invoked. A method invocation at the state may result in remaining in the same state. The "all methods" requires all methods of the class to be invoked. In addition the sequence must be compliant with the MtSS of the class. For example, using this method, the following test case can be generated for the class Account :

TestCase1 ⇒ open.deposit.balance.withdraw.report.close

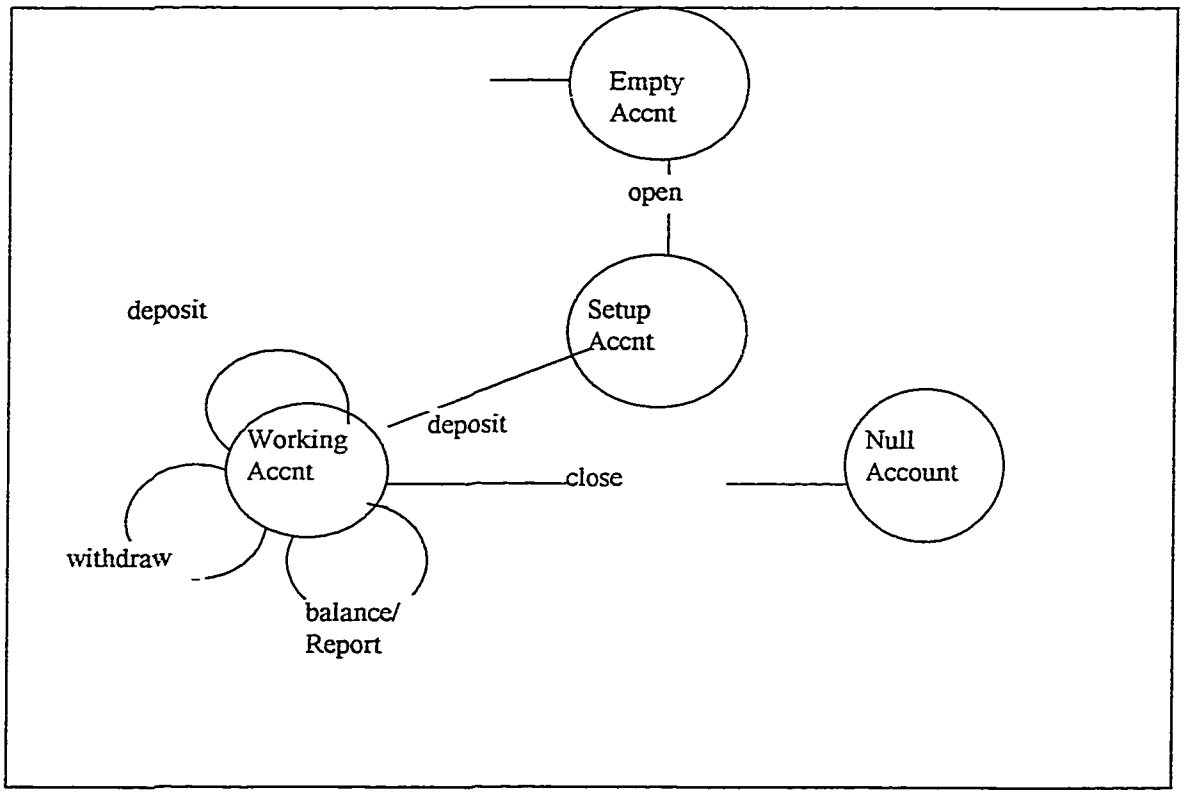


Figure 4-4: Example of generating test cases from a STD

4.4.3. Analysis of the Change impact and selection of Regression Test Cases

As mentioned in previous chapters, one major component of any regression analysis and testing methodology is the reselecting test from an existing test suite cases to retest the system [6] . This activity is concerned with selection of test cases to rerun. It involves the generation of new test cases and the reselection of the old ones. In [6], the old test cases are divided into three classes:

- *Reusable test cases*: those which tested unmodified parts of the specification and their corresponding unmodified parts of the implementation. They remain valid after the modification to the specification and implementation. and do not need to be rerun.
- The test cases relevant to the modified parts (of the specification and the implementation) which we refer to as the affected test cases have two categories: those still valid are called *retestable test cases*, and should be rerun; those which have become irrelevant or out of date due to the changed specification or programs are called *obsolete test cases*. We restrict our analysis in the following sections to the latter category. Our strategy classifies the test cases into two categories: the obsolete scenarios or test cases, and the reusable and retestable test cases.

4.4.3.1. Test case and trace specification notation and example

We define the structure of a test case as a sequence: <method,input,output/state>. The causal order of the methods is important. For example, let t_i be a test case defined to test a class C by executing the following sequence of methods, called $Trace(t_i) = m_0.m_1.m_2.m_3$. Changing the structure of the class C may affect the $Trace(t_i)$, and the test case may become obsolete. In fact the equivalence between a test case and a trace (which means that the test case is still valid and reusable or retestable) can be established if one of the following major changes was not introduced to the trace.

Possible modification introduced to a trace	Original trace	Modified trace	Test case Status
Added new methods	$m_0.m_1.m_2.m_3.$	$m_0.m_1.m_2.m_3.m_4.$	Retestable
Delete d methods	$m_0.m_1.m_2.m_3$	$m_0.m_1.m_2.$	Obsolete
Change s in order and the sequence of methods	$m_0.m_1.m_2.m_3$	$m_0.m_1.m_3.m_2.$	Obsolete
Modification in the internal structure of a method (the method should have the same interface or signature)	$m_0.m_1.m_2.m_3.$	$m_0.m_1.m_2.m_3$	Reusable
Relationship between the input and expected output / state is alternated.	This means that either the input is no longer valid due to the specification changes or the expected output / state should be modified due to these changes		Obsolete

Table 4-5: Modification that can be introduced to method traces.

At this point, we note that we do not consider the data changes that may be introduced either to the class specification or the changes introduced to method interface. This is still an

open problem that future research may address. We mention also that the above changes are considered at an atomic level. However, they may all affect a trace at the same time depending on what is the purpose of the changes. Therefore the equivalence between a method and a trace is established, if they have the same number of methods, the same order and they are synthetically equivalent.

The second step of our methodology detects the major changes introduced to the class specification. Hence, the test case designed to test these classes or class interactions marked as unmodified, need not be rerun. They are classified as reusable. The other test cases are classified as obsolete or retestable depending on the result of the equivalence comparison. In the following we propose a methodology for reselection of affected test cases.

4.4.3.2.Rationalizing Test cases by equivalent Behaviour

Based on our previous definition of regression analysis and testing, one can notice that any changes introduced to the specifications of a class or the class interactions will be reflected in the structure of its corresponding MtSS and MgSS. Hence, to detect if a test case is still reusable, retestable or obsolete, we can verify if it matches a substring (or a sequence) of methods in the language generated from the modified version of the MtSS or the MgSS. This problem is known in the literature as the "regular expression pattern-matching problem". In [66] Leeuwen states the problem as follows:

Given a pattern consisting of regular expression r and an input string $s=s_1\ s_2\ s_3 \ldots\ s_k$ answer " yes" if r matches a substring of s; "no" otherwise, we assume the length of r is m. In our case the MtSS corresponds to the regular expression s, and r is the test case to check. The "no" answer means that the test case does not verify any valid sequence of the modified class and therefore it becomes obsolete and should be removed from the regression test suite. The "Yes" means that the test case is still valid and can be reused to revalidate the class specification.

In [3, 66] efficient algorithms are presented for verifying whether one regular expression is the same as another. Each regular expression defines a Finite Automata (FA). A FA accepts a set of strings (equence of methods in our case) from the language defined by the FA (set of methods of a class). If L_1 and L_2 are the languages represented by two regular expressions, then the two regular expressions are equal if $L_1 = L_2$. One can verify this by constructing a new FA D_1 satisfying $L_1 \oplus L_2$ ($\oplus$ is the exclusive-or operator), then verifying whether D_1 accepts only an empty string. If D_1 represents an empty regular set, then $L_1 = L_2$. Efficient methods and proofs for constructing intersection and complement operators are provided in [3].

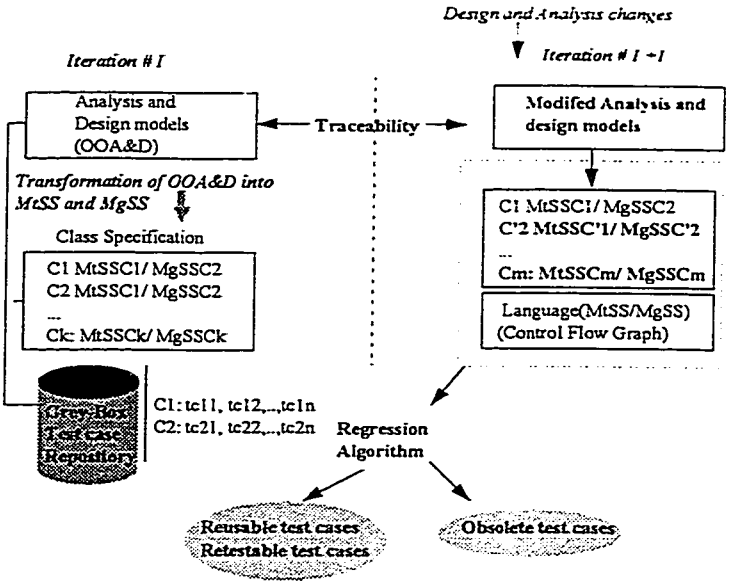


Figure 4-6: Overview of the test case reselection process

4.4.3.3. Algorithm for matching regular expressions

In the following, we describe a Deterministic Finite Automata (DFA) regular expression pattern matching algorithm [66]. The algorithm first constructs a syntax tree for regular expression such is one in (Figure 4-7) for the regular expression (a|b)*aba#. The symbol "#" is an endmarker and a dot "." is the concatenation operator.

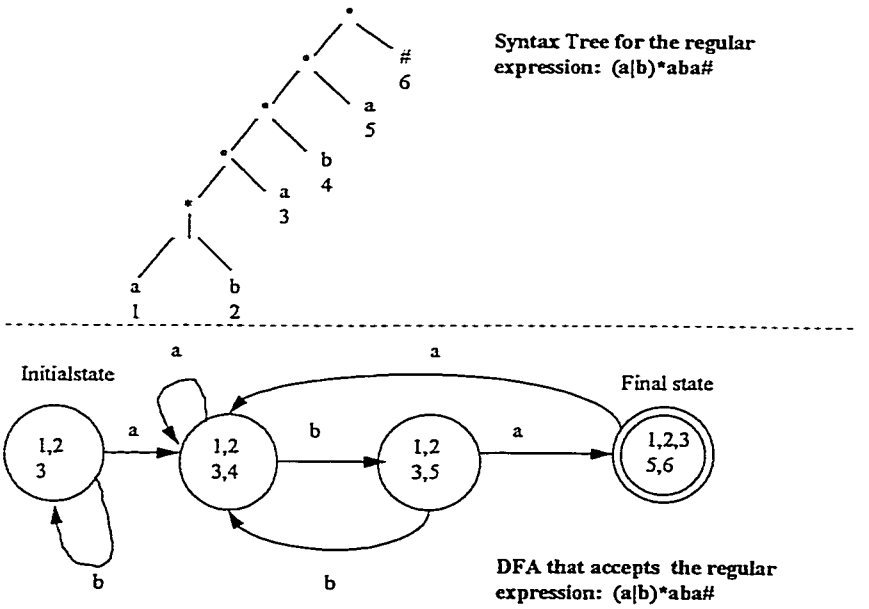


*Figure 4-7: A DFA for the regular expression (a|b)*aba*

The leaves of the tree are labeled by non-metacharacters (such as +, *, ..) defined in the regular expression. Using a technique suggested in [67], each leaf is associated with a unique integer called the *position* of the leaf. Positions are shown below the leaves in the syntax tree of Figure 4-7. The transitions of the DFA are constructed directly from the syntax tree. Each state of the DFA is the set of positions corresponding to the leaves that are active after having read some prefix of the input string. Initially, leaves 1, 2, and 3 are active so the initial state of the DFA is the set {1,2,3}.

The next state *Q'* representing the transition from state *Q* on an input character c is computed as follows: For each transition *i* in *Q* whose leaf symbol matches *c*, we add *follow(i)* to *T*, where *follow(i)* is the set of positions that can "follow" the leaf labeled by *i* in the syntax tree. The following rules define the *follow(i)* function:

- If the left and right leaves are the two children of a node labeled by concatenation operator in the syntax tree and transition *i* is a position that can last be active in the subtree rooted at *left*, then all the positions initially active in the subtree rooted at right are in *follow (i)*.
- If *i* is a position that can last be active in a subtree rooted at a *-node in the syntax tree, then all the positions initially active in the subtree are in *follow (i)*.

For the above example, let us compute the *Q* representing the transition from the initial state {1,2,3} on the input symbol *a*. The leaf corresponding to the position 1 matches *a*, so we add *follow(1)* ={1,2,3} to Q. Position 2 does not match a, but position 3 does so. We add follow(3) ={4} to Q. Thus the transition from state {1,2,3} on *a* is the state {1,2,3,4}.

To complete the DFA, all the transitions should be computed. Any state containing the position corresponding to the endmarker "#" is made an accepting state. Transitions for the symbol "#" are not computed in Figure 4-7.

The transition function can be represented in a 2-dimensional array. A DFA can be simulated efficently by the table look-up algorithm:

Function match_RegularExpression (s,r : regular_expression): boolean;

{ Function returns "True" if the input string s can simulate the DFA generated from the regular expression r, otherwise "False" }

```
Begin
  Q:=initial state
  if (Q is accepting ) then return "True"
  for j:= 1 to n do
    begin
      Q:= goto[Q,sj]
      if (Q is accepting ) then return "True"
    end
  return "False"
End
```

Once a DFA has been constructed from the regular expression, it is possible to determine whether it matches the input string in *O(n)* time. For further discussion about the complexity analysis of the algorithm, please refer to [66].

4.4.3.4. Classification of test cases by regression analysis

The idea behind the regression algorithm is to compare the test cases generated during one iteration for testing one class C, and its interaction and with the corresponding MtSS and MgSS and report if they become invalid reusable or retestable .

In the following, we describe a generic algorithm that can be used to reselect the regression test cases. The algorithm take as input a test suite designed to test a given class C at the unit level and its interactions, and the MtSS and MgSS of a class. It returns the set of obsolete and the set of reusable and retestable test cases.

Reselection of class test cases

algorithm:

Input : Regular expression RE { It can be the MtSS or the MgSS of a class C}

TDB: Test case suite for a class C.

Output: Obsolete_Test : the set of obsolete test cases.

Retestable_test: the test of Reusable and retestable set of test cases.

Begin

-step-1 Construct the DFA corresponding to the RE.

{the DFA is constructed based on the algorithm described in the previous sections}

- step-2 For (Each test case TC from TDB of C) do

if (match_regularexpression(RE,TC)) then {call the procedure to the simulate the test case on the DFA constructed from the RE}

Add TC to Retestable_test.

Else

Add Obsolete_test .

endif

end.

The algorithm is based on the regular expression formalism which is a subject well studied in the literature. We have already described in the previous sections the definitions and the theoretical foundations of our analysis. We can establish the correctness and completeness of this algorithm which basically uses two functions: The first one is to construct the DFA of regular expression and the second one is to simulate the test case on the DFA. The correctness and completeness of the two algorithms are demonstrated in [66].

4.4.3.5.Example

The goal of this example is to show how our approach can detect the set of obsolete test cases after two methods are deleted from a given class.

We assume that we have the following original MtSS of the class Account:

MtSSC1= open.deposit.(deposit|withdraw|balance|report). close.*

We suppose that the following modifications are introduced to the Account class delete the {*balance, report* } methods. The resulting MtSS is:

MtSSC2= open.deposit.(deposit|withdraw). close.*

In section 4.4, the following test cases are generated to test the original version or the class Account:

TestCase1 ⇒ open.deposit.close .

TestCase2 ⇒ open.deposit.deposit.withdraw.close.

TestCase3 ⇒ open.deposit.withdraw.balance.report.close.

In section 4.4.1.2 we generated a subset of the language of *MtSSC2*:

P1: open.deposit.close.

P2: open.deposit. deposit. deposit .close.

P3: open.deposit.deposit.withdraw.close.

P4: open.deposit.withdraw.withdraw.close.

P5:open.deposit.withdraw. deposit. close.

By applying the above regression algorithm, one can easily verify that the following test cases are obsolete:

TestCase3 ⇒ open.deposit.withdraw.balance.report.close.

The following test cases are still valid:

TestCase1 ⇒ open.deposit.close.

TestCase2 ⇒ open.deposit.deposit.withdraw.close.

4.5. Chapter summary

In this chapter, we provided a set of rules that can be used to detect the changes introduced to the class specification. The modifications covered by these rules are adding, deleting methods, modification of the calling sequence of methods at the instance of a class and its interactions with other classes. A regression analysis algorithm for classifying the test cases into obsolete, reusable and retestable test cases is also provided.

5. Reconciliation between the class implementation and its specification

5.1. Introduction

In the O-O software engineering life cycle, design specifications are used in implementing the final code. However, since implementation from design is usually a manual process, a final program might contain inconsistencies and errors. The main purpose of testing activities is to detect and potentially remove such design and implementation defects from the program under development, and to ensure that the system reflects its requirements.

During the development process, it is often the case that changes are introduced directly to the code to modify existing functionality or to add new features. Such changes are not reflected in the software documentation (analysis and design models) and the system specifications which leads, of course, to inconsistency between the implementation and the user requirements.

For O-O software, the class is the basic unit for building programs. We introduce in this chapter a method which reconciles a class implementation affected by code changes and its specifications. The method is based on two concepts: the method and message sequence specification as described in the previous chapter and the concept of class firewall. The class firewall is computed from the code; it describes the set of classes that are possibly affected by a change [62]. Our goals are the following:

- First, reinforce the regression analysis and testing activities. The method checks the correctness and detects the inconsistencies between the class implementation and its specifications.
- Second, help the development team to keep the software models up to date during the migration between successive iterations using the traceability framework.

5.2. The Object Relational Graph (ORG) and the Class firewall

5.2.1. Existing work and discussion

In [26,62], Kung, Gao and Hsai focus on the regression analysis from an implementation perspective. They propose the following two step method:

- Built an object relational graph and class firewall to detect the class that might be affected by code changes.

- Based on those results, they propose a method for selecting the regression analysis and testing package. The selection strategy is designed for C++ implementation and can be used with white-box and black-box test cases.

5.2.2. New results

In this research, we use the same definitions and concepts. However, we add new concepts such as the MtSS and MgSS to capture the class specification. We introduce also two new concepts called the message and method use sequence. Our goal is to detect the discrepancies between the class specification and its implementation. The method can be summarized in the following steps:

- Generate the MtSS and MgSS for the different classes of the system.

- Built an object relational graph and class firewall to detect the classes that might be affected by code changes.

- Control and data flow methods [65,12] are used to perform static and dynamic analysis of the code. The results of the analysis are called method and message use sequences (see section 5.4).

- Compare the sequences generated from the implementation with those generated from the specification, and report the differences.

5.2.3. Limitations and assumptions:

In the following section, the limitations of the proposed reconciliation method, and a few assumptions are discussed:

- The implementation of classes may contain more detail than the design models. Usually designers and developers add new methods and data when migrating from the design models to the code for different reasons, such as the implementation environment, or to comply to certain performance constraints (e.g. predefined functions for the compiler). Although those modifications are not the focus of our reconciliation analysis, they can be detected and reported as changes introduced to the class definition.

- The reconciliation between the class specification and its implementation assumes that coding and the design are done in a development framework with no code generation capabilities. In practice, this is usually the normal scenario for software development where designers use modeling tools (e.g. OMT) to produce the analysis and design models, and a configuration management tool linked to a specific programming language to produce the code.

- Data and control flow analysis for O-O system are code dependent. They are not the focus of this research. Future investigation and research is needed in this area.

5.2.4. The Object Relational Graph [62]:

The ORG is introduced to represent the inheritance, aggregation, and association relationships between classes [62] (see Figure 5-1). It is used to identify the classes affected when one or more classes are changed. Before we proceed to describe the ORG, we first briefly review the inheritance, aggregation and association concepts used in O-O modeling. In O-O programs, there are three different relationships between classes. They are inheritance, aggregation, and association. In O-O programming languages (like C++ or Smalltalk), the inheritance feature is provided to support the generalization and specialization concepts in order to encourage code reuse in the implementation. An inheritance relation between a class and its subclasses means the properties defined for an object class are automatically defined for all its

subclasses (unless selective and/or overriding inheritance are specified). Aggregation is supported in O-O programming languages through encapsulation, class methods, and instance variables. Using the aggregation concept, a composite object can be defined based on its component objects. We call a composite object an aggregate class object. The relation between an aggregate object and its component object class is called an aggregation relationship. An association relationship means that two independent object classes associate with each other in some manner. The associations includes data or control dependence, or message passing between two independent object classes.

Definition 4.1:

An edge-labeled digraph $G=(V,L,E)$ is a directed graph where $V=\{V_1,..., V_n\}$ is finite set of nodes, $L=\{L_1,..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is a set of labeled edges.

Definition 4.2:

The ORG for an O-O program P is an edge-labeled digraph $G=(V,L,E)$ where V is the set of nodes representing the object classes in P, $\{I,Ag,As\}$ is the set of edge labels, and $E \subseteq E_I \times E_{AG} \times E_{AS}$ is the set of edges defined below.

Definition 4.3:

$E_I \subseteq V \times V \times L$ is the set of direct edges representing the inheritance relation between the classes. For any two classes C_1, $C_2 \in V$, $<C_1,C_2,I>. \in E_I$ indicates that C_2 is derived from the class C_1 .

Examples: In C++ programs, the inheritance relationship are identified by declarations of the following forms (they are defined in the header files):

class *Class_A*: *Class_B*

class *Class_A*: public *Class_B*

class *Class_A*: protected *Class_B*

class *Class_A*: private *Class_B*

Definition 4.4:

$E_{AG} \subseteq V \times V \times L$ is the set of direct edges representing the aggregation relationship between the classes. For any two classes C_1, $C_2 \in V$, $<C_1,C_2,Ag>. \in E_{AG}$ indicates that C_1 contains one or more objects of class C_2. .

Examples: In C++ programs, there are three types of aggregations:

- Automatic aggregation is identified by declarations of the following forms:

(1) class *Class_A*: {

Class_B b;

Class_C c[m];

//...};

(2) class *Class_A*:: f (...) {

class *Class_B b;*

Class_C c[m];

//...};

- Static aggregation are identified according to the declarations of the following forms:

(1) class *Class_A*: {

```
        static Class_B b;
        static Class_C c[m];
//...};
(2) class Class_A:: f (...) {
    static class Class_B b;
    static class Class_C c[m]
     //...};
```

- Dynamic aggregation is identified by the declarations of the following forms:

```
(1) class Class_A: {
       static Class_B *b;
      static Class_C *c[m];
     //
     Class_A () ; // constructor.
    };
 Class_A:: Class_A () {
        // Dynamically create b;
        //  Dynamically create c[m];
      .};
```

Definition 4.5:

$E_{AS} \subseteq V \times V \times L$ is the set of direct edges representing the aggregation relation between the classes. For any two classes C_1, $C_2 \in V$, $<C_1,C_2,As>. \in E_{AS}$ indicates that C_1 is associated with the class C_2. in the following three ways:

- *Class C1 uses instance variables of class C2. This called data dependence.*
- *A Class C2 member function is invoked by some member function of class C1. This called* ***message passing.***
- *Class C2's objects are defined as formal parameters of member functions in class C1.* ***This called object parameter passing.***

Examples: In C++ programs, there are three types of associations:

- Friend member function associations. These are identified by the declarations of the following forms:

```
(1) class Class_A: {
       //...
    friend return_type Class_B::f ( ..) ;
  //...};
(2) class Class_A:: f (...) {
    class Class_B b;
     Class_C c[m];
```

//...};

- Friend class associations are identified by declarations of the following forms:

(1) class *Class_A*: {

//....

friend class *Class_B;*

//...};

- Ordinary associations are established through parameter passing of an instance of one class (or a pointer of another object) to a member function of another class.

5.2.5. The Class firewall

Any implementation of a class can be changed in many ways. Here, we classify various changes into three types, as described in [23,25,62]:

- (1) They affect its behaviors, such as states and transitions.
- (2) They affect operations and the behavior of its members functions.
- (3) They affect the relationships between classes and others.

In the following we state a set lemmas and theorems to define in a formal way the concept of class firewall, for a detailed description of the proofs please refer to [62].

Lemma 4.1:

Let a class A be a subclass of class B in the inheritance hierarchy , and class B is changed without affecting its relationship with other classes. If the changes affects the inherited members (from a class B) of class A, for adequate testing, not only class B should be unit tested but class A should also be retested and reintegrated with class B.

Lemma 4.2:

Let a class A be in aggregation relationship with a class B, and only B is changed, without affecting its relationship with other classes. For adequate testing not only should class B be unit retested, but class A should be retested with class B.

Lemma 4.3:

Let A and B be a two independent classes, and class A is associated to class B, and if class B is changed without affecting its relationship with other classes then for adequate testing, not only class B should be unit tested but class A should also be retested and reintegrated with class B.

5.2.6. Constructing a class firewall:

To compute the class firewall, we first introduce a binary relation R that is derived from the direct edges of an $ORG=(V,L,E)$:

$$R=\{ <C_1,C_2> \mid C_1,C_2 \in V \wedge (\exists\, l)\ (l \in L \wedge <C_1,C_2,l> \in E)\}$$

We call R the dependence relation because it defines the dependence between the classes, according to the inheritance, aggregation, and association relations. More specifically , $<C_2,C_1> \in R$ if one of the following cases holds:

- (1) C_1 is derived from the class C_2.
- (2) C_1 is an aggregate class of C_2.

- (3) C_1 is associated with class C_2 by either accessing its data members or passing some messages.

In all of these cases, C_1 is dependent on C_2 in the sense any code change to C_2 would affect the behavior of C_1. The computed class firewall for a class C, denoted CCFW(C), then is defined as :

$$CCFW(C)=\{ C_j | <C,C_j> \in R^*)\}$$

where R^* the transitive closure of R. that if $<C_i,C_j> \in R$ and $<C_j,C_k> \in R$, *then* $<C_i,C_k> \in R$, the transitive closure of R can be computed by the algorithm proposed in [64].

Theorem 4.1:

Let G be an ORG for a given O-O program P, and R be the dependence relation derived from G. Let C be a class in which a change is made to its defined or redefined members. Assume the dependencies between the classes of P are dependencies of inheritance aggregation, and/or association relation, then CCFW(C)= CFW(C), that is :

1. CCFW(C) $\subseteq$ *CFW(C).*

2. CFW(C) $\subseteq$ *CCFW(C).*

Theorem 4.2:

Let CCFW(C) be the computed firewall for class C. For any class C_i, *if* C_i *is not in CCFW(C), then* C_i *is not affected by a change in class C and hence no retest is needed.*

The notion of class firewalls can be extended to a set of changed classes.

Let S= $\{C_1,C_2,C_3,...,C_k\}$ be a set of classes (that are changed), then the class firewall for S denoted CFW(S) is defined as follows:

$$CFW(S) = \bigcup_{\forall C \in S} CFW(C)$$

Example:

Figure 5-1 shows an ORG of the elevator example [25].

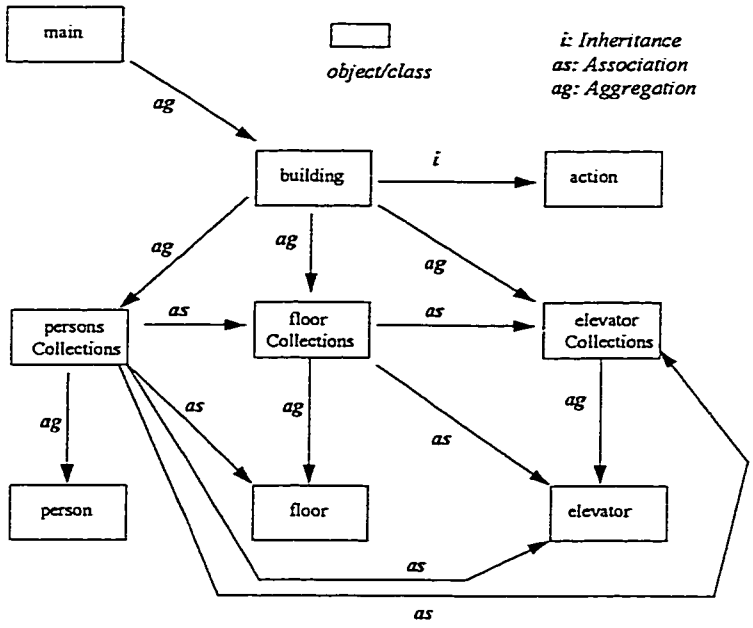


Figure 5-1: ORG of the elevator example

The ORG is constructed based on the implementation of the different classes. For example the declaration "*Class building: public action*" is represented by the inheritance edge from building to action.

The following declaration are represented by the aggregation edges that connect *building* to the *persCollection*, *floorCollection*, and *elevCollection*. The aggregation edges are one-to-one which means that each building has exactly one *persCollection*, *floorCollection*, and *elevCollection* instance.

```
class building : public action {
private :
   persCollection  thePersons;
    floorCollection theFloors;
   elevCollection theElevators;
public:
//...
}
```

5.3. Verification of the code changes and the class specification

5.3.1. Use sequence definition

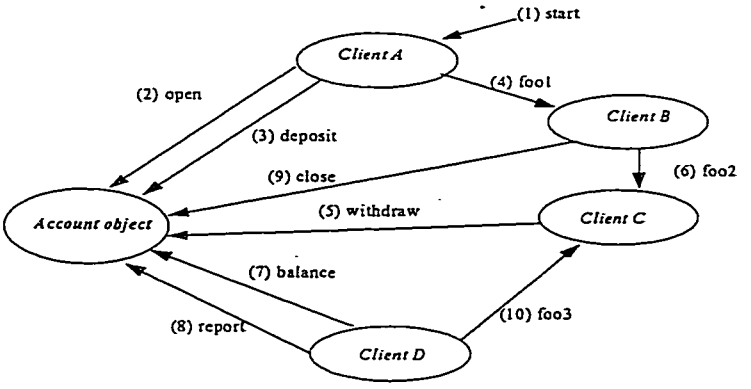
The MtSS of a class specifies the correct order in which messages can be sent to instances of the class. Similarly, the MgSS specifies the causal order in which messages can be sent by each method to different class instance. Even though a class supports many methods, in practice, only subset of the methods might be invoked in a class instances. The sequence in which methods are invoked in an instance object, or sent to other collaborating objects, are called *use sequences* with respect to that object. The concept of use sequences is very similar to the concept of MtSS and MgSS. We call a Method Use Sequence denoted as MtUS as the sequence of methods invoked at an instance object, and we call a Message Use Sequence, denoted as MgUS, as the sequence of messages sent out by an object to other instance objects.

For each modified class, we compute its corresponding class firewall. We define the use sequence for the classes included in the class firewall. They are determined by analyzing the code using static or dynamic control flow analysis [29,35]. Then, the use sequences can be compared against the MtSS and the MgSS of the corresponding classes to determine the ripple effect of changes and the inconsistencies between the class specification and the modified implementation. In this chapter we stress the regression analysis aspects, rather than the control flow analysis.

Example:

Consider the MtSS of the class Account: *(open.deposit.(deposit|withdraw)*.close)*

Figure 5-2 depicts an Account object with four other client objects sending messages



Generating the method use sequence based on control flow analysis

Figure 5-2: Example of generating the MtUS based on control flow analysis

The number on are arc corresponds to the sequence in which the methods are expected to execute. Control flow analysis can be used to generate such use sequences:

For this particular example, the following sequence is generated:

MtUS-1: *open.deposit.withdraw.balance.report.close.*

The algorithm described in [66] can be used to compare the generated use sequence with the original MtSS. One can notice that two methods {*balance, report*} are added. Hence the tester can find the gap between the class specification and the modified implementation.

5.4. A conceptual model for a run time verification system

The static control flow analysis of an O-O program may not be able to identify all the possible use sequences due to the presence of dynamic binding, polymorphism, dynamic typing and pointer data structures [1]. For dynamically typed languages such as Smalltalk, it may be impossible to derive all the use sequences. Here, we propose a conceptual model of a run time verification system that can help the development team identify any inconsistency between the modified class implementation and its specification. Our approach consists of the following steps:

- First, identify the classes affected by changes. This is done by computing the class firewall from the code using the algorithm described in [62].
- Second, perform run time regression analysis for each class defined in the class firewall. During execution, each object receives a set of messages from client objects and as a response to these stimuli it executes corresponding methods. The method-message binding can take place at run time, therefore at each object one can monitor the set of messages the object receives and the sequence in which it invokes its methods. Also, it possible to control at each object, the set of messages that are sent by each method to other instances of collaborating objects (see Figure 5-4).
- Third, compare the method and message use sequences generated for each class affected by the code changes with the corresponding method and message specification sequences for the same class, and report the differences.

5.4.1. Functional description of the run time verification system

The run time verification system monitors all methods invoked at an instance object to construct the MtUSs. It then compares them with the corresponding method sequence specifications (MtSSs). If the methods invoked are not compliant with the specifications, then the system will report the inconsistency between the implementation and the specification. In a similar way, the run time verification system will construct the MgUS of each object and will check the modified class interactions with its corresponding MgSS. We can use the same algorithm described in [3] to compare the use sequences generated from code with the MtSS and MgSS of the corresponding class and report the differences. The system will flag modifications in the internal structure of class with added new methods, deleted methods or changes in the calling sequence of methods. It can also detect the modification of interactions between classes (added interactions, deleted interactions).

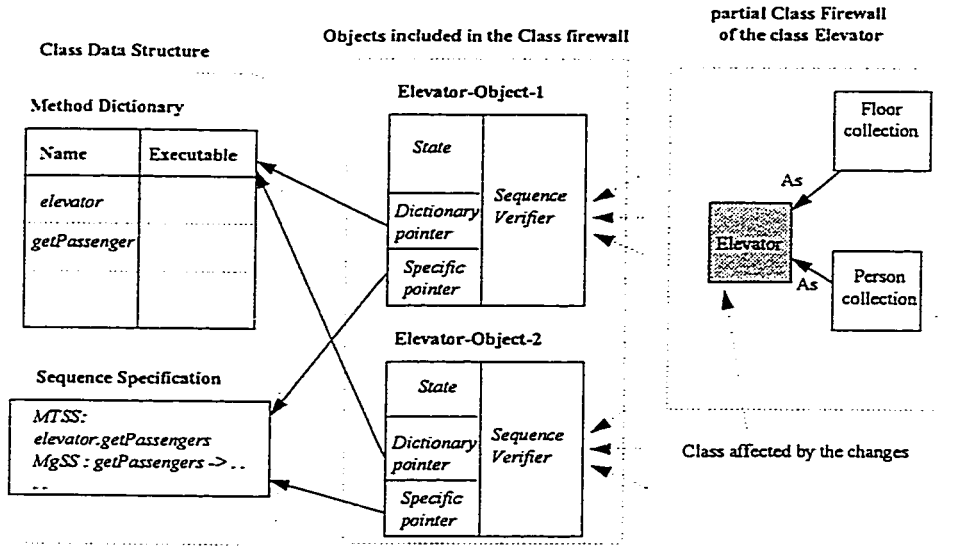


Figure 5-3:Data structure diagram for implementing the run-time verification system

Figure 5-3 depicts a data structure model that can be used in implementing such a runtime verification system. Each class computed in the firewall will maintain links with its objects when they are created at run time. The class maintains the method dictionary for all its instances. The method dictionary contains the name of the method and the address of the executable binary code. Thus, all the objects of a class share the method dictionary. The MtSS and MgSS of a class will be included with the method dictionary so that all the instances of the class can have access to the same MtSS and MgSS. Each object instantiation stores space for instance variables, a pointer to the method dictionary, a pointer to the MtSS, and a pointer to the MgSS. At each object, a method specification verifier and a message sequence verifier is kept. They keep track of the messages received and sent by the instance object and store them in the internal object state a method and message use sequences. To detect the inconsistency between the implementation and the class specifications, one can compare the use sequences generated with MtSS and MgSS of the classes affected by the changes.

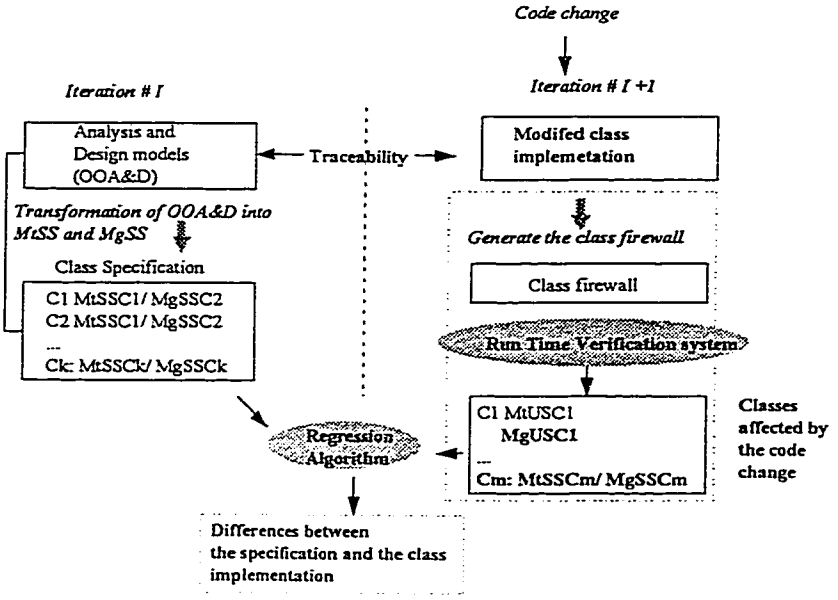


Figure 5-4: Reconcilation between the class implementation affected by the changes and its specification

The above analysis can be summarized by the following algorithm:

Input : Let $S=\{C_1, C_2, ..., C_k\}$ the set of changed classes;

Step-1: Compute the class firewall CFW(S) to detect the set of affected classes;

Step-2: For each C_i in CFW(S);

Begin

Construct Method Use Sequences $MtUS(C_i)$;

Construct Message Use Sequences $MgUS(C_i)$;

Compare(MtSS,MgUS) and report modifications;

Compare(MgSS,MgUS) and report modifications;

End.

Example:

Suppose we are given the ORG of the class *elevator*. Based on the class firewall concept [62], if the class *elevator* is changed then the following classes are also affected: *persCollection*, *floorCollection*, *elevCollection*, *floor* and *person*. They need to be retested.

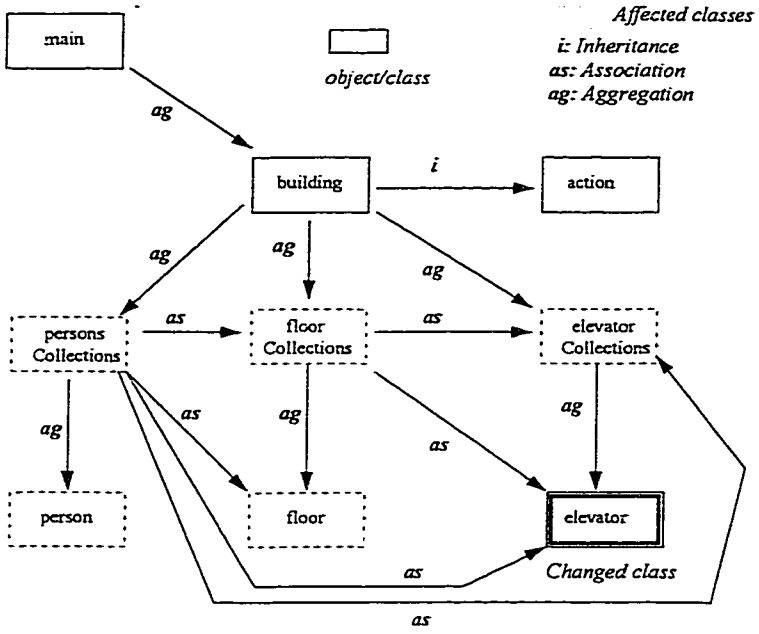


Figure 5-5: Example of the ripple effects of changing the Elevator class

If we consider the following C++ implementation of the class *elevator*:

```
class elevator {
//...
public:
elevator();
int getPassengers(void) { return passenger;}
void setelevNumber(int n);
void showElevator(void);
void setDirection(floorCollection &theFloors);
int elevStopping(floorCollection &theFloors);
void action(floorCollection &theFloors, persCollection &thePersons);
};
```

The following use sequences can be generated by the run time verification system:

MtUS-1: elevator.getpassengers.~elevator.

MtUS-2: elevator.action.setelevNum.setDirection.elevStopping.~elevator.

MtUS-3: elevator.action.setDirection.elevStopping.~elevator.

A simple comparison of the above sequence of methods with the MtSS defined for the elevator class using our regression analysis algorithm can detect added methods, deleted methods and modification of the sequence methods.

5.5. Chapter summary

In this chapter, we described a method for reconciliation between the class specification and its implementation affected by code changes. The method is based on the concept of MtSS and MgSS discussed in previous chapters, and the concept of class firewall for detecting code changes. The purpose of the method is to detect the inconsistencies between the implementation and the design, and therefore supports the validation and verification activities during an iterative incremental development process.

6. Traceability and O-O Regression analysis and testing

6.1. Introduction and motivations

Traceability is a technique used to link together the user and system requirements, the analysis and design models, and the implementation and the test cases. It is designed to solve what we call the " Requirement Traceability problem", that is, the ability to describe and follow the life cycle of the software requirements in both forward and backward directions [47]. According to [65], a requirement is traceable if the origin of the requirement is clear and it facilitates referencing in future development activities. Each requirement should have an explicit reference to its source in previous documents if any source information is available, and each requirement should have a unique reference identifier so it can be clearly referred to in future development activities.

Requirements traceability is considered as an audit trail that can verify that subsequent work products satisfy the given products requirement. The main objectives of traceability are:

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

Based on the objectives stated above, the following criteria have been established for mechanisms and methodologies used to implement a traceability procedure [65]:

- The mechanism should provide features so that the requirements in a document can be clearly stated, indexed, segmented and cross referenced to improve requirement completeness and stability.
- Requirements can be traced forward and backward.
- The validation and verification process is based on the indexed, enumerated requirements. The corresponding mapping between tests and requirements can be explicitly listed.
- The mechanism is easy to learn and use.

- Modifications introduced to requirements, or other software analysis and design models, can be traced and documented effectively.
- The mechanism is tightly coupled with the documentation development process.
- The mechanism produces requirement information in a format which can facilitate testing/verification and problem evaluation and analysis.

Numerous technique have been used to solve the requirement traceability problem including cross referencing schemes [52], keyphrase dependencies [53], templates [54], requirement traceability matrices [55], matrix sequence [56], hypertext [57], integration documents [58], and matrix assumption-based truth maintenance networks [59]. Many commercial tools already exists [16]. They use similar concepts, and differ mainly in cosmetics, and the time and the effort and the manual intervention needed to achieve the requirement traceability. Many of these tools are inadequate for the O-O development process; they do not consider particulars of the O-O paradigm and the iterative nature of the development process [16].

In our proposal, we use traceability as a framework for conducting and managing the O-O regression analysis and testing activities. Our claim is that to support the latter activity, it is important to provide information about the scope of changes to maintain the test data. In other words, traceability, with its notions of nodes, analysis, design and implementation models , documents and links [44], allows the user to identify elements in the software models in order to establish relationships between these elements, and to navigate and report this information. Furthermore, by using traceability to maintain these connections, the development team can quickly assess the impact of the proposed change across all the views of the software provided by the models. They can also use these traceability links to assist in creating particular kinds of models. Any relationship to existing models acts as an integrity check to identify missing or extraneous elements. For example, the development team, in their review of a newly created design model, can verify that every requirement has been addressed by at least one design element. It is important to note that although the establishment of links between the models is crucial to developing high quality software, without some form of automation, performance of this task is laborious, costly and error-prone.

In this chapter we describe a conceptual model of the TOOQE methodology that supports the regression analysis and testing activities as identified in the previous chapters. The framework will include our proposal for solving the reselection of test case problems based on the class specification and assessing the impact of changes from the code and use case perspectives. These latter activities should be supported by a set of tools, and a TOOQE model should be used in a central, common area allowing these tools to communicate with each other, and should be integrated into a maintenance environment.

6.2. Traceability in the TOOQE methodology

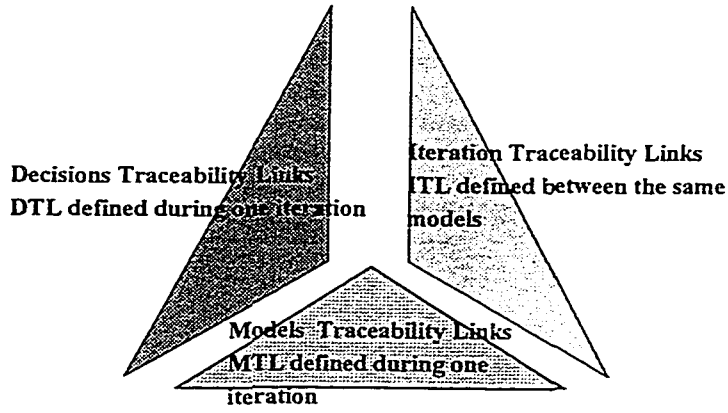


Figure 6-1: Traceability facets

Traceability in our model has three facets (See Figure 6-1):

- Traceability among all artifacts produced during one iteration of the development process. This facet links together the user requirements, the analysis models, the design models, code and test cases. This will provide forward and backward navigation mechanisms and change impact analysis capabilities.

- Traceability among the different iterations which link different versions of one model through the whole process of software development. This will provide mechanisms to track the full history and the evolution of an artifact and its links.

- Traceability among the main change decisions and the software models to be modified. Change decisions are new models that we define in order to document the major reasons for modifications. The changes are usually requested by the user or due to implementation or performance constraints. Such models will hold the following information: For example, the models to be updated, date and purpose of the change, and the designer responsible of implementing the modification. The traceability mechanism operates by constructing what we call a model dependency descriptor [63]. The links will be used to keep track of all change decisions, the models to be modified and their evolution. It provides navigation mechanisms between the software models and the change decisions through the different iterations. For example, after assessing a new release of the software system at the end of one iteration the user requests to split one use case into two new ones. This request may cause major modifications in other analysis, design and implementation models. We define a decision request model to keep track of such major modifications. Change decisions may be in waiting, active, or old status. A waiting change decision exists but it is isolated or partially linked. The waiting status represents an incomplete specification during constructive design or modification. After all links have been added, the decision becomes active by explicit command and the links are consolidated. In active status no modification can be made to the model dependency descriptor. The active status represents a stable situation where decision is used to support program comprehensive and impact analysis. Active decisions become old at the end of iterations and the software assessment phase. They are kept for documentation and maintenance purposes. The concept of traceability is used here as a means of capturing change rationales.

6.3. Models used in the TOOQE methodology

A TOOQE repository should be populated with data associated with the regression analysis and testings namely analysis models, design models, the class implementation and the test cases. These models are expressed using a certain formalism depending on the O-O methodology adopted during the development phase. Hence we define what we call the "TOOQE templates", which contain the useful information and incorporate additional ones needed for traceability purposes. The structure of the template depends mainly on the model that we are going to abstract. The models used in our framework are the following [69]:

- requirements.
- use cases .
- analysis object specification or CRC cards.
- interaction diagrams.
- state transition diagrams.
- class specification: contains an abstract description of the methods, variables and collaboration between classes. The MtSS and MgSS are defined and stored within class specification model.
- code.
- test cases.
- decision change request.

The motivation behind using the above models is that they are common to almost all O-O methodologies. They summarize any information provided in additional models expressed in different formalisms [11,18,19]. They cover the structural, dynamic and functional views of the system under development. Also, they are used and generated by many commercial O-O case tools, which will facilitate the task of gathering the information by a traceability tool automatically. The last template called decision change request is added to the above list to document any request for changes. It will keep track basically of the models to be updated and the purpose of the change.

At this point, we introduce a new concept called "anchors" which will identify the atomic elements of a given model. An anchor is point where a traceability link could be established. For example a responsibility for an analysis specification object is an anchor that could be linked to a use case. The main reason of such decomposition is to reinforce the ease of verification and validation activities. Anchors may be nested within each other as long as they do not overlap which means a model can be considered as an anchor which contains other basic anchors. For example, an interaction diagram contains consist of objects and messages. We identify the anchors using a label which is a unique identifier of the anchor; a type which will refer to the model being decomposed and a description or annotation. For each model (analysis, design implementation and change decision), a set of generic anchors can be defined based on its type and its basic components. A detailed description of fields defined in each model and the implementation constraints will be provided in Chapter 7.

6.4. Traceability links

In order to provide traceability between the models, we define links which allow the connection between two or more models (anchors). We define a link as a mutual inverse association (a bi-directional link) that can be traversed from an original model (anchor) to a destination model. We describe a link by the following set of attributes (see Table 6-1, page 1):

Attribute	Description
A link label	Used to identify the links between the models
Link description	Annotation of the link
History information	Date and time of creation/ responsible person etc..
Version	Number iteration
Original model type	Type of the origin model (e.g., analysis, design, implementation)
Destination model type	Type of the target model (e.g. analysis, design, implementation)
Link types	Iteration/Decision/ Model link

Table 6-1: Traceability link attributes

We identify three categories of links (see Figure 5-2) :

• *Model Traceability Links (MTL):* They link the analysis, design, implementation models, and test cases during the same iteration. They provide the mechanism for navigation forward and backward between the models in order to perform the regression analysis and testing activities.

Origin model type	Destination model type
Analysis	Analysis
Analysis	Design
Design	Design
Design	Code
Analysis	Test suite
Design	Test suite
Analysis	Code
Design	Code
Code	Test suite

Table 6-2: Model traceability links

• *The Iteration Traceability Links (ITL):* They link different versions of the same model between different iterations.

Origin Model	Destination model
Analysis	Analysis
Design	Design
Code	Code

Table 6-3: Iteration Traceability Links

• *Decision Change Links (DTL):* They link the different change request decisions with the models to be updated during the different iterations of the development process.

Origin Model	Destination model
Decision	Analysis
Decision	Design
Decision	Code

Decision	Test suite

Table 6-4: Decision Traceability Links

Figure 6-2 shows the different types of traceability links used in our regression methods.

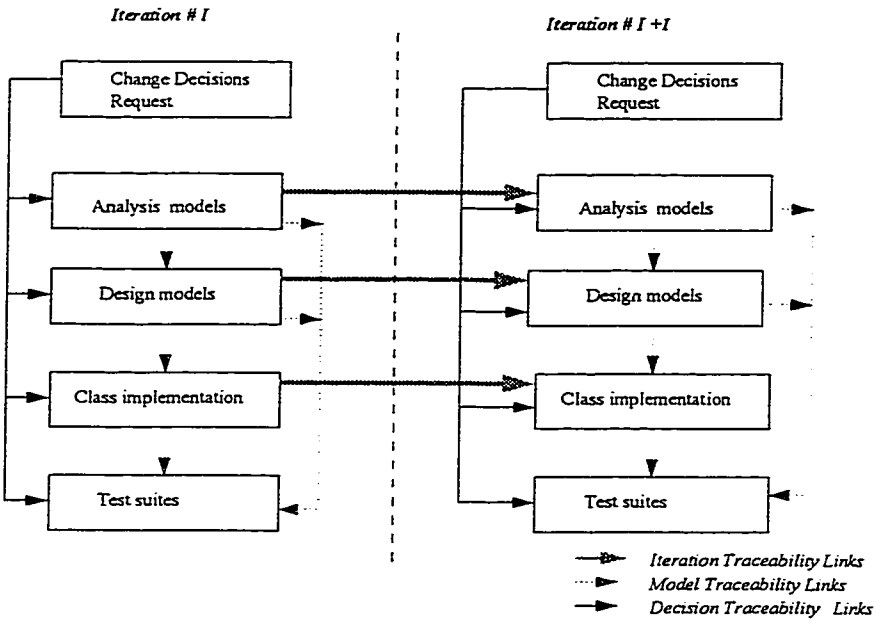


Figure 6-2: Traceability links in the TOOQE methodology

6.5. Change Analysis

In previous chapters, we identified changes that can be introduced to the analysis, design and implementation models. Here we describe how we can use traceability to detect such changes, and how we can track and assess the ripple effects of modifications. Our analysis takes consideration the changes introduced during the migration from one iteration into the next one (see Table 4-1, page 42). We classify them into two categories:

- *Forward changes:* We have already discussed this type of change in Chapter 4. We emphasize that they can be introduced from a use case perspective and propagates to the analysis, design and the implementation models. This is the normal scenario of software development and it is usually introduced during the migration between iterations.

- *Backward changes:* We have described this type of changes in Chapter 4 and Chapter 5. These changes are introduced during the same iteration (e.g., fixing bugs) or between successive iterations (e.g., adding new methods to a class). Here we describe how traceability helps in keeping up to date the different analysis and design models and allows the user to perform change impact analysis by comparing the class specification and its implementation.

6.5.1. Forward changes

Changes are documented in the change request model. The development team establish the decision links (DTLs) with the models to be updated. For example, if the user requests the modification of a use case, a new version will be created and all the model traceability links tied to the original version will be reported to be insecure. These MTLs will be used to navigate and update the models. New models and links will be created; others will simply be deleted or modified depending on the scope of the change. The resulting relationship between the models should be complete, consistent and correct. Figure 6-3, shows an example of propagation of changing a use case:

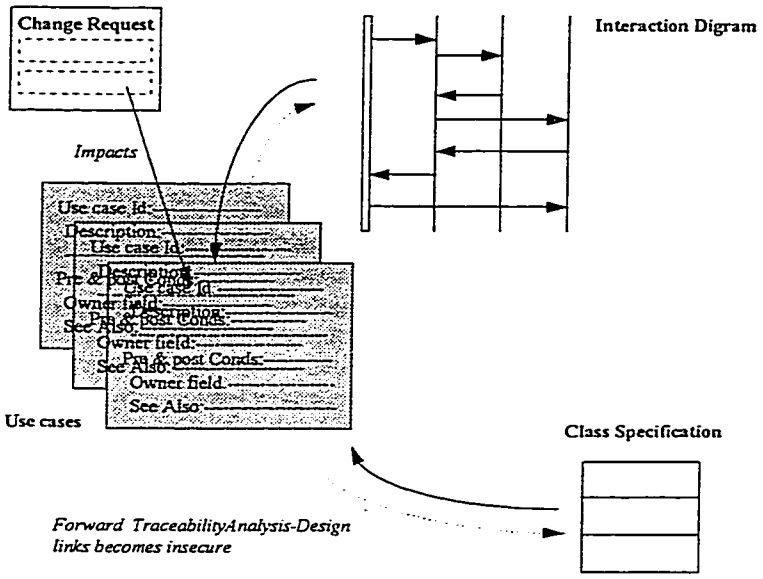


Figure 6-3: Example of propagation of changes after modifying a use case

Previously, we introduced the concept of anchors (section 6.3). Here we point out that they provide a basic mechanism for detecting the changes introduced to the internal structure of one model. In fact a model link consists of a set of sub-links depending on the number of anchors defined within it. In table 6-5, we give an example of the composition of MTLs.

Model type	Origin Anchor type	Destination Anchor type	Label
Requirement	Requirement	Use case	Req_UC
Analysis specification object	Responsibility Responsibility Responsibility Analysis Object Specification	Use case Interaction diagram Class specification Class specification	Res_UC Res_ID Res_CL Obj_CL
Class specification	Method Instance variable Method	Responsibility Responsibility method implementation Variable	Met_RE Var_RE Met_ME

	Instance variable	implementation	Var_ME
Interaction diagrams	Use case Method	Interaction diagram Class specification	Use_ID Met_CL
Test cases	State Transition Diagram Use case Class implementation Class specification	State based test case BB test case WB test case GB based test case	Stc_SD Btc_UC Wtc_CL Gtc_CL

Table 6-5: Example of Model Traceability links

In the following, sections we will describe first how we can use the model traceability links to perform validation and verification activities at the end of one iteration. Second, we show how the iteration traceability links could be used to analyze the class specification modification.

6.5.1.1. Model Traceability Graph and change detection:

Figure 6-4 depicts the main models' traceability links among the TOOQE templates. We assume at this point that during the IIDP, changes can be introduced to an analysis, or design model or implementation independently. The traceability links are transitive. For example if a use case is modified, the analysis object specification will be marked as affected model but the class specification, which is linked not to the use case but to the Analysis Object Specification will also be marked. Moreover, there is an implicit order of propagation of changes (see Table 6-6) between the models [7].

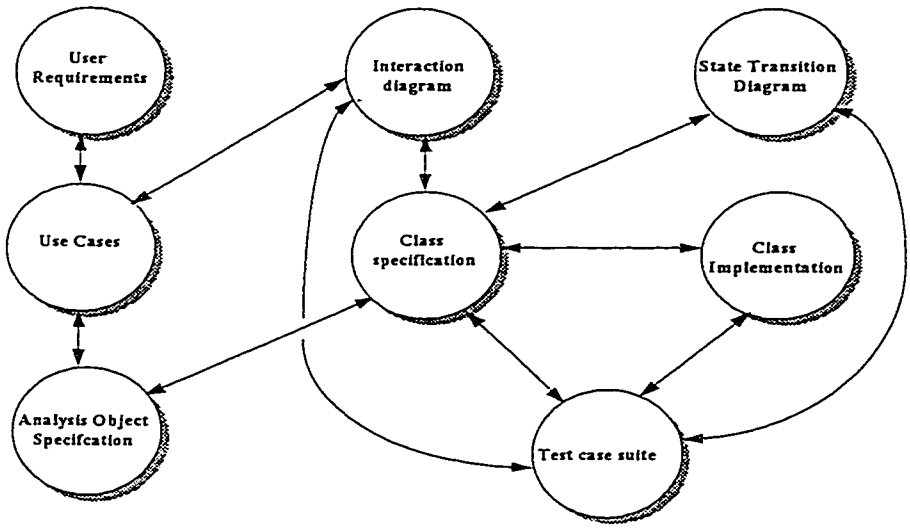


Figure 6-4: The Model Links between the TOOQE templates

In a formal way, the relationship between the models can be described by a Model Traceability Graph (MTG). AMTG is an edge labeled directed graph *MTG=(V,L,E)* where V is the set of nodes representing the different analysis and design models (e.g., use cases, ID). L={DecL, AnaL, DesL} is the set of edge labels (for decision link, analysis link, design link), and $E = E_{DecL} \cup E_{AnaL} \cup E_{DesL}$ is the set of the traceability links between the models. Modification to

[7] We need an acyclic graph to propagate the changes.

the MTG can be classified into three basic cases: adding a model, deleting a model, and changing a model. These structure changes can be identified as follows:

Let *MTG=(V,L,E) and MTG'=(V',L',E') be the MTG for two versions of the same software generated in two different iterations. A structure change in the MTG is:*

- If $V' - V \neq \varnothing$ then any $v \in (V' - V)$ is an added model.
- If $V - V' \neq \varnothing$ then any $v \in (V' - V)$ is a deleted model.
- If $E - E' \neq \varnothing$ then any $e \in (E' - E)$ is an added edge.
- If $E' - E \neq \varnothing$ then any $e \in (E - E')$ is a deleted edge.
- If any $v \in V \cap V'$ is changed, then a residual model is changed.

In the following section, we give examples of the possible changes that may be introduced to the analysis and design models. We illustrate also the propagation of such changes between these models.

6.5.1.1.1. Example of propagation of the changes between the models

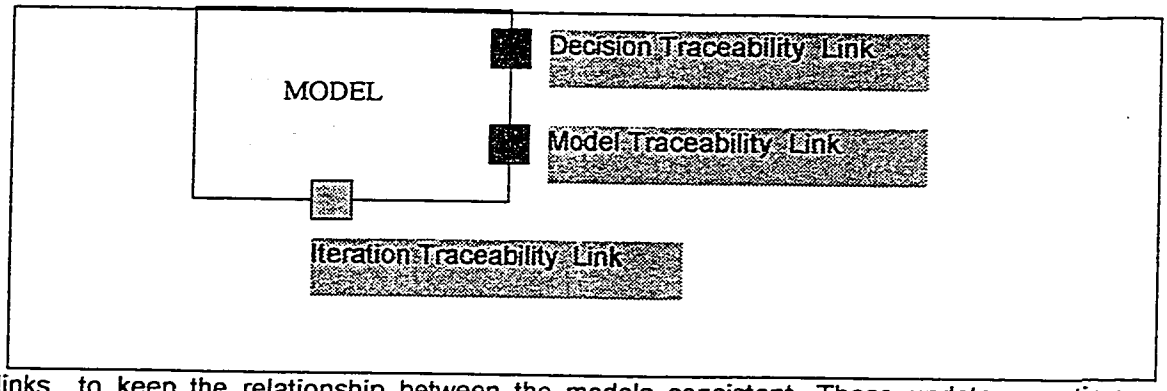
In Table 6-6 we provide a list of the effects that result from changing one model on the other models linked to it.

Model changed	Type of change	Models affected
Use Case	add/delete/modify	• Interaction Diagrams • Object Analysis Specifications • Test cases
Interaction Diagram (ID)	add/delete modification of the ID: add/delete participant object add/delete/ a message	• Use cases • Analysis Object Specification • Class specification • Test cases
Analysis Object Specification (AOS)	add/delete Modification of the AOS: add/delete responsibility add/delete collaborating objects	• Use cases • Analysis Object Specification • Interaction Diagrams
Class specification	See Chapter 3	• Use cases • Analysis Object specification • Class implementations • Test cases • State Transition diagram
Class implementation	See Chapter 3	• Class specification • Test cases
State Transition Diagrams	add/delete a state add/delete a transition	• Class specification • test cases

Table 6-6: Propagation of changes between the models

6.5.1.1.2. Updating the traceability links

One of the major problems associated with traceability is the need for frequent updates of



the links to keep the relationship between the models consistent. These updates are time consuming, requiring the need for an automatic procedure to check and update the traceability links. This problem depends on the implementation environment of the TOOQE method. A data base management system can provide this service (see Chapter 7) by initiating a periodical checking procedure of the links (the time may be set by the user). For example, verify that each model is properly linked based on the pervious rules (see Table 6-1, Table 6-2, Table 6-3), and report any inconsistencies (e.g., Each model need one model traceability model).

6.5.1.2. Traceability and the Validation and Verification of the models

For each model defined in the TOOQE templates, we can use the traceability links to reinforce the validation and verification activities. In the following tables, we give examples of the traceability links traversed in order to check the correctness, consistency and completeness of the analysis and design models that we have already identified in our method [69]. We mention here that this list is not an exhaustive one; it can be expanded to include further details. In addition, we do not deal with implementation issues; they will be addressed in the following chapter.

- Use case specification V & V features :

Links used	Verification to be performed
Req_UC	Verify that all the requirements has been addressed.
Req_UC	Verify that all the use cases can be tracked back to at least one requirement.
Req_UC, Res_UC	Verify that a traceability link has been specified for each usecase

- Analysis object specification V & V features:

Links used	Verification to be performed
Res_UC	Verify that each responsibility is tracked back to at least one usecase.
Res_ID	Verify that each responsibility is associated with at least one interaction diagram.
Obj_CL	Verify that each analysis object specification is referred to by at least one design class specification.

- Interaction diagram specification V & V features:

Links used	Verification to be performed
Use_ID	Verify that all use cases have at least one associated interaction diagram.

Use_ID	Verify that all interaction diagrams are tracked back to at least one use case.
Met_CL	Verify that the design class specification for each relevant analysis object of an ID offers the message used in it.

- Class specification V & V features:

Link used	Verification to be performed
Met_RE, Var_RE	Verify that all responsibilities of the originating analysis object specification are addressed by a design class specification.
Met_Res Var_Res	Verify that each method and instance variable of the design class specification is associated with a responsibility
Met_IM Var_IM	Verify that all the code matches the specification with respect to the existence of instance variables and methods.

6.5.1.3. Analyzing the Class specification modification

After establishing the scope of changes and updating the different models, the development team has to perform regression analysis and testing analysis. They can use the generated MtSSs and MgSSs and compare them with old versions to solve the reselection of test case problem. Furthermore, they can analyze the modification introduced to the class specifications and assess the ripples effects of such changes (see Chapter 3 for further details).

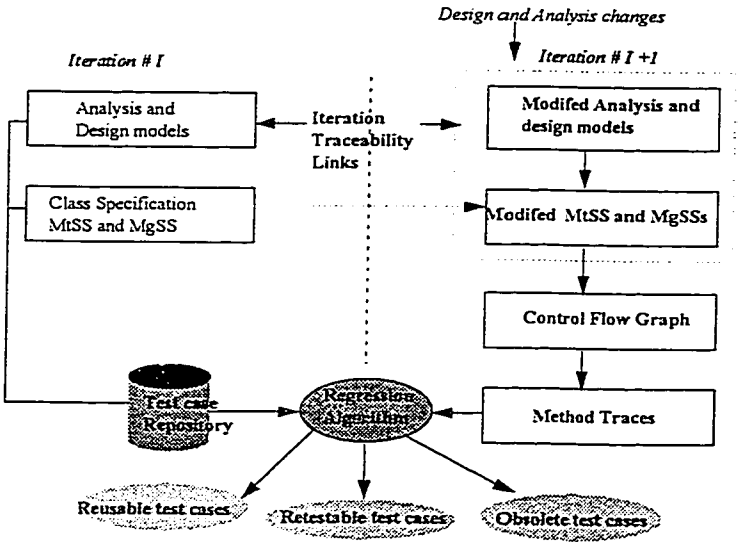


Figure 6-5: Reselection of the test cases based on the class specification

6.5.1.4. Analysis of the changes introduced to one model during the IIDP

The iteration traceability links and decision traceability links are defined to track the full history of one model and the rationale behind the changes introduced to it through the entire development process (see Figure 6-6).

The key benefits of establishing ITLs and DTLs between the models are:

- Reinforcement of the change impact analysis process, by establishing links between the different versions of one model. In this way the development team can navigate forward and backward and control the evolution of the system.
- Facilitation of maintenance activities. The changes are documented in the request change model, so the maintainer will have not only the actual changes introduced to the different models over the development life cycl, but also the rationale behind them.

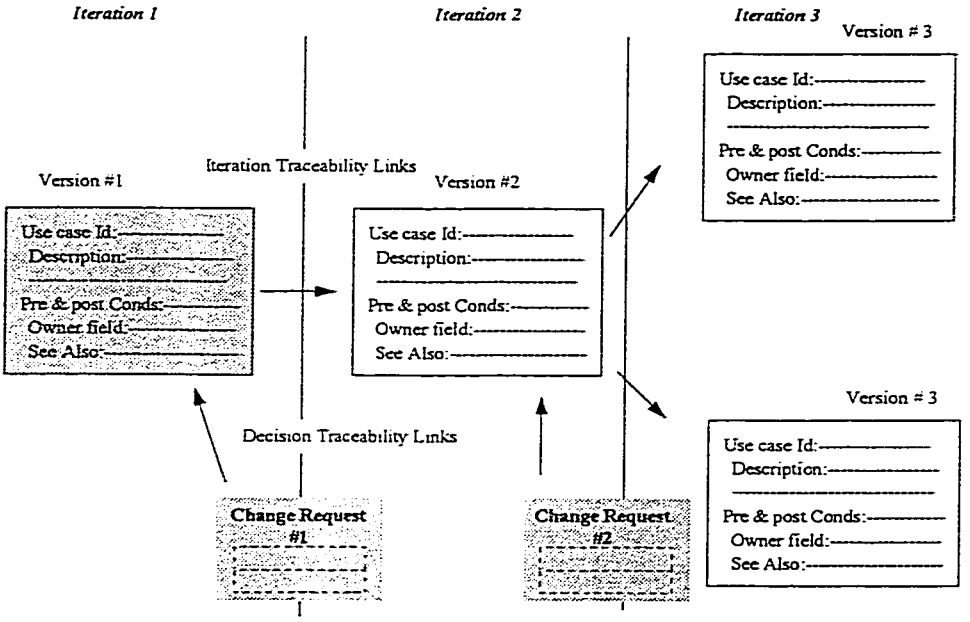


Figure 6-6: Example of change analysis between the iterations

6.5.2. Backward changes

Changes introduced to the code can be detected using a code analyzer tool. We have pointed out in Chapter 2 that code based O-O regression analysis and testing strategies already exist. They compare the old version of a section of code with the modified one and report the changes. Here, we focus on the effects of code changes on the relationship between the code and the analysis and design models.

We use the concept of class firewall [62] to detect the classes affected by the changes. For each class included in the class firewall, we use the backward traceability links to update, delete and create analysis and design models as needed. The MTLs and ITLs will be used to limit the scope of changes. The MtSSs and MgSSs can be used to check for inconsistencies between the modified class implementation and its specification. This task could be automated and incorporated within a traceability tool. Figure 6-7 illustrate the use of traceability links to perform this task.

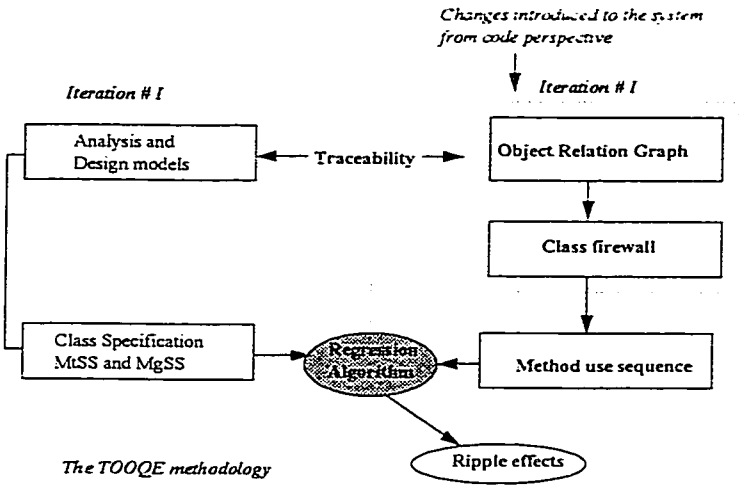


Figure 6-7: Reconciliation between the class implementation and its specification

The following (Figure 6-8) presents how we can use the iteration backward traceability links to detect the analysis and design to be updated.

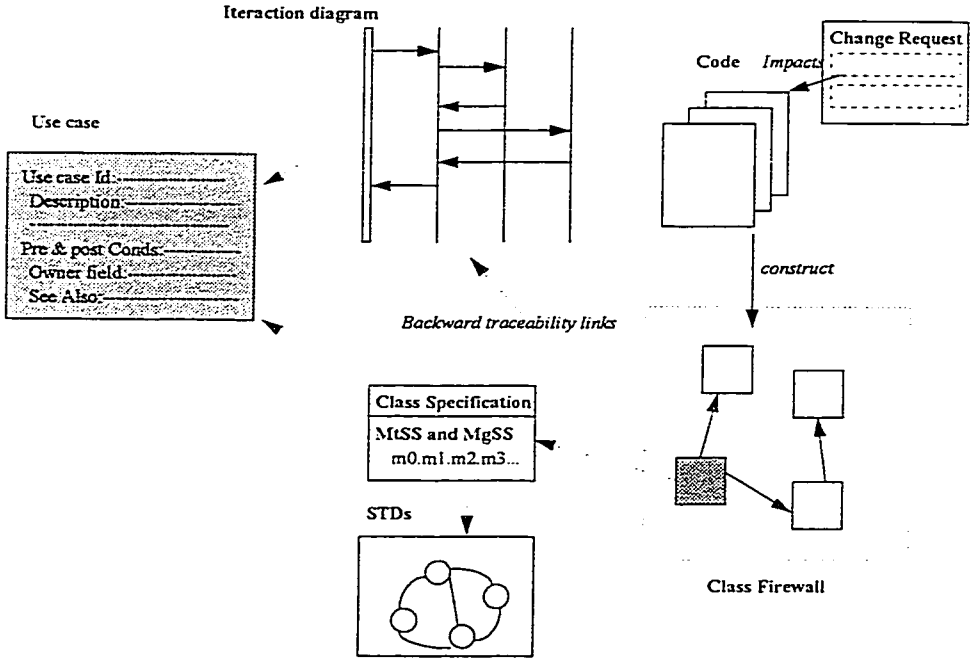


Figure 6-8: Example of the ripple effects of code changes

Changing the code is a manual process. The development team will perform regression analysis by comparing the modified class implementation and the affected classes with their specifications. Next, the backward traceabiltiy links are used to update the corresponding analysis and design models.

6.6. Chapter summary

We have described the conceptual model of the TOOQE methodology designed to manage and conduct the O-O regression analysis and testing activities during an IIDP as identified in previous chapters. We showed how we can use traceability as a basis for identifying the affected analysis, design and implementation models. Traceability will be considered as a essential component of a IIDP framework that includes our method for reselecting the regression analysis and testing package and analyzing the ripple effects of modifications on the class specification and the class implementation. The implementation issues of a tool that will support the method is discussed in the following chapter.

7. Implementation Framework for the TOOQE methodology

7.1. Introduction

In this chapter, we present the functional requirements for implementing the TOOQE framework which includes software repositories, reverse engineering tools, testing tools, traceability tools, measurement tools and a user friendly interface. We describe the components of the system in terms of expected functionality. We do not focus on detailed implementation issues. In the TOOQE model, we abstract software engineering concepts already discussed in the previous chapters, such as traceability, regression analysis and testing, and the iterative development process. We discuss some theoretical issues related to a given functionality of the system, to justify some of the analysis and design choices. Specifically we emphasis the problem of regression analysis and testing in the context of an O-O development process.

The implementation of the TOOQE system requires mainly a relational or an object oriented data base management system or a hypertext framework to ensure the storage of artifacts produced during the development process, and management of the traceability links.

7.2. The TOOQE architecture

The TOOQE environment consists of a repository and six major components, as shown in Figure 7-1.

1. the extractor,
2. the translator,
3. the generator,
4. the Traceability Management System (TMS),
5. the Modification Control System (MCS), and
6. the Quality Monitor System (QMS).

The goal of the first three components is to interface the TOOQE system with software engineering tools such as CASE tools, testing tools, and programming languages. The translator, the extractor and the generator tools cover the different artifacts generated during the IIDP to TOOQE templates. Thus, the system will be totally independent from the environment for development. Also, any modification in the format or the structure of an artifact due to a change in software engineering tools will be handled at the level of these

interfaces and will not directly affect the system. Such interfaces are built to insure the portability of the TOOQE system and to facilitate future maintenance activity.

The TOOQE repository retains a record of the analysis and design process, including both software objects and rationales. The TMS provides functions for capturing the analysis and design artifacts, rationale for decisions, test cases and implementations models, linking them and managing the traceability links. The MCS is used to track the modifications and changes introduced to the artifacts during the development process. It assists the user in V & V activities via the regression analysis and testing functionality. The QMS enables the collected measures to be analyzed with respect to some quality criteria.

In the following sections we describe in detail the functionality of each component. Although the translator, the generator and the extractor are important component of the TOOQE framework , they are not the focus of our analysis and design effort in this research. We simply list here their expected functionality and their high level architecture.

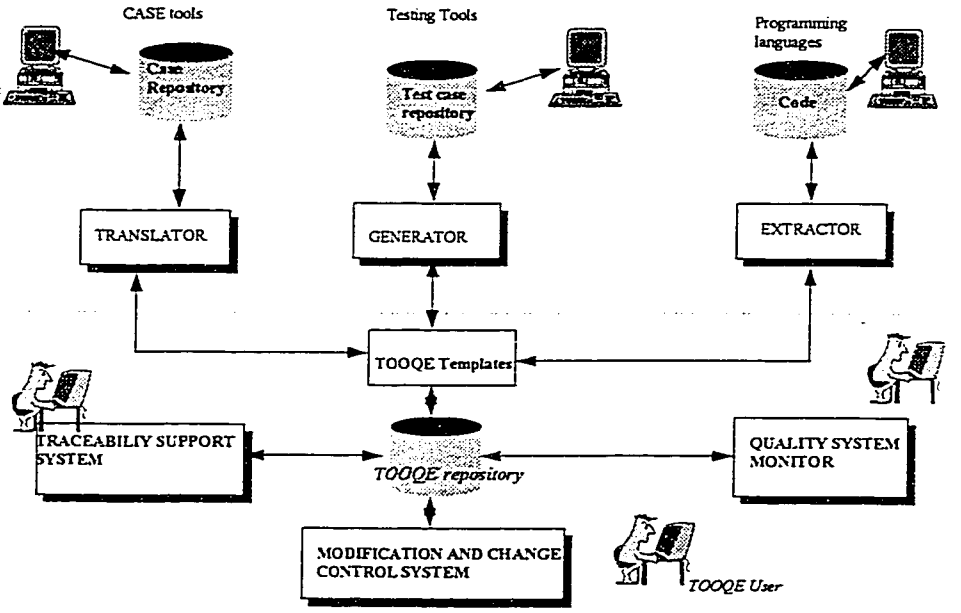


Figure 7-1: The TOOQE architecture

7.3. The Translator

The translator is a bridge technology tool which integrates in the TOOQE system the commercial CASE tools used in forward engineering. The translator has three main functions:

- import
- export
- metric computation

The first function, *Import*, populates the TOOQE repository after forward engineering tools have been used by the development team. In our case, the models produced by the analysis CASE tool may be converted into TOOQE analysis templates. The process of conversion

depends on the method used by the analysis tool. In a similar way, the artifacts produced by the design CASE Tool are converted to TOOQE design templates depending also on the design strategy adopted.

The second function, *Export*, is used to generate useful information which shows the internal structure of the system as implemented in the TOOQE repository. The user of the system can view useful abstractions of the different links between the models generated by the analysis and design CASE tools. Ideally the export function reports to the user modifications introduced to the system during the IIDP. Thus, he/she can update easily his documentation.

The third function, *Metric computation*, works when the import function is activated. Pseudo-code and design metrics are computed and stored in the TOOQE repository. The pseudo-code metrics are computed by parsing the class detailed design generated by the CASE tool. The computation of the design metrics, on the contrary, has the TOOQE repository itself as input. It captures all the necessary information form the TOOQE templates describing the links between the different objects and models of the system.

7.4. The Extractor

The extractor is a reverse engineering tool which produces structural information according to a language-oriented schema, and computes software metrics. The function of the extractor is similar to that of a compiler: It analyses the sources code of some high-level programming language except that code generation is replaced by database generation.

The extractor works in five phases:

- *Lexical phase:* collects data related to operators specifying actions, or operands representing data.
- *Syntax phase:* produces objects including instances of entity types, calling relationships, and declaration of components.
- *Semantic phase:* Analyses the structure of objects in terms of methods and instances variables and their interaction with the other objects.
- *Storage phase:* stores entities and relationship types which characterize one language oriented schema in the TOOQE repository.
- *Design metric computation phase:* uses the structural information, which has previously been stored in the TOOQE repository, to compute inter-object attributes.

Such a tool can be developed using tools such as LEX, the lexical analysis tool and YACC, the parser generator tool.

7.5. The Generator

The generator is a tool which integrates commercial testing tools into the TOOQE system. It provides three main functions:

- import
- export
- metric computation

The first function, import, populates the TOOQE repository after the testing tools have been used. It maps the generated the generated test cases to the TOOQE testing templates. The conversion depends on the testing strategy used by the tool.

The second function, export, is used to update the testing plans of the system. It provides the testing team with the necessary feedback after modifications and changes are introduced to

the system. This function covers the regression analysis and testing package, and the modified parts of the system to be revalidated.

The third function, metric computation, collects structural and functional metrics and quality attributes generated during the testing process, and stores them in the TOOQE repository. These metrics are used to assess the quality of the software under development, and to assist the testing team in their validation and verification activities.

7.6. The TOOQE templates

The TOOQE templates collect the useful information from the models generated by the different tools during the IIDP process. We mean CASE, testing and development packages. They also incorporate additional information needed for management of traceability and modification information. The structure of the template depends mainly on the model that we are trying to abstract. This not an easy task, especially in the case of the absence of a certain formalism in a particular methodology used to describe the model.

7.6.1. Use Cases template

- unique identifier
- title
- requirement field
- owner field
- pre and post conditions
- revision history
- key words for cross-referencing
- description
- forward traceability links
- testing link

7.6.2. Interaction Diagram template

- unique identifier
- type
- description
- backward traceability link
- forward traceability link
- owner field
- testing traceability link
- revision history

7.6.3. Object Specifications template

- object identifier
- description
- object type

- responsibility table
- backward traceability table
- forward tractability table
- interaction table
- revision history
- keywords

7.6.4. Class Specifications template

- identifier
- description
- class type
- release status
- inheritance information
- responsibility table
- instance table
- method table
- testing table
- revision history
- backward traceability link
- keywords

7.6.5. Object behavior template

The template describes the behavior of an object in terms of states and transitions(messages exchanged between the objects). We can use a state transition graph, or an SDL diagram.

7.6.6. Test case template

This template describes the different fields of a test case designed to test the implementation of a class or a group of interacting objects.

- unique identifier
- test case purpose
- test case type
- testing strategy
- list of the test case steps. Each step will contain:

  * A list of specified states for the object under test (If applicable).

  * A list of messages / methods that are to be used in the execution of the test case step (if applicable).

  * A list of exceptions to be raised by the object being tested (if applicable).

* A list of interrupts to be generated by the object being tested (if applicable).
* A list of external conditions (if applicable).
* A list of additional comments.

- backward traceability table
- forward traceability table
- owner field
- revision history

7.7. The Traceability Management System (TMS)

The Traceability Management Tool (TMS) tool implements the three facets of traceability as specified in Chapter 6:

- Traceability among all the models produced during one iteration of the development process. The tool links together the user requirements, the analysis models, the design models, code and test cases. It provides forward and backward navigation facilities.
- Traceability between the different iterations which link the different versions of one model through the whole process of software development. The tool provides mechanisms to track the full history and the evolution of an artifact and its links.
- The tool allows the development team to establish links between the main analysis and design decisions and the software models. The tool keeps track of all the decisions and their evolution. It provides navigation mechanisms between the software models and decisions through different iterations.

The TMS has a layered architecture (See Figure 7-2) consisting of: [(1) the TMS core, (2) the service layer, (3) the interaction layer].

1. The core provides database schemes for recording decisions and linking them to software objects in the TOOQE repository. It is also responsible for providing a search language for retrieval of models, and their mutual relationships. The TMS core is the only layer which depends on the DBMS used to manage the TOOQE system.
2. The service layer provides operations for organizing information about the model decisions and software objects. It manages the traceability links among the different models and provides the development team validation and verification services.
3. The interaction layer defines the user interface and presentation of the different models of the system under development. It provides a friendly graphical and textual environment that allow the user to interact easily with the TOOQE framework. It also covers the user help and the user assistance services.

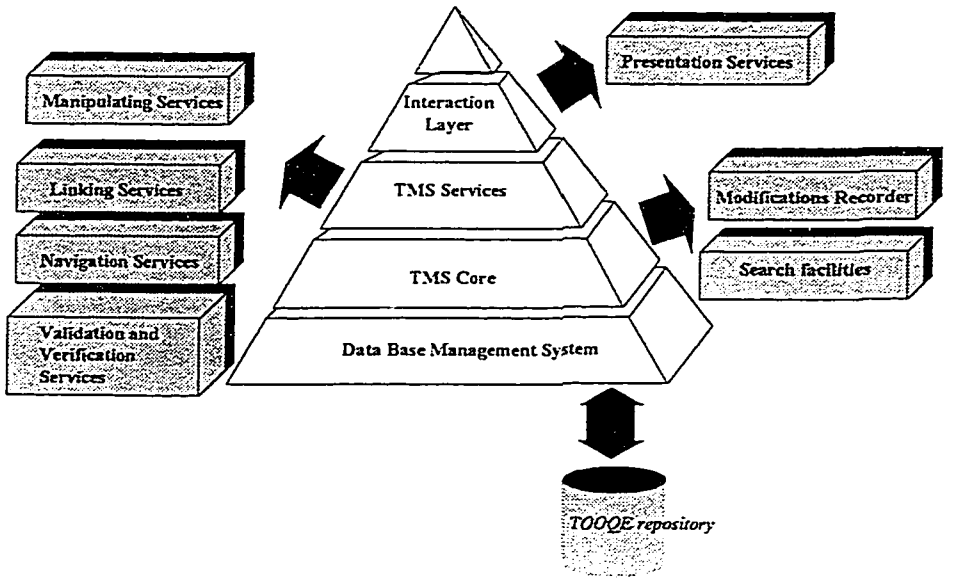


Figure 7-2: Traceability Management System (TMS).

7.8. The TMS tool services layer

7.8.1. The manipulating services:

They are used to build the traceability model. They are classified in two categories as follows:

- Decision manipulating services:

Name	Description
Create	Store a new decision after it has been edited.
Delete	Remove a waiting decision from the repository.
Update	Modify the content of a waiting decision.
Instance	Instantiate a decision from a generic one adding a link.
Replace	Substitute an active decision with a new one adding a link.
Connect	Modify a waiting decision to active status.
Disconnect	Revert a waiting decision to waiting status.

- Model manipulating services :

Name	Description
Creatmodel	Store a new model after it has been edited.
Updatemodel	Modify the contents of a model.
Deletemodel	Delete a model from the repository.

7.8.2. Linking services:

The traceability links are bi-directional and they are classified in different categories:

- Analysis and design links cover :
  1. Links between two analysis models.
  2. Links between an analysis model and a design model.
  3. Links between two design models.
  4. Links between an analysis model and an implementation model.
  5. Links between a design model and an implementation model.
- Test case links encompass :
  6. Links between a test suite and an analysis model.
  7. Links between a test suite and a design model.
  8. Links between a test suite and an implementation model.
- Decision links cover :
  9. Links between a decision and analysis model.
  10. Links between a decision and design model.
  11. Links between a decision and implementation model.

The linking services provided by the TMS tool are summarized as follows:

•Model linking services:

Name	Description
Linkmodel	Links two models.
Deletelink	Deletes a link between two models.
Updatelink	Updates a link between two models.

- Decision linking services:

Name	Description
LinkDemodel	Links a model to a decision.
Linkcause	Links a cause decision to an effect decision.
LinkDeremove	Removes a link between a decision and a model.
LinkJustified	Links a justifying decision to a justified decision.

7.8.3. Navigation services:

They are summarized in the following table:

Name	Description
Tracebackward	Starting from a model, navigate backward the traceability links to the originating higher-level model. It encompasses the decisions and models involved in one iteration.
Traceforward	Starting from a model navigate forward the traceability links to the following lower-

	level models. It includes the decisions and models involved in one iteration
Traceamodel	Track the evolution of a given model through the entire development process
Whichlink	Returns the link type between two artifacts.
History	Returns all historical predecessors of a decision

7.8.4. Validation and Verification services

This layer provides the development team with the necessary information and feedback needed to test the analysis and design models. To assist the V& V activities, we adopt three interrelated testing criteria that are common to all models:

- correctness
- completeness
- consistency

These attributes must be interpreted in the context of the iterative incremental approach. Early versions will not be expected to attain the same level of detail as later versions. The emphasis of the testing process is the feedback into the development process. Correctness and completeness are judged against the aspects of reality which the model is intended to represent.

The model is complete if it is judged that the entities describe the aspects of the knowledge being modeled in sufficient detail for the goals of the current iteration. Consistency is judged by considering the relationships among the entities in the model. An inconsistent model has a representation in one model that is not correctly reflected in other portions of the model. These may be contradictions, or differences in the level of detail.

The process of mapping one model into another model can introduce errors. Testing the new model is partially accomplished by comparing it to the original model. The mapping should be complete, which means that every entity in the original model should be accounted for in the transformation. The mapping may result in an existing entity being eliminated form future models or that the entity may be split into two more smaller, more specific entities. The mapping should maintain the same relationship among the entities in the previous model.

The TMS uses a certain number of techniques to assist the development team in testing the analysis and design models. It implements the set of verfication and validation services specified in Chapter 6 by relying on the definition of the traceability links defined between the models and the TMS core services for searching and retrieving the information. The services which cover the V & V of the TOOQE templates are the following:

- Use Case Specification V & V features :

Name	Description
VerfiyReq	Verify that all of the requirements have been addressed.
VerifyUseCase	Verify that all use cases can be tracked back to a least one requirement.
VerifyRedUseCase	Detect redundancy between use cases.
VerfiyTracUseCases	Verify that a forward traceability link has been specified for each usecase

- Analysis Object Specification V & V features:

Name	Description
VerifyRespUc	Verify that each responsibility is tracked back to at least one usecase.
VerfiyRespId	Verify that each responsibility is associated with at least one interaction diagram.
VerifyAnaly	Verify that each analysis object specification is referred to by at least one design class

	specification.
VerifySubsyst	Verify that highly-coupled objects exist at the same subsystem.

- Interaction diagrams Specification V & V features:

Name	Description
VerifyUcId	Verify that all use cases have at least one associated interaction diagram.
VerifyIdUc	Verify that all interaction diagrams are tracked back to at least one use case.
VerifyIdClass	Verify that design class specification, for each relevant analysis object of an ID offers the message used in it.

- Class Specification V & V features:

Name	Description
VerifyObjClass	Verify that all responsibilities of the originating analysis object specification are addressed by a design class specification.
VerifyMethResp	Verify that each method and instance variable of the design class specification is associated with a responsibility.
VerifyClassCod	Verify that all code matches the specification with respect to the existence of instance variables and methods.
VerifyTestmeth	Verify that at least one test case, and one test case driver, exist for each method.

7.9. The System Quality Monitor (SQM)

The SQM is a tool which incorporates three main quality functions:

- assessment
- control
- prediction

Quality assessment provides relative comparison of the quality of software components. Quality control identifies software components whose quality values exceed standard quality thresholds, or degrade over time. Quality prediction forecasts the quality of software components using measures which are available early in the software life cycle.

In each case, a model of quality is used to define an association between the external and the internal attributes of software products in the TOOQE repository. Although external attributes, such as maintainability, reliability, or usability, are the ones that most software people need to know, they cannot be measured directly. Indirect measures of internal attributes, such as size, or coupling, are needed to measure the overall software quality.

The SQM tool represents an important component of the TOOQE system. However, due to the limitation of time and space, we cannot go further in our analysis. We mention that the tool should adopt a model to describe the external and internal attributes in order to quantify them. It will not incorporate built-in analysis capabilities, but it can interface with other external tools with different underlying analysis techniques.

7.10.The Modification Control System (MCS)

The MCS is a tool for capturing modifications and tracking changes introduced to the different models produced during the development process. It also enables designers to perform change impact analysis, and perform regression analysis and testing activities.

The MCS offers the following set of functionalities:

- A full history of all the modifications introduced to one model during the different iterations.
- The rationale behind major change decisions, and their impact on analysis and design models. For example, a given use case is split into new use cases may imply the creation of new objects and associations.
- The modifications introduced to one model during a micro iteration, and their ripple effects on other linked models.
- The regression analysis and testing activities.

The first three features in the previous list are related to the analysis impact activity. They are conceived to help the user in documenting the system and to assist him in the analysis and design effort. The last one is oriented to the testing activity.

In a similar way, the MCS service layer has layered architecture, consisting of

1. the MCS core,
2. the service layer,
3. the interaction layer (see Figure 7-3: Modification Control System (MCS))

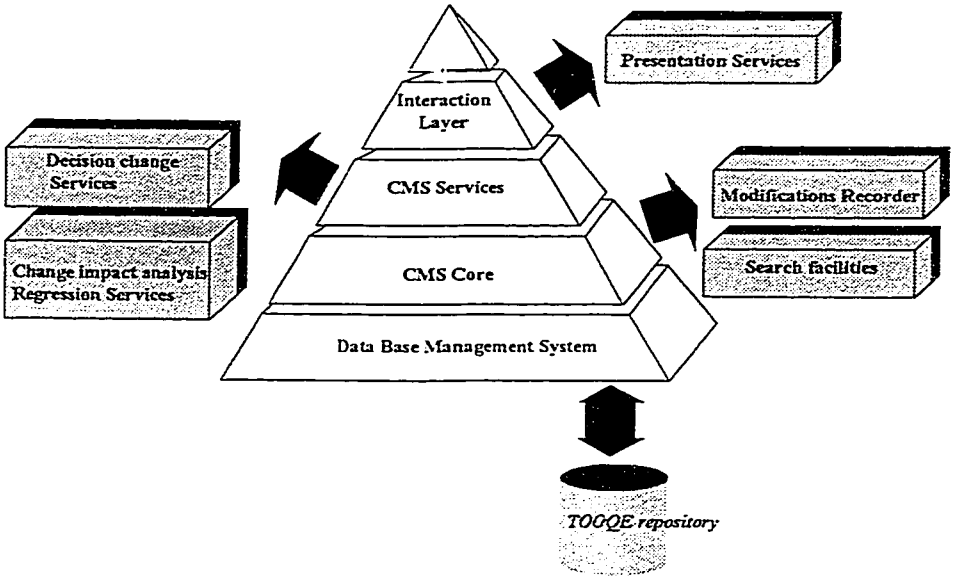


Figure 7-3: Modification Control System (MCS)

7.10.1. MCS services:

- Decision analysis services:

Name	Description
DecisionImpact	Track the impact of a decision on the different models.
DecisionHistory	Track the evolution of decisions and their relationships through the entire development process.

- Change Impact services:

Name	Description
ModifyModels	Track the modification introduced to one model and their ripple effects on other linked models
HistoryModels	Track the changes introduced to one model through the different iterations of the development process.
SelectTestCases	Select the regression analysis and testing package.

7.11.Chapter summary

In this chapter, a framework for implementing the TOOQE methodology is discussed. The specifications and functional requirements for the different component of the a TOOQE system are described informally. We define a set of templates extracted from the analysis and design models and we introduce two main components of the system: a Traceability Management System (TSM) and Modification Control System (MSC). The TSM is designed to handle the traceability links and navigation functionality between the TOOQE templates. The MSC is designed to perform regression analysis and testing.

8. Integrating of the TOOQE concepts within the ObjecTime tool

8.1. Introduction

ObjecTime ™ is a powerful graphical modeling environment for the object-oriented design and simulation of real-time systems. It is ideal for rapid prototyping of distributed, event-driven systems using synchronous or asynchronous communication, and the development of efficient implementation for a target real-time platform. The implementation is generated directly from the Toolset, so that the implementation always stays in synchronization with the models.

ObjecTime is set of tool that spans critical portion of software development lifecycle. Currently, ObjecTime has tools for capturing requirements, and designing, executing, and documenting designs. The tools are integrated within the development environment.

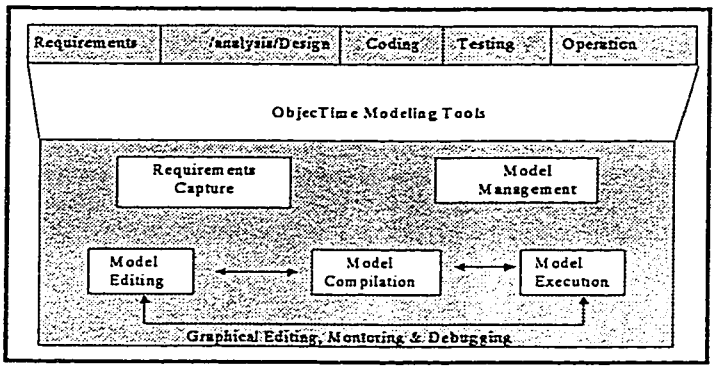


Figure 8-1: ObjecTime toolset

8.2. The Basic ObjecTime concepts

8.2.1.1. Actors

Structural components in ObjecTime are called actors and are the main unit of design. Actors have the following properties:

- They are potentially concurrent.
- Actors communicate by sending messages.
- They are encapsulated.
- They have a structure.
- They can have a behavior.

8.2.1.2. Messages

Actors communicate and interact with each other by sending messages. Each actor has a set of messages that it will respond to. A design represents some overall system, and each actor makes up some part of the system. The overall behavior of the system is comprised of the behavior of all its interacting components. Furthermore, actors are not directly aware of other actors in the design; they only see their own interface through which they may communicate. This is what is meant by encapsulation. Other actors send messages to request that an actor perform the functions for which it is responsible.

8.2.1.3. Actor classes, Inheritance and Actor structure

An actor class is a specification for a type of actor that will appear in the design. All actors have a class specification. Two or more actors of the same type used in the design are said to be references of the same actor class. Hence, a class is like a template for creating actors. An actor class specifies an actor's structure and behavior, as well the messages it can send and receive. Classes can be stored in library for reuse purposes.

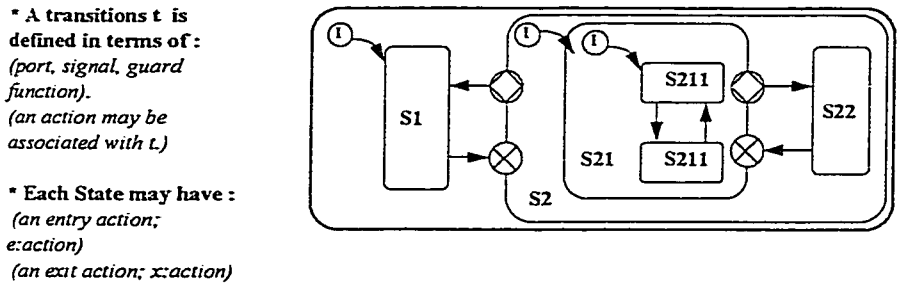
Inheritance is one of the means by which classes are reused. Classes are organized in an inheritance hierarchy. Subclasses inherit various attributes from superclasses such as structure, behavior and design documentation. Inheritance is an abstraction and reuse mechanism for the system components.

An actor class may contain references to other actor references. This is a way of simplifying designs by allowing complex actors to be decomposed into simpler actors. Decomposition is an important principal in the ObjecTime tool set. The structure of an actor captures the communication and containment relationships among system components.

8.2.1.4. The Actor behavior model

The components of a system must have a way of reacting to a system event (message). Also, they must have a way of communicating with other components of the system (or even outside the system).

All actors can have a behavior, which defines how the actor will respond to incoming messages. The behavior is specified using hierarchical finite statecharts. When an actor receives a message, a transitions may occur causing the state machine to perform some action, and possibly move to a new state.



•The State behavior of a ***complex Actor*** is viewed as consisting of three co-parts:

–The inherited part (State charts of the base actors from which the complex Actor is derived).

–The aggregate part (State charts of the components Actors).

–The defined part (State charts defining the " unique" behavior of the Actor itself).

Figure 8-2: Actor behavior

8.2.1.5. Ports and binding

Ports are a means for actors to communicate with each other. For an actor to communicate with another, they must each have a port, and the ports must be connected by a binding. Actors communicate and interact primarily through ports and bindings. A port is a reference to a protocol class which defines the set of messages that a port is permitted to send or receive. Ports can be attached to the interface of an actor's structure or be attached internally for communication within an actor.

8.2.1.6. Data objects

Data objects are used in the actor behavior. A data object is similar to an actor. It is encapsulated and thus is accessed via messages on its interface, except that a data object can be considered to have a single, implicit port on its interface. Furthermore, it is passive and always executes within the thread (process execution) of control of an actor.

A variety of base data types are supported, which are based on language independent types. Data objects can also be sent and received by actors using messages. The data in a message are processed by the behavior of the receiving actor.

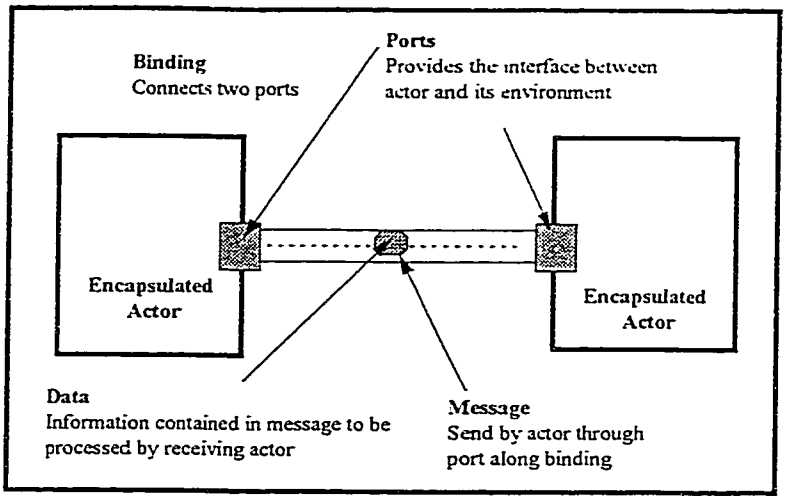


Figure 8-3: The ObjecTime actor concepts

8.2.1.7. Requirement Traceability:

The ObjecTime requirements definition and traceability features provides the ability to create or edit requirements, and to link these requirement to entities in the model. Three level of manipulation of requirements are considered in the tool:

- The requirements definition facilities are intended to provide a simple set of capabilities for the definition and manipulation of requirements. In many cases, these facilities may be all that is required for a group to manage the requirements relating to their object models. Users who have an existing requirement tool (e.g., DOOR) may exchange requirements files between ObjecTime and their existing external tool.

- The requirements definition functionality allows the customization of the types of the requirements that can be defined, as well as the properties for each type of the requirement. Three types of requirement are considered: general requirements, functional requirements, and interface requirements [68]. Requirements are organized in a hierarchy, and the tool provides facilities to create, update and delete requirements.

- The requirements traceability functionality allows the user to link requirement model entities within an ObjecTime model. Verification functions are provided to allow the user to ensure that all the requirements are linked to model entities, and to ensure all model entities are justified by the requirements. The model entities affected by the changes to requirements can be easily determined.

8.2.1.7.1. Requirements Verification test

The ObjecTime requirements definition and traceability features provide a simple collection of verification tests that can be applied to requirements specification or an ObjecTime model.

In the following, we list an example of the verification requirements tests provided by the tool:

- All requirements have a unique name.
- All the requirements are properly related.
- All the requirements are linked to a model entity.
- All requirements are assigned to individual.
- Model entities, such as actors, bindings, message sequence charts, etc. are linked properly to the requirements.

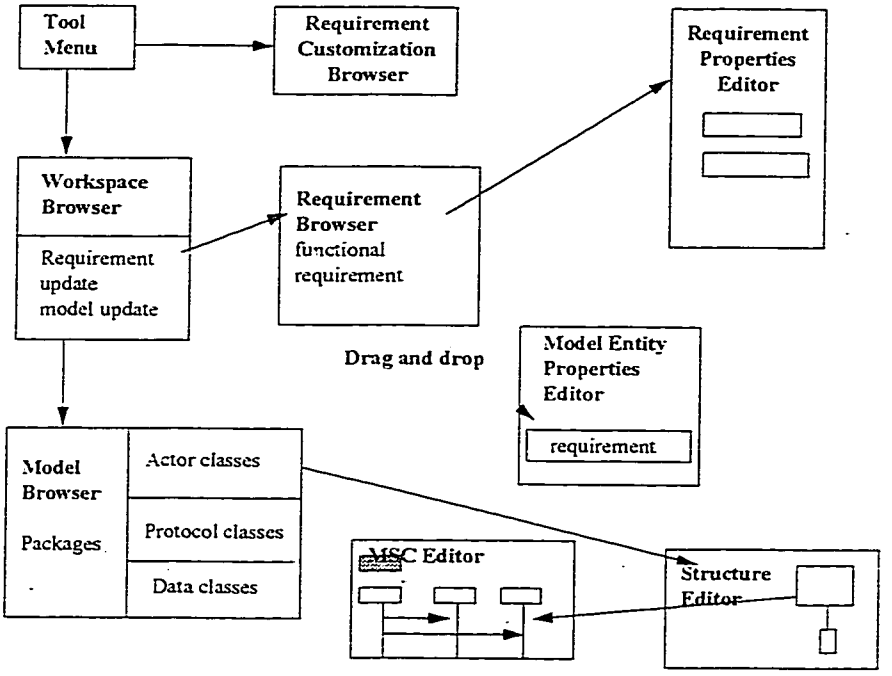


Figure 8-4: Requirements capture in ObjecTime.

8.3. Possible Extension of the ObjecTime tool set:

8.3.1. Introducing the Iteration and Gate concepts

ObjecTime assumes that designers adopt an iterative incremental development process. The tool provides a set of functionalities to produce and manage different version or prototypes of the system under development. The main concept is called *context* [68], where the user can define the requirements, design the models, execute the system and save the changes introduced to the system. The primary purpose of the definition of the context is configuration and management of the different releases of the software in a multi-developer environment. Although, the functionality as implemented in the tool allows the designers to keep track of all updates of the systems, it lacks what we call "check points" where the system can be validated with user and tested against its requirements or previous versions. A validation *Gate* ends the assessment phase of a macro-iteration and marks the beginning of another one. We introduce the concept of *iteration* which should be associated with a subset of the original requirements of the system and validated by the user. The production of the software will be divided into a certain number of iterations. An iteration contains several micro iterations which can be mapped

directly into the context concept of ObjecTime. The latter includes several updates generated by different software developers. Figure 5-5 depicts the proposed model.

The number of iterations, and the functionality to be implemented should be decided by the project manager in collaboration with users and the developers. The iteration is ended after the assessment of the prototype produced and correcting the different models (analysis, design, and implementation).

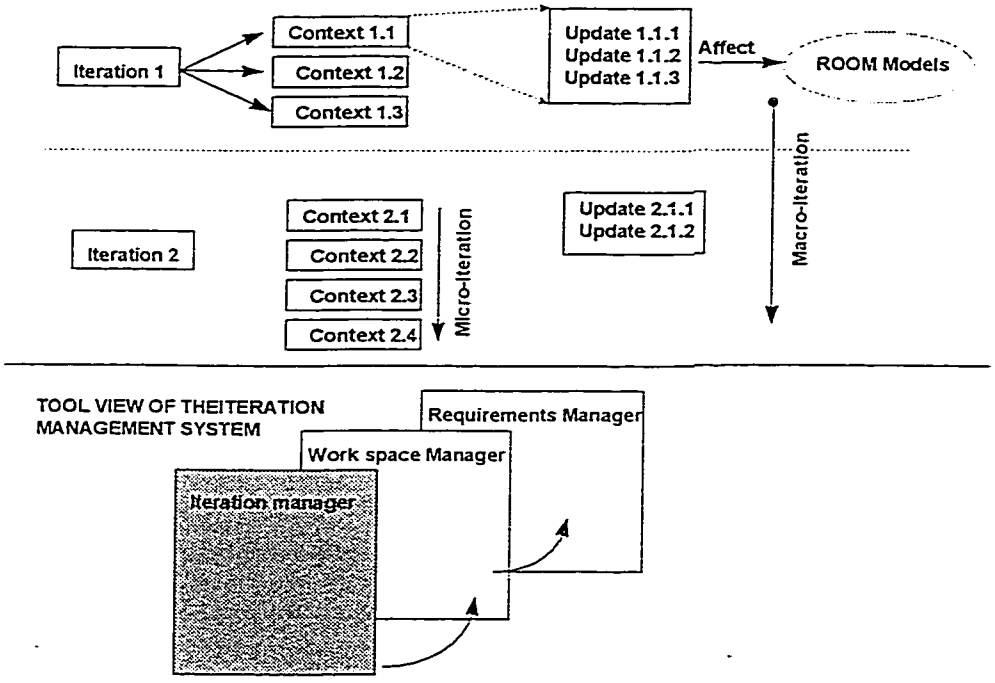


Figure 8-5: Possible iteration management in ObjecTime

Regression analysis and testing takes place to verify that the modified version of the system meets the original and new set of requirements of the system.

The implementation of such functionalities will not need major changes to ObjecTime tool set.

8.3.2. Feasibility and advantages

One of the main advantages of such features are that the software developers will have a more consistent development process where the user is involved in the development process at planned intervals. The implementation of such functionalities will not need major changes to the ObjecTime tool set. The original repository will be extended to incorporate new tables. For example, one iteration can be characterized by the following fields:

- starting date
- finishing
- list of people involved in the iteration
- list of tasks assigned to each individual (e.g., subsystems to be built, subsystems to be validated)
- list of requirements addressed by the iteration

Such feature can be implemented and provided to the users as an optional functionality that can handle some aspects of the project management during the development process. It may also be used to incorporate information gathered from project management third party tools .

8.3.3. Improving the traceability functionality

We have already described the traceability concept, and how it helps in the change impact analysis and regression analysis and testing activities. Here, we propose to include the following models and linking services in the traceability tool set of ObjecTime:

- decision models
- test suite models
- vertical links between the different versions of one model
- navigation and traceability verification services

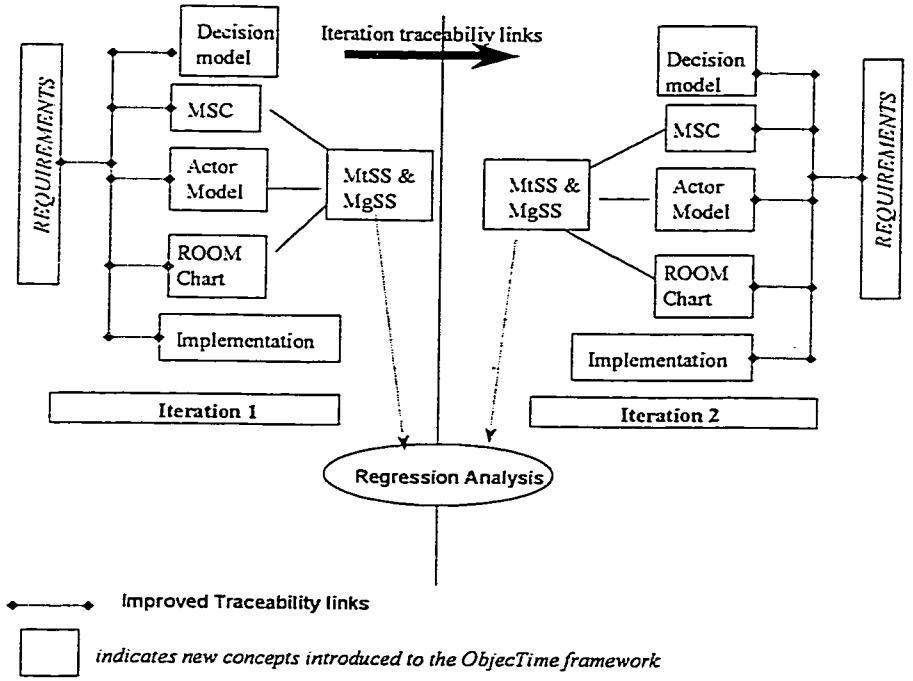


Figure 8-6: Possible extension of the ObjecTime environment

8.3.3.1. Decisions models

As mentioned in Chapter 6, the decision model will keep track of the major decisions taken as designers update the system. Such decisions may affect the requirements, and they are usually motivated by the implementation environment constraints or performance concerns. The decision models keep track of all the information needed to identify a change such as purpose of the change, owner, and date of the change. The designer establishes explicitly the links between the models to be updated and the change decisions.

8.3.3.2.Linking and navigation services

One major difficulty with traceability is how to update the links between the models and how to ensure the consistency and correctness of such operations. We propose the following extensions:

- Add semantics to the traceability links. ObjecTime traceability links are a simple cross referencing between the models. We suggest that a link will contain not only the information about the models tied together, but also the owner of link, the date of creation, and many other fields (see Chapter 6 for complete description). The latter information can be used to check the consistency of links, and initiate periodic updates (e.g., links can checked automatically after a certain period of time).

- The links between the models provides the following services. First, the ability to navigate between the models during the same iteration, to ensure the traceability of the requirements or to know the rationale behind changes. Second, detect and assess the ripple effect of changes. Whenever a model is changed, its links become insecure and the models tied to it are reported for update and modification (further discussion are provided in Chapter 6).

The implementation of such links and services requires the additions of new subsystems to control and manage the modification and traceability links. The user can have a unique interface to access to such functionality or it can be distributed over the other subsystems (e.g., requirements management system).

8.3.3.3.Regression analysis based on the MtSS and MgSS Actor Specifications

We introduced the concept of MtSS and MgSS in Chapter 3. We describe the process of generating such sequences from the analysis and design models. Our approach can be used with ObjecTime models, namely the message sequence charts, the ROOM charts and the actor structure model. The method can summarized as follow :

- Generate the new MtSS and new MgSS for each modified actor from the analysis and design models at the end of the iteration.

- Perform regression analysis and match the existing test cases with the new MtSS and MgSS using the regression algorithm described in Chapter 4.

- Classify the regression test suite.

8.4. Chapter summary

In this chapter, we discussed how some of the TOOQE concepts can be integrated in an existing CASE tool called ObjecTime. The current version of the tool provides users with an integrated development environment, including analysis, design, and code generation capabilities. We presented how to extend the existing validation and verification capabilities of the tool the by including the IIDP concepts, traceability and regression analysis features.

9. Conclusion and future research directions

Object-oriented software development utilizes iteration much more than does software development using traditional techniques such as structured analysis and design. The multimodel nature of the different O-O development methodologies promotes several iterations. The system evolves, and functions (classes) are incrementally added and modified from one iteration to another. Changes are introduced to the different models from analysis, design and code perspectives to accommodate additional user requirements or development environment constraints.

Throughout this thesis we demonstrate the need for a systematic methodology to handle requirements changes. We achieve this by detecting design changes and deriving the ripple effects on a key set of existing interaction sequences during the migration between iterations. The problem of regression analysis and testing in an iterative incremental development process is defined. Existing regression analysis and testing methodologies are discussed and criticized. A novel method for conducting and managing such activity is presented. The underlying process of the method is based on the method (MtSS) and message (MgSS) sequence specification concepts. The latter concept is used to model the inter-class and intra-class behavior based on a regular expression formalism. It is important for the reader to note that this formalism is not adequate for representing the complete functionality of most objects. Nevertheless, the order of interactions and calling sequences of the methods for a class can be represented by regular expression. Since our intention in this thesis is regression analysis, this formalism is well suited and it is used here. Chapter 2 provides the reader with background information related the to O-O technology, especially O-O testing. The problem of regression analysis and testing is identified and related work is discussed. Chapters 3 through 6 describe the details of the method. In Chapter 7, a high level support tool for the method is presented. Implementation and scalability issues are discussed. A possible integration of the method with an existing real-time, Object-oriented software design toolset called ObjecTime is presented in Chapter 8. The latter discussion is provided as an explicit proof of the feasibility of the TOOQE concepts.

9.1. Future work

The validation and verification activities in the O-O paradigm focus on the analysis and design models. Hence, investigation and new research is needed in order to integrate these activities earlier in the development process. This thesis opens up a set of new research topics related to Object-oriented regression analysis and testing.

In the following, we briefly discuss these issues:

- Integration of V & V with the analysis and design phases. This is still an open area of research. The main advantages of such approach are reducing the cost and the effort devoted to the field testing and maintenance of a system. New methods for verifying the

correctness, completeness and consistency of Object-oriented analysis and design models are needed. Our approach is a step in that direction. The MtSS and the MgSS can be used to perform such validation.

- In the following, we discuss a set of possible extensions in order to carry out the validation and verification of the models:

  ⇒ First, new constructs and operations can be defined and added to the formalism to facilitate the manipulation of the MtSS and the MgSS.

  ⇒ Second, incorporate data in the message traces.

  ⇒ Third, enhance the method of generation of the MtSS and MgSS from the detailed and design specification models. Our approach, as described, depends on the model used to generate the MtSS and the MgSS. However, there is no agreement in the software engineering community on what is the best formalism used in O-O methodologies. An improvement of the method would involve a more precise and formalized definition of the process of generating the MtSS and MgSS.

- Object-oriented test case perspective. We mentioned in Chapter 2 that many of the existing regression analysis and testing methodologies consider the test cases as set of inputs and expected outputs. However, in O-O design, software objects encapsulate data and behavior. Hence, there is a need for a method to assess the impact of changes on the relevance of state-based test cases during regression analysis and testing. Our approach handles this by using the state transitions diagrams. However, the method can be extended to include more complicated behavior models such as hierarchical and communicating state charts.

- Integration of the TOOQE methodology with an existing commercial O-O CASE tool called ObjecTime. As we mentioned earlier in Chapter 8, the tool provides the user with an integrated environment of development which includes requirements capture, structure and behavior modeling and code generation. Our work will improve the traceability capabilities and extend the verification and validation activities.

- Tool support for promoting traceability: Although many commercial tools provide traceability, they do not provide full change capabilities. Adding semantics to the traceability links, and incorporating intelligence to automate the linking services (capturing /modifying/ deleting) between the models, are new areas of research. These will help reduce the overhead of maintenance activities and improve the efficiency of such tools.

- Complexity analysis and generating metrics. We did not investigate the effects of using our regression analysis and testing methodologies on the coverage of the modified code and the system specifications including the analysis and design models. Gathering metrics is essential to measure the effectiveness of a method. However, generating metrics and performing complexity analysis of O-O software is still largely a research problem [70]. Many testing experts suggest that existing strategies designed for procedural languages can be applied at the class level, but such strategies are not suitable for higher level components such as class hierarchy testing and class library testing. Moreover dynamic biding and polymorphism complicate such tasks.

Bibliography

[1] S. Kirani and W.T. Tsai. Specification and Verification of Object-Oriented Programs. *PhD Thesis. University of Minnesota USA*. December 1994.

[2] G. Adams . Describing groups of interacting objects using Path Expression. *Masters Thesis. University of Ottawa Canada*. December 1992.

[3] J.E Hopcroft and J.D Ullman. *Introduction to Automata theory, Languages, and Computation*. Addison-Wesley Publishing company, 1979.

[4] D.W. Bustard and C. Winstanley. Making changes to formal specification, Requirements and an Example. *IEEE transaction on Software Engineering*. Vol 20, No 8, page 562-569, 1994.

[6] C.D Turner and D.J. Robson. State Based testing and Inheritance. *Technical report: TR I/93. University of Durham England*, 1993.

[7]G. Rothermel and M.J. Harrold. A comparison of regression test selection techniques. *Technical Report. Clemson University USA*. October 1994.

[8] R.M . Poston, "Automated Testing from Object Models," *Communication of the ACM*. Vol 37 No9 (September 1994), pp.

[9] P.C. Jorgensen and C. Erickson. Object-oriented Integration testing. *Communication the of ACM*. Vol.37, No.9. September 1994.

[10] R. McDaniel and J.D McGregor. Testing the polymorphic Interactions between classes. *Technical Report. University of clemson USA*. April 1994.

[11] I. Jackobson . *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM/Addison-Wesley, 1992.

[12] G.Rothermel and M.J Harrold. Selecting Regression tests for Object-Oriented Software. *Technical Report. University of clemson USA*. June 1994.

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

[14] J.D McGregor and T.D. Korson. Integrated Object-Oriented testing and development processes. *Communication of the ACM*. Vol.37, No.9. September 1994.

[15] E.V. Berard. *Essays in Object-Oriented Software Engineering*. Prentice Hall Publishing 1993.

[16] R.M. Poston. Testing Tools combine Best of New and Olds. *IEEE Software* . page 122-127. March 1995.

[17] B. Marick. *The Craft of Software Testing: subsystem testing including Object-based and Object-Oriented testing*. PTR Prentice Hall Publishing, 1995.

[18] J. Rumbaugh, M. Blaha, W. Premelani, F.Eddy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall Publishing, 1991.

[19] G. Booch, *Object Oriented Analysis and Design with applications*, second edition, Benjamin/Cummings, Redwood City CA USA, 1993.

[20] J.M. Drake, W.W. Xie and W.T.Tsai. Document driven analysis: Description and formalization. *Journal of Object-Oriented programming*, Page 33-50, November 1992

[21] B. Myer. *Object-Oriented Construction*. Prentice Hall International 1988.

[22] J.C Huang. Detection of Data Flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, pp 226-236, May 1979.

[23] D.Kung, J.Gao,P. Hsia, J. Lin,Y. Toyoshima, and C. Chen . Change Impact identification in Object-Oriented software maintenance. *In proceeding of the International software maintenance*, pp. 202- 211, (1994).

[24] D.Kung, J.Gao,P. Hsia, J. Lin,Y. Toyoshima, and C. Chen . Desgin recovery of Object-Oriented programs. *In proceeding of Working Conference on reverse Engineering*, (Baltimore, MD), pp. 202-211, (May 1993).

[25] D.Kung, J.Gao,P. Hsia, J. lin,Y. Toyoshima, and C. Chen. A test strategy for O-O software. *In proceeding of computer software and application conference*, (Dallas, Texas), pp. 239- 244, (1995).

[26] D.Kung, J.Gao,P. Hsia, J. lin,Y. Toyoshima, C. Chen, Y. Kim and Y. Song . Developing an O-O software testing and maintenance environment. *In Communication of the ACM*, Vol. 38, No. 10, pp. 75- 87, (October 1995).

[27] J. Hattmann and D.J. Roberson. Revalidation during software maintenance phase. in *IEEE Conference on Software Maintenance*, pp. 70-80, (1989).

[28] 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).

[29] J. Laski and W.Szermer . Identification of Program Modification and its Application in Software Maintenance. *In Proc. IEEE Transactions on Software Engineering*, 18:1045- 52, (December 1992).

[30] N. Leung and L. White. A Study of integration testing and software regression at integration level . *In Proc. IEEE Conference on software maintenance*, pp. 290- 301, (1990).

[31] V. Rajlich, N. Damaskine, and P. Line. VIFOR: A tool for software maintenance. level . *In Software: Practice and Experience*, Vol. 20, pp. 67- 77, (1990).

[32] W. Teilelman and L. Masinter. The Interlisp programming environment. *IEEE Computer*, Vol. 14, No. 6, pp. 668- 675, (1988).

[33] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramarmoothy. The C information abstract system. *In IEEE Transactions on Software Engineering*, Vol. 16, pp. 325-334 (March 1990).

[34] N .Wilde and R. Hwitt. Maintenance supports for O-O programs. *In IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038-1044 (Dec 1992).

[35] N .Wilde and R. Hwitt. Dependency analysis tools: Reusable components for software maintenance. *In Proc. IEEE Conference on Software Maintenance*, pp. 126-131, (October 1989).

[36] S.C. Choi and W.Scachi. Extracting and restructuring the design of large systems. *In IEEE Software*, Vol. 17, No. 1, pp. 66-71, (January 1990).

[37] S. Lui and N. Wilde. Identifying objects in Conventional Procedural Languages: An example of Data Design Recovery. *In Proc. Conference on Software Maintenance*, pp. 266- 271, (1990).

[38] R.T. Croker and A.V Mayrhauser. Maintenance support needs for O-O software. *In Proc. Conference on Software Maintenance*, pp. 266- 271, (1990).

[39] L. Linz. *An introduction to formal languages and automata*. D.C Heath and company, (1990).

[40] H. Agrawal, J. Horgan, E. Krauser and S. London. Incremental regression analysis and testing. *In Proc. Conference on Software Maintenance*, pp. 348- 358, (1993).

[41] K.F Fischer. A test case selection method for the validation of software maintenance. *In Proc. COMPSAC'77*, pp. 421- 426, (November 1977).

[42] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression analysis and testing. *In 16th International Conference on Software Engineering* . pp. 211-222 , (May 1994).

[43] E. Siepmann and A.R.Newton. TOBAC: A Test case Browser for Testing O-O software . *In SIGSOFT software Engineering Notes*, Vol. 19, pp. 154- 168, (1994).

[44] F.K Gardne. RAYTRACER: Traceability for Software Engineering. *International Conference on Requirement Engineering. (Colorado Springs)*, pp. 94- 101, (1994).

[45] D.G Firesmith. Testing Object-Oriented Software. *Technology of Object-Oriented Languages and Systems (TOOLS 12 USA '93)*, pp. 407- 426, (1993).

[46] G. Canfora, F. Lanbile and G. Vissaggio. IESEM: Integrated Environment for Software Evolution Management. *International Journal of Soft. Eng. And knowledge eng.*, Vol. 5, pp. 49- 71, (1995).

[47] O.C.Z Gotel and A.C.W. Finkelstein, "An analysis of Requirement Traceability Problem," *Proc. Third Symposium on Assessment of Quality Software Development Tools.*, (1994) pp. 224- 232.

[48] C. Geldrez, "Testability in the O-O Software methodology," *Masters Thesis*, University of Ottawa, May 1995.

[49] E. Casais. Managing class evolution in O-O systems. *Technical Report, Centre Universitaire d'informatique, Geneve*, (1990).

[50] M. Lorenz. Object-Oriented Software Development t: A practical guide. *PTR Prentice Hall* , (1994).

[51] V.R. Basili. Viewing maintenance as Reuse-Oriented software development. *IEEE Software*, Vol 7,1 , pp. 19- 25, (1990).

[52] M.W. Evans. The Software Factory. *John Wiley and sons*, (1989).

[53] J. Jackson. A keyphrase based traceability scheme. *In Tools for maintaining traceability during design, IEE. Colloquium, Computing and Control Division, Professional group C1, Digest No.: 1991/180*, pp.2/1-2/4.

[54] Software Development environment (1991). *Software through pictures Products and Services Overview, IDE., Inc.*

[55] A.M. Davis. Software Requirement: Analysis and Specification. *Prentice Hall. Publishing ,1993.*

[56] P.G. Brown. Echoing the voice of the customer. *AT&T Technical Journal*, pp. 21-31, (March/April 1991).

[57] H. Kaindl. The missing link in Requirement Engineering. *ACM SIGSOFT Software Engineering Notes*, Vol. 18, No 2, pp. 30-39, (1991).

[58] M. Lefering. An Incremental Integration Tool between Requirement Engineering programming in the large. *In Proc. of the IEEE Symposium on Requirements Engineering, San Diego, California*, pp. 82-89, (January 1993).

[59] T. Smithers, M.X. Tang, and N. Tomas. The Maintenance of Design History in Ai-Based design. *In Tools for maintaining traceability during design, IEE. Colloquium, Computing and Control Division, Professional group C1, Digest No.: 1991/180.*

[60] J. Bowen, P. O'Grady, and L. Smith. A constraint Programming Language for Life-cycle Engineering . *In Artificial Intelligence in Engineering*, Vol.5, No. 4, pp. 206-220, (1990).

[61] S. Kirani and W.T. Tsai. Method sequence specification and verification of classes. *In Journal of Object-Oriented programming*, pp. 28-38, (October 1994).

[62] D. Kung, J.Gao,P. Hsia, J. lin, and Y. Toyoshima. Class firewall, test order , and regression analysis and testing. *In Journal of Object-Oriented programming*, pp. 51-65, (May 1995).

[63] A. Cimitile, F. Lanubile and G. Visaggio. Traceability based on design decision. *In Proc. Conference on Software Maintenance*, pp. 309-317, (November 1992).

[64] L.J. White. *Software testing and Verification.* Elsevier Science publisher B.V. 1991.

[65] B. Beizer. *Software testing techniques.* International Thomson publication Inc, 1990.

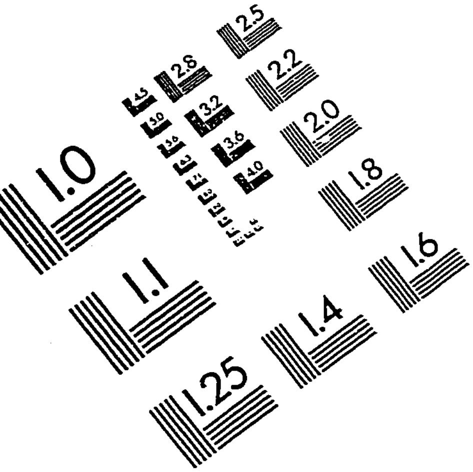
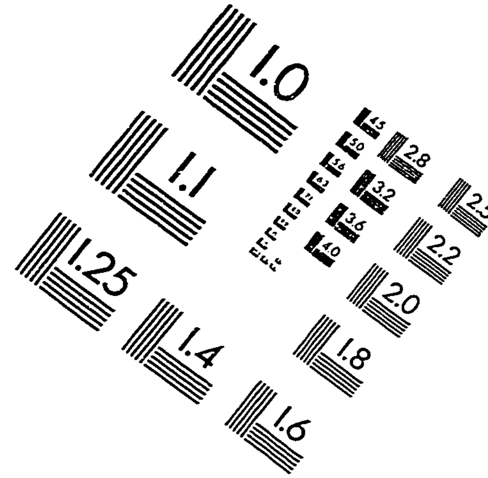
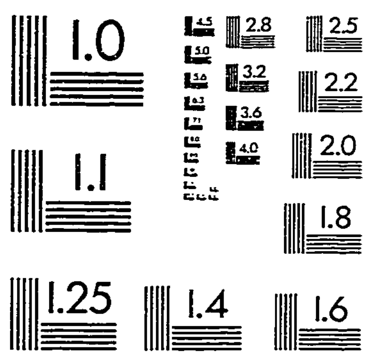
[66] J.V. Leeuwen. *Algorithms and complexity.* Elsevier Science publisher B.V. 1990.

[67] J.E. Hopcroft and J.D. Ullman. Set merging algorithms. In *J. Comput.* 2 (1973) pp. 294-303.

[68] ObjecTime reference manuals (Toolset guide, user manual, developer guide). ObjecTime Inc, 1994.
[69] J.P. Corrieveau, Object-Oriented software testing course notes, Carleton University, 1994.

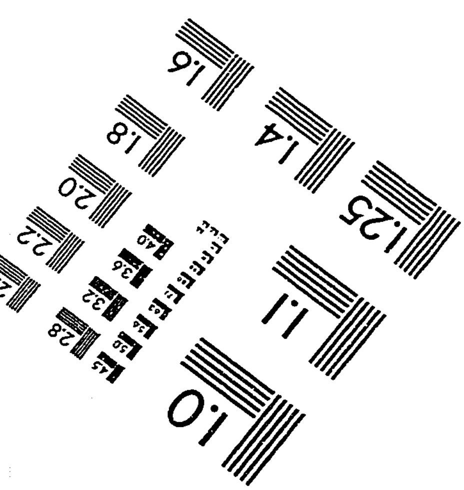
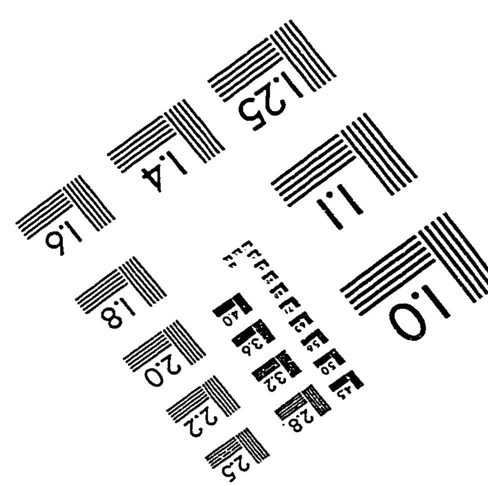
[70] M.Lorenz and.J.Kidd *Object-Oriented Metrics.*, *Prentice Hall Publishing* 1994.

IMAGE EVALUATION TEST TARGET (QA-3)

150mm

6″

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989