UML-Based Testing of Object Oriented Programs

by

Hong Ye

B.Eng., Zhejiang University, 1990

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming to the required standard

[redacted]

Dr. Issa Traoré, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Daler N. Rakhmatov, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Margaret-Anne Storey, Outside Member (Department of Computer Science)

[redacted]

Dr. Daniel M. Hoffman, External Examiner (Department of Computer Science)

Supervisor: Dr. Issa Traoré

Abstract

Software testing remains one of the most challenging and costly steps of software development process. Although object-oriented programming has become the dominant paradigm in industry, object-oriented software testing is mostly still in the research stage. Traditional software testing techniques are not well suited for object-oriented programs due to the powerful mechanisms, such as encapsulation, inheritance and polymorphism, characteristics of object-oriented programming. Hence, a need for the development of appropriate testing techniques for object-oriented applications becomes more urgent. This thesis introduces some test strategies based on a subset of Unified Modeling Language (UML) for testing object-oriented programs, more specifically, state diagrams and class diagrams. Corresponding coverage criteria are also presented. The approaches discussed in this thesis are implemented as a module of an existing UML-based verification tool called the Precise UML Development Environment (PrUDE).

[redacted]

Dr. Issa Traoré, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Daler N. Rakhmatov, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Margaret-Anne Storey, Outside Member (Department of Computer Science)

[redacted]

Dr. Daniel M. Hoffman, External Examiner (Department of Computer Science)

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work has been performed at the Department of Electrical and Computer Engineering at University of Victoria, Canada.

First of all I would like to thank my supervisor, Dr. Issa Traore, for suggesting the topic of this thesis and for the valuable guidance throughout my degree program. Without his support, this thesis could not have been completed. My thanks also go to Mr. Lawrence McGill for his reviewing and helpful comments, and the people at ISOT group for their time and assistance.

My family has provided me much support over the years, and I want to thank them for their encouragement and patience!

Chapter 1

Introduction

This thesis describes an investigation in the area of testing of object-oriented software. Much attention is focused on features in object-oriented programming language such as encapsulation, inheritance and polymorphism, which present many new challenges to testers. Some solutions for these challenges are presented herein. The goal of our work is to define appropriate test strategies for test data generation and test result evaluation based on the UML specification. In particular, defining test strategies based on UML state and class diagrams is conducted in this thesis. In addition, test coverage analysis and test automation are also taken into account. The test strategies defined in this thesis are implemented in Java and integrated into a prototype UML-based verification tool named the Precise UML Development Environment (PrUDE).

1.1 Motivation

Testing remains one of the most challenging and costly steps of the software development process. Especially for mission-critical systems, such as air traffic control and health information systems where failure is unacceptable [01], the cost of testing may be even higher. Although nowadays object-oriented programming has become the dominant paradigm in industry, object-oriented software testing is mostly still in the research stage [02, 03]. Traditional testing techniques, which are designed for testing procedural software, have been found to be inadequate for testing object-oriented

programs. The main reasons for that are the new challenges raised by inherent features from which the power of the object-oriented paradigm derives, such as encapsulation, inheritance and polymorphism. Thus, a need for developing appropriate testing techniques for object-oriented programs becomes more urgent. Among other issues underlying software testing is the knowledge that even though we can generate test cases based on sound methodologies, we may still question these test cases on how complete they are and whether they can uncover errors with an acceptable level of reliability. The main concern behind this is test coverage, which can increase the confidence that an implementation has been thoroughly tested after running the test cases generated.

1.2 Problem Statement

A number of researchers have reported that not all forms of traditional testing techniques are applicable and effective for object-oriented testing [17, 18, 19]. In traditional software testing, testing units are procedures or functions [01, 05]. Testing a procedure or a function usually consists of selecting representative set of input values and observing the corresponding outputs (e.g. test oracles) after the execution of a program. However, things become different in object-oriented programs since a class encapsulates state information in a collection of variables, also referred to as state variables, and has also a set of behavior represented by a collection of methods that operate on those state variables. Client objects can interact with a server object through its public interface, but they cannot see how its behavior and state are implemented. The unobservable nature of state makes it often impossible to decide whether an error has occurred.

Another significant difference between traditional and object-oriented programs is inheritance. In fact, inheritance is the biggest innovation in modern object-oriented programming and provides the most powerful mechanism for reuse. However, from the perspective of testing, one could point out that this may also provide the mechanism of mis-reuse by which anomalies can be propagated from a class to its derived classes. Therefore, when, what and how we should test or retest derived classes or the base class appears important. There are some other problems related to polymorphism and dynamic binding. Polymorphism permits instances of different types to be bound to a reference of another type according to the structure of inheritance. Dynamic binding means

implementation of a method is unknown until runtime. Polymorphism and dynamic binding present major challenges and make object-oriented testing extremely time consuming and costly.

The problems that this thesis will address are twofold: (1) define test strategies for test case generation and test result evaluation in order to find errors that are related to those new features in object-oriented software, and (2) define adequacy criteria to regulate test case selection and evaluate what level of coverage of code execution can be achieved.

1.3 Approaches

1.3.1 Specification-Based Testing

Traditionally, there are two main approaches to testing software: (1) structural testing or white-box testing, which is based on program implementation, and (2) functional testing or black-box testing, which is based on program specification. White-box testing is concerned only with testing the software products, i.e. code; it cannot guarantee conformance of the implementation to the specification. By contrast, black-box testing is concerned mainly with specification; it cannot guarantee that all parts of the implementation have been tested. In black-box testing, software is exercised over a set of inputs, and outputs are observed for correctness. It doesn't matter how those outputs are achieved. Both techniques have advantages and disadvantages. Table 1.1 compares white- and black-box testing techniques.

Although good results can be obtained by white-box testing, the gain in reliability of software is often limited. It is possible to write an apparently flawless program, but with resulting behavior differing from the program specifications. Hence, our primary interests are directed to black-box testing, also called specification-based testing [20, 21].

	Advantages	**Disadvantages**
White-Box Testing	• High percentage of code test coverage, almost every line of code can be tested • A number of tools support test automation • Test some complex segments of code	• Too many test cases when testing large-scale systems • Programs may not behave as expected in the specifications • Failure of testing may require a great deal of modification that can make testing unduly costly and time consuming
Black-Box Testing	• Exposes any ambiguities and inconsistencies in specifications • Test cases can be generated earlier: as soon as specifications are complete • More effective for testing large-scale systems • Test cases generated can be used for different programs, such as Java and C++ • Specification can be used to generate expected results for output checking, which can reduce costs significantly	• Test cases are difficult to design without clear specification • Some code or paths may not be tested • Few tools support it

Table 1.1 Comparison of White-Box and Black-Box Testing

1.3.2 A Test Model Based on the UML

In this research work, the UML [22] specification is selected as the test model to investigate some new error features related to object-oriented programs. We believe that the UML specification can provide a good representation for test case generation and test

result evaluation for object-oriented software. The selection of UML is motivated by the following considerations [15]:

- UML is an OMG standard and very popular specification language that is used to model modern object-oriented systems by many organizations in software industry. Test approaches based on UML may be widely accepted.
- UML is an object-oriented notation, hence the specific characteristics of object-oriented programs can be easily analyzed.
- The richness of UML, which provides nine different sub-models. Each of these sub-models covers a different view of the system and may also be used to generate test data for different testing levels, such as unit testing, system testing, integration testing.
- The existence of efficient commercial UML tools, for example Rational Rose [23], may also facilitate the use of this approach because the construction of the specification would no longer be problematic any more.

The UML specification, used as the test model, should be precise and describe system behavior rigorously, otherwise testing may still be flaw-based. Fortunately, several works on formalizing UML notations have been proposed or are under development [24, 25, 26, 27]. In previous work, Traore and Aredo have defined the formal semantics of UML state, class and sequence diagrams using PVS-SL (Prototype Verification System-Specification Language) [28, 29]. Detailed discussion of that is beyond the scope of this thesis, and interested readers are referred to [30, 31, 32].

As mentioned previously, UML has nine sub-models or diagrams, each of which models an object-oriented system from different points of view. Meanwhile, each of these diagrams can be also used for testing at various degrees and from different levels. Our objective is to investigate how we can define systematic test strategies for object-oriented programs based on a formalized version of UML sub-models. Our work focuses on the definition of efficient strategies for test cases generation by adapting and improving existing strategies. The UML state and class diagrams are among the most important of the nine UML diagrams. Accordingly, these two UML diagrams are initially investigated. UML state diagrams, used to model the dynamic behavior of an object, are suited to

define test strategies for class testing. For class diagrams, which contain a cluster of classes, we will focus mainly on integration testing based on the relations among these classes.

1.3.3 Completeness of Software Testing

One of the most difficult questions to answer when testing a program is deciding when to stop [03, 05]. In order to address this issue well, we need to do *test coverage analysis*, which is twofold: how complete the test cases generated are, and what percentage of the code has been exercised by executing the test cases. As mentioned earlier, a test case involves a collection of input values that drive the execution of a software program under specific condition. Test cases are run against software in order to reveal defects. Unfortunately, the entire domain of values for software testing cannot be exhaustively searched. A good selection of test cases should be complete and effective. Adequacy criteria are therefore defined for testers to evaluate whether the derived test cases are enough to detect the expected errors. If the test cases are not enough, the test criteria may also guide testers to identify which test cases are missing and which additional test cases should be generated to achieve good coverage.

A large variety of test coverage measures exist, but three fundamental measures from which most other coverage criteria are derived are worth mentioning: (1) *statement coverage*, (2) *decision coverage,* and (3) *condition* coverage. *Statement coverage* can report whether each statement is encountered, but it is insensitive to logic operators, such as && (AND) and || (OR). *Decision coverage* reports whether boolean expressions tested in a program (e.g. if-statement or while-statement) evaluate to both true and false. *Decision coverage* is also known as *branch coverage* and *all-edges coverage.* Condition coverage reports the true or false outcome of each boolean sub-expression separated by logic-or and logic-and, and measures each sub-expression independently. Criteria for adequacy is based mainly on the combination of these three coverage criteria to evaluate the test cases generated from the test strategies defined in this thesis.

1.3.4 Test Automation

Test automation is also one of the important aspects we consider in software testing. Over the past years, time-to-market has been the driving force behind many software development organizations. This has also increased the pressure on testers, who are often being asked to test more code in less time. Test automation improves dramatically their productivity.

Test automation is based on software that automates any aspect of testing a software system [03]. Without tool support, testing is always tedious and of low productivity. Our objective is to develop a test environment for object-oriented programs involving all the aspects of testing activities such as test planning, test case generation, test case execution, test result review and test coverage analysis. In this thesis, we focus mainly on defining some strategies for test case generation and test results evaluation, and also defining adequate criteria to evaluate the test cases and their execution. Then, we implement these test strategies as a specific component of the PrUDE tool suite.

1.4 Contribution

In this thesis, we propose a transition test strategy [15], which is based on a UML statechart diagram, and suited for primitive as well as object variables. The proposed transition test strategy extends the conventional domain analysis technique. We propose more specifically a new approach that allows analysis of boundary conditions involving primitive as well as object variables.

We also introduce relational test strategies to test class association and inheritance based on UML class diagrams.

We define a set of functional coverage criteria that can be applied to evaluate the transition and relational strategies.

1.5 Thesis Outline

The remainder of this thesis is organized in chapters dealing with some of the steps of developing comprehensive and efficient solutions to some of the challenges involved in object-oriented programs testing. It is divided into six major parts. The first part (chapter

2) describes background material. Part two (chapter 3) presents a test strategy based on UML statechart diagrams. Part three (chapters 4, 5, and 6) introduces some test strategies based on UML class diagrams. Part four (chapter 7) describes test coverage criteria. Part five (chapters 8 and 9) shows the practical use and implementation of the test strategies defined in this thesis. The last part (chapters 10 and 11) summarizes our results and discusses related work.

Chapter 2 – Background

Chapter 2 presents background material underlying the specification-based testing process, software testing metrics and process, UML diagrams as test model and domain testing.

Chapter 3 – A Test Strategy Based On UML State Diagrams

Chapter 3 presents the state transition test strategy, which is based on UML statechart diagrams.

Chapter 4 – Testing Based On UML Class Diagrams

Chapter 4 introduces some test strategies based on UML class diagrams that focus mainly on testing relationships between classes. In particular, the investigation of errors related to two important mechanisms for constructing new classes in object-oriented languages, class association and class inheritance, is made.

Chapter 5 – Testing Class Association

Chapter 5 presents a relational test strategy for testing class association.

Chapter 6 – Testing Class Inheritance

Chapter 6 describes some of the problems encountered in class inheritance hierarchies and corresponding test strategies.

Chapter 7 – Test Coverage Analysis

Chapter 7 defines adequate criteria and metrics for test coverage.

Chapter 8 – PrUDE

Chapter 8 presents the implementation of the test strategies as defined in this thesis in a prototype tool named PrUDE.

Chapter 9 – Example – A Patient Document System (PDS)

Chapter 9 presents a case study – Patient Document Service in order to show the practical use of our strategies.

Chapter 10 – Related Work

Chapter 10 presents related research on testing object-oriented programs.

Chapter 11 – Conclusion

Chapter 10 makes some concluding remarks and gives some directions regarding future work on testing object-oriented programs.

Chapter 2

Background

In this chapter, we start by presenting a brief overview of UML and some background information on software testing and test metrics. Then, the specification-based software testing process is reviewed briefly. A popular testing technique, namely domain testing, is also discussed. All these principles and techniques represent important foundation for the testing strategies proposed in this thesis.

2.1 Overview of Software Testing: Process and Metrics

2.1.1 Life Cycle Testing

Software testing is the process that tries to explore and uncover evidence of flaws and explore flaws in software systems. These flaws may result from various reasons such as mistakes, misunderstandings, and omissions occurring during any phase of software development. Testing is important because it substantially contributes to ensuring that software applications perform as intended.

Software testing, as an integral part of the software development process, used to be placed at the latter phases immediately before operation and maintenance in the traditional software development life-cycle process shown in Figure 2.1. For most projects, testing after coding is the only verification technique used to determine the adequacy of the system. If testing is restricted to a single phase, there is the potential that errors with significant and costly consequences may occur. Studies have demonstrated

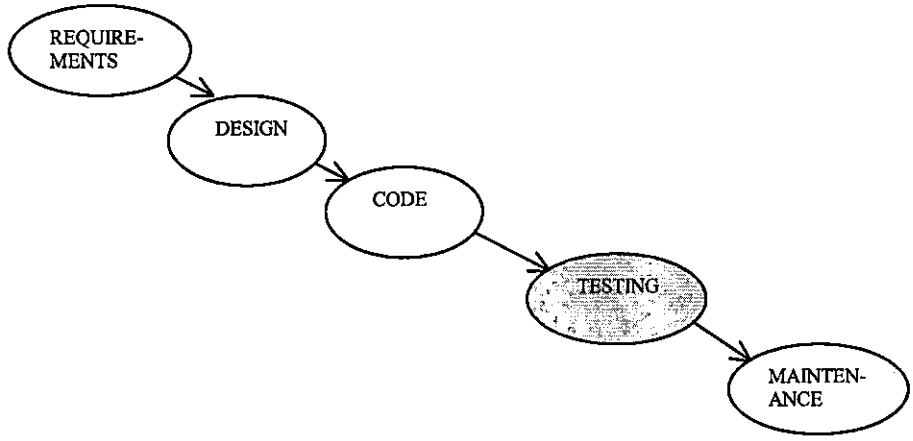


Figure 2.1 Traditional Software Life-Cycle Development Process

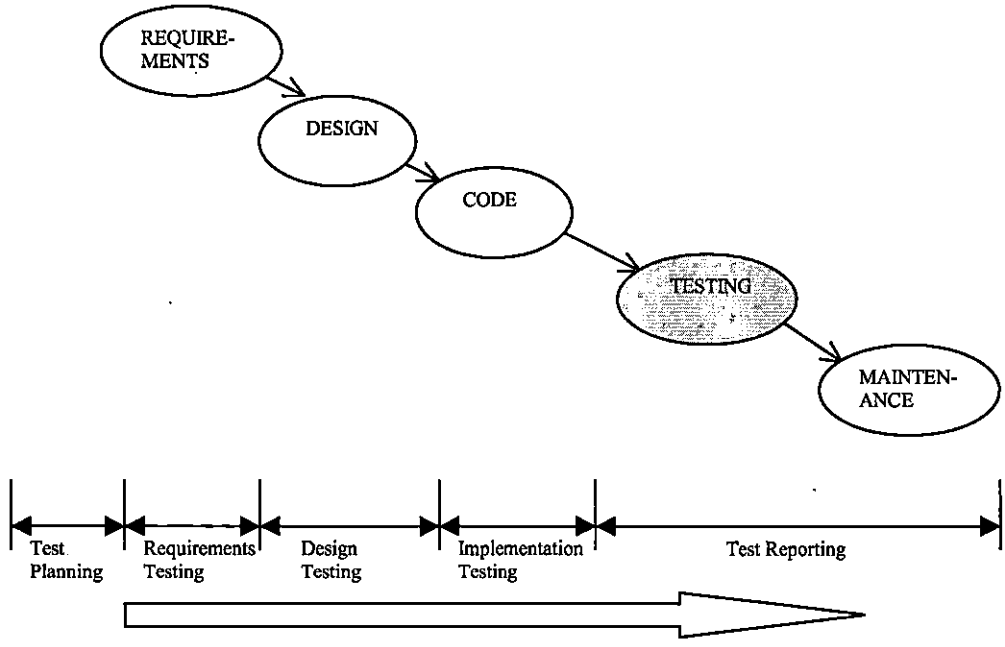


Figure 2.2 Software Testing Process

that it is at least 10 times more costly to correct an error after coding than before, and two-thirds of the errors occur in the design phase, before coding [56]. Hence, testing should start early and must not be isolated to a single phase in the development process if

lower costs and higher quality systems are the main concerns of an organization from the testing management point of view. Moreover, testing activities should be incorporated into each phase of software development showing different verification requirements. The concept of a life-cycle testing process that may span the whole life cycle of development is proposed and introduced by Perry [56]. Life-cycle testing falls roughly into the five phases shown in Figure 2.2, paralleling the software development process: test planning, requirements testing, design testing, implementation testing, and test reporting, each of which are reviewed in the following paragraphs:

Test Planning. Test planning can start once a development project starts. Developing a test plan is similar to any software planning process. In this stage, the potential members of the test team are initially identified, and the objectives of testing are determined. Developing a test plan involves the following activities: specifying the types and scope of testing activities to be performed on each part of the application and the group who is responsible to produce them; outlining the test environments such as software, hardware, testing tools needed to conduct the tests; and, developing a preliminary schedule for testing activities. Of course, the costs of testing need to be estimated as well. The test plan can be written at a high level. Its main purpose is to ensure that a systematic approach to testing is established, and that the testing is adequate to verify the functionality of the software product.

Requirements Testing. The requirements phase is a user-oriented phase in software development process. This phase has always been critical in the implementation of software systems. Requirements must be clearly described or defined without any room for misinterpretation and ambiguities because any errors could be compounded later in the development process. Testing the requirements ensures that the requirements are recorded or documented properly according to user needs, and adequately address the business problem. Testing activities during this phase may comprise the mediation between end users and developers, the identification of business process risks, and the review of documentation, which captures business rules or user requirements. If the requirements contain no errors, the costs of testing would be significantly reduced.

Design Testing. In the design phase of the software development, requirements are converted into an abstract document, often called the design specification, which serves

as the basis for building the final software product. There are different ways and tools currently used for describing or modeling the system structure during the design phase. Moreover, the design process could result in an almost infinite number of possible solutions. Thus, the methods and efforts involved in design phase testing are diverse and depend strongly on the kind of specification language and tool used for modeling the system. Design testing may be static by using raw design models or dynamic by using simulation. Whatever approach is chosen, the overall goal of design testing is to ensure that the design is complete, accurate and matches with the requirements.

Implementation Testing. Because many of the implementation tasks are tedious and repetitive, they are error prone. The main goal of implementation testing is to ensure that the design specification has been correctly implemented. Testing in this phase is highly technical and specific, and it requires testers with strong programming experience. Implementation testing usually consists of two approaches: static testing and dynamic testing. Static testing uses techniques such as review and inspection to uncover defects; dynamic testing tries to verify the functionality of programs through their execution. In this phase, testers may face thousands of test cases, and execute programs many times; therefore, implementation testing may be time-consuming and costly.

Test Reporting. Reporting the test results is also one of the most important phases in the software testing process, and is mostly for management needs. The project test status report mainly addresses the issues that are of high interest to management such as when the final product will be released, whether or not enough testing has been achieved, and how reliable will the system be. In order to do so, the best way is to use test metrics (see Section 2.2.3) to evaluate test approaches, and define work products[1] to report what has been achieved for current testing.

2.1.2 Test Levels

During development the errors that may occur in a software system are diverse and complex. No single test technique can cover all kinds of errors. It is necessary for testing activities to distinguish errors and to define different and appropriate test strategies to

[1] Typical testing work products usually include test cases, failure reports, plans, logs, test scripts etc.

deal with them. Program testing may involve different levels traditionally designated as *unit*, *integration* and *system testing*:

- **Unit Testing.** Individual modules are tested to ensure that they work properly, but no guarantee is given that when the modules are tested as a whole, they won't fail.
- **Integration Testing.** A collection of related functions that can be viewed as a component or subsystem is tested to ensure that the component or subsystem works properly
- **System Testing.** Consists of testing the whole system after all the components or subsystems are combined into the final product. System testing is usually performed in the real deployment environment.

2.1.3 Test Metrics

Test metrics are the measures that are used for evaluating the effectiveness of individual testing strategies or techniques and the complete testing process. Mostly, test metrics can measure the overall test *quality* from a *quantitative* point of view in software testing just as blood pressure is widely used in the medical field to predict the probability of heart attack or stroke. Quantitative management of software quality is a broad area. There are many kinds of approaches or metrics used to evaluate quality of testing. Traditional metrics, such as McCabe's cyclomatic complexity (MCC) and number of lines of code (LOC), are usually used for functional or procedural programs testing. Several new metrics geared specifically to object-oriented programs such as depth of inheritance tree (DIT), have been developed recently; see [60][61] for a survey on these metrics.

As mentioned previously, the objectives in each phase of the testing process are different. This may result in the different use of test techniques, which result in the different use of test metrics as well. In fact, one metric is usually insufficient to conclusively evaluate everything: multiple metrics must be applied. How to choose proper metrics for a project or which metrics are practically useful for which test phase seems to be the main concern. For example, in the test-planning phase, the cost metrics evaluating test budget is very important. In requirement or design testing, object-oriented metrics are mostly suitable for measuring the complexity of a project in order to decide how much effort should be devoted to testing. In program testing, lines of code are

obviously among the most important metrics for test coverage. Test metrics are particularly important in the test report phase. The use of test metrics is a powerful means to generate concise and clear test results for reporting the whole test process. Such reports can enhance quantitative management decision-making and help to determine whether the final software products are ready for release. Unlike technical reports with considerable detail, test metric reports should be short and easy to read.

2.2 Specification-Based Software Testing Process

Most software professionals argue that requirements and specifications are critical to successful system delivery and support. In a well-organized software project, the specification should be clearly recorded and made available from the early stages of the software development to the end. However, for a long time, specifications were used only by developers to write code. In [62] Poston suggested that software testing should start with a written or modeled specification in order to reduce the costs of testing. Specification-based testing consists of identifying behavioral difference between the system models and the actual implementation.

Specification-based testing process, illustrated in Figure 2.3, parallels a typical software development process [62]. The testing process starts with a well-modeled specification that describes the behavior and characteristics of a software system to be developed. Test cases are created from the specification, executed and evaluated. Specification-based testing may cover all levels of testing including unit, integration or system testing. We review briefly in the sequel the main activities involved in a specification-based test process:

- **Specification validation and verification (V&V)**: the specification, if used as a basis for software testing, must be unambiguous, consistent and complete. In order to do so, rigorous verification of the specifications should be conducted. This may consist of informal reviews or formal verification, or a combination of both approaches. The purpose of the V&V is to ensure the correctness of the specification and its conformance to the customer requirements.
- **Test case generation**: it is an essential phase of the specification-based testing process because other activities, such as test case execution and test result

evaluation, depend on the generated test cases. Test cases are derived from the valid and correct specifications. The expected results corresponding to the test cases can also be generated at the same time.

- **Test case execution**: test case execution waits until the code is completed. The IUT (Implementation Under Test) is run against inputs (generated and documented test cases), and outputs known as actual outputs are produced. In order to find whether the test passes or fails, the actual outputs are compared with expected outputs (e.g. test oracles).
- **Test result evaluation**: test result evaluation includes the evaluation of overall testing techniques and the quality of software products. This can be achieved by evaluating each test case to determine the pass/failure of that test case, and the effects of exercising all the test cases together. For example, a type of test quality evaluation is known as *test coverage analysis*. In test coverage analysis, adequate criteria are defined to evaluate the completeness of test cases, and the thoroughness of code execution.

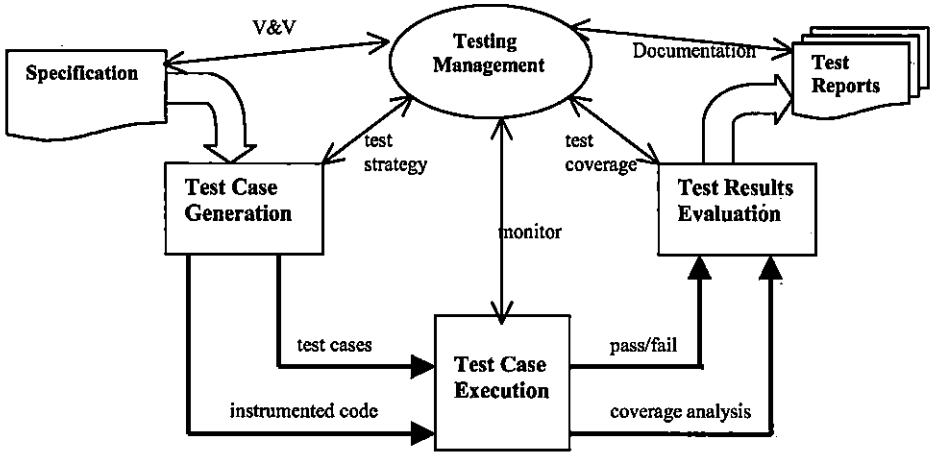


Figure 2.3 Specification-Based Software Testing Process

2.3 Using the UML in Software Testing

2.3.1 What is the UML

The UML is a notation for specifying, visualizing and documenting modern object-oriented software systems. It was originally developed by Grady Booch, Ivar Jacobson and James Rumbaugh, and was standardized by the Object Management Group (OMG) in January, 1997 [16]. Since then, UML has rapidly been accepted in the software industry as the standard modeling language.

A significant benefit of UML is a graphical notation, which is used to describe the blueprints of an object-oriented software system. This makes for much easier communication between designers and developers in an organization, or in different development teams. Also UML plays an important role in the modeling process of large and complex software systems. UML consists of nine types of diagrams, which can be divided into two categories: four types of diagrams describe static application structure, and five describe dynamic behavior.

- **Structural Diagrams:** include Class Diagrams, Object Diagrams, Component Diagrams, and Deployment Diagrams.
- **Behavior Diagrams:** include Use Case Diagrams, Sequence Diagrams, Activity Diagrams, Collaboration Diagrams, and Statechart Diagrams.

A *Use Case Diagram* is used to document and capture user requirements; *Class Diagrams* and *State Diagrams* model class structures and their dynamic behaviors, which may result in the implementation of a software product, i.e. code. An *Activity Diagram* illustrates the dynamic behavior of a system as well, and is used to model control flow from activity to activity. *Sequence diagrams* and *Collaboration Diagrams* describe interactions among classes in terms of message exchange over time or a series of sequenced messages. A *Component Diagram* describes the organization of physical software components including source code, binary and executable code. A *Deployment Diagram* describes hardware topology and software components deployed on particular hardware or network.

2.3.2 Testability of UML Models

UML models are basically developed and used when building object-oriented software systems. The UML provides a notation for expressing OO models, but it doesn't prescribe any specific process about how to use the notation. The flexibility and extensibility of UML models make them appropriate for software testing and test case generation. There are two main concerns with the use of UML test models: firstly, the identification of the elements of UML diagrams that can be used for test design, such as the constraints used to express business rules or system requirements. Secondly, the selection of the testing levels (e.g. unit, integration and systems); UML diagrams can be potentially used in one or several of these levels. The nine different types of UML diagrams as mentioned previously provide the capability to explore static structure, dynamic behavior and physical deployment of a software system. Test design based on them can also cover various aspects of the software system from a testing point of view. We revisit, in the following, each of the nine diagrams, and discuss how they can be used for testing:

Class Diagrams and Object Diagrams. Class diagrams describe the static structure of a system; they are mostly used to represent classes of entities with common characteristics and the relationships between these classes. Object diagrams are closely linked to class diagrams. Just as an object is an instance of a class, an object diagram could be viewed as an instance of a class diagram. Class diagrams may be used to develop interesting integration testing strategies. This may consist of testing relationships, such as associations and generalizations, and identifying and verifying the properties and constraints of a relation. For example, are association multiplicities consistent? Do all subclasses implement the *is-a-kind-of* relation properly? We may also pay attention to constraints between relations, or perform traceability testing between class and sequence diagrams by tracing whether every message appearing in the sequence diagram are defined as a method in the appropriate class.

Component Diagrams. A component diagram describes the organization of the physical components of a system. Component diagrams are good candidates for integration and system testing. Components can be grouped together to form a new

component or subsystem. When testing components or component-based software, test case generation based on UML component diagrams is usually the best choice because in most cases the source code of the components delivered by vendors is unavailable, and only interface specifications are provided. Hence, many white-box techniques will not be applicable. Since the functionalities of a component are provided through its interface, testing the services of the interface (e.g. testing operations in the interface through message sending) based on information in a component diagram seems to be the main concern.

Deployment Diagrams. Deployment diagrams depict the hardware and network topology on which the software components are deployed. Deployment diagrams provide guidance in software delivery and installation. Therefore, they are suitable for system testing. Whether any software or software components described in a deployment diagram can be executed properly in a real system environment will be tested. We may also investigate how deployment diagrams can be used for test case generation in distributed testing because they are instrumental in modeling the deployment and maintenance of distributed software models.

Use Case Diagrams. Use case diagrams model the functionality of a system, and are good sources of integration testing and system testing as well. Use case diagrams describe scenarios that are instances of a use case. Therefore, from a testing point of view, a good way to start is to identify the scenarios involved in a use case. Each scenario specifies a flow of events. Depending on how deeply you want to test, test cases may be generated for each flow. Unfortunately, use cases in a use case diagram are described at a very high level, and quite often important operational variables corresponding to the inputs for each flow are not precisely described by the diagram. Sequence and collaboration diagrams complement use cases by providing a more detailed and accurate definition of the scenarios involved.

Sequence Diagrams and Collaboration Diagrams. Sequence diagrams describe interactions among classes in terms of an exchange of messages over time. Collaboration diagrams describe interactions in terms of communication links among classes. Both are particular kinds of interaction diagrams. Because sequence and collaboration diagrams are semantically equivalent, one can convert one diagram into the other without loss of

information. Their contribution to testing is at the integration testing level although they may also be used at the module level. Testing may focus on tracing the sequences of messages among objects; more specifically, all end-to-end paths (sequences of messages) should be identified and exercised.

Statechart Diagrams. A statechart diagram describes the behaviour of a class in response to external stimuli. Statechart is one of the more appropriate for test case generation among the nine UML diagrams, hence, several test strategies based on state diagrams are available in the literature [03] [51] [52] [53] [54]. Most of these approaches are based on finite state machines (FSM). Testing may consist of exploiting state information by connecting it with pre and post conditions, and invariants. State-based testing is mostly conducted at the module level. At the system level, we may also consider combining local state diagrams to form a global one that can serve for system testing.

Activity Diagrams. An activity diagram illustrates dynamic behavior by modeling control flow from activity to activity. Because the model borrows ideas from flow charts and state transition diagrams, activity diagrams use some of the same modeling conventions, and can be viewed as a special kind of state diagram. For testing, this type of diagram can be used to develop test models for control flow-based techniques, such as logic-based testing, since activity diagrams support all elements of a basic flow graph. In addition, if an activity diagram supports concurrent action states, it may also be of interest in developing corresponding test strategies as concurrent models. Since an activity in the activity diagram represents an operation on some class in the system that results in a change in the state of the system, activity diagrams are usually good for integration and system testing.

2.4 Domain Testing

2.4.1 Overview

The domain testing model was first proposed by White and Cohen [07] for software testing in the early 1980's. Basically, a program is divided into different execution paths, so-called control flows. In order to ensure that a program is running under the right path,

a path condition, usually a predicate expression, must be explicitly specified. In domain testing, domain analysis provides the definition of a domain corresponding to the path and its boundaries corresponding to the path conditions in a program, and the domain testing fault model reveals anomalies indicated by incorrect path conditions.

The domain and its boundary conditions can be defined based on either requirements (specifications) or programs (source code). No matter what is used, defining a domain and its boundaries is the central step. A well-defined domain should always be complete, precise and unambiguous, and its boundary conditions should be consistent with the constraints from requirements or programs. By contrast, an ill-defined domain due to missing boundaries and domain overlapping, for example, could misguide the evaluation of the test result or cause additional errors, regardless of finding some program defects. Once a domain is defined, test values can be quickly chosen in terms of On-Off and In-Out point selection criteria [07].

2.4.2 Terms and Concepts

Domain

A domain is the collection of all possible input values for the program under test.

Sub-domain

A sub-domain is a subset of a domain or a partition of the domain based on predicate expressions (also called boundary conditions). A sub-domain may correspond to an execution path in the program. For example, suppose that a fragment of the program under test is given as follows:

```
...
x: float
if x>=0 then
y = √x
else ...
```

Assuming that there is only one input variable, the domain will be a one-dimensional domain corresponding to (-∞, +∞). Based on the two possible paths of the program execution, it is necessary to define two sub-domains D1 with boundary (-∞, 0) and D2 with boundary [0, +∞). D1 and D2 correspond to two paths -- *else* and *then*, respectively.

If a selected test value is in domain D1, the program executes the *else* path, otherwise, the *then* path is executed.

On and Off, In and Out points

The definition of test data using domain analysis consists of identifying special points in the domain referred to as On, Off, In and Out points. An **on point** is a value that lies on a boundary. An **off point** is a value not on a boundary. An **in point** is a value that satisfies all boundary conditions and does not lie on a boundary. An **out point** is a value that satisfies no boundary conditions and does not lie on any boundary. For example, Figure 2.4 describes a two-dimensional domain defined by the following boundary conditions: $x \leq 100$ and $y-x \leq 15$ for a function $y = x+15$. The two sub-domains defined are SD1, striped area including two bold lines (boundaries) and SD2, outside of striped area. Among all the points indicated, A and E are **on points**, C, D and B are **off points**. C is an **in point**, B and D are **out points**.

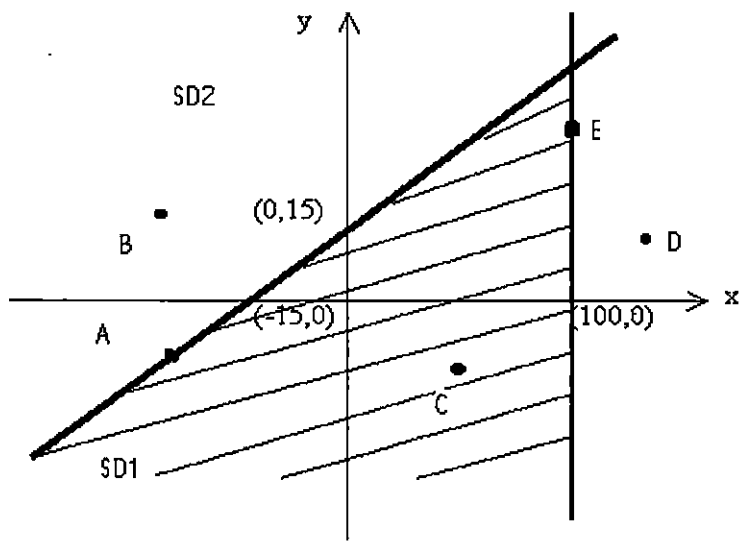


Figure 2.4 On and Off, In and Out Points

2.4.3 Domain Matrix

The results of the domain analysis are expressed in a domain matrix, which consists of a table used to build a complete test suite. A domain matrix may be either a column-like or row-like table. Table 2.1 shows a column-like table for the above example.

Column *Variable* lists all the input variables in a domain, column *Condition* indicates the boundary conditions for each variable where the term *Typical* in this column means no restrictions for that variable, and column *Type* specifies the kind of points. The remaining columns labeled *Test Cases* correspond to the test values generated. For example, the first test case (number one) is x=100 and y=113. Row *Expected result* indicates whether the test cases are either accepted or rejected.

Variable	Condition	Type	Test Cases			
			1	2	3	4
x	x<=100	On	100			
		Off		101		
	Typical	In			99	99
y	y<=x+15	On			114	
		Off				116
	Typical	In	113	113		
Expected result			Accept	Reject	Accept	Reject

Notes: *Accept: IUT (Implementation Under Test) accepts this value and produces correct results.

**Reject: IUT (Implementation Under Test) rejects this input.

Table 2.1 Domain Matrix

2.4.4 Limitations

Not all types of anomalies or errors in a program can be revealed by domain testing. Domain testing has limitations, as do other software test techniques. The following are some of the main drawbacks. First, domain testing is generally expensive requiring the selection of a relatively large number of ON and OFF points. When the number of variables becomes large, the actual number of test cases could be explosive. Secondly,

domain testing does work well with predicates that include only AND operators, but has trouble with those predicates that may include OR operators. If a domain is defined by a predicate that contains OR operators, its subdomains can be separated or partially overlapped. Thirdly, loops may also restrict the usefulness of domain testing since loops may lead to different predicate expressions, and in turn, the corresponding domain boundaries are changed.

Chapter 3

A Test Strategy Based On UML State Diagrams

UML state diagrams are usually used to model the dynamic behavior of an object. They are therefore well suited for performing tests at the class level. This chapter presents a transition-based test strategy for object-oriented testing based on UML state diagrams. Both primitive variables and object variables are considered when using this test strategy.

3.1 Overview of UML State Diagrams

UML state diagrams are used to model the dynamic behavior of any modeling element, such as a class, a use case or an entire system [16]. A UML state diagram is mainly used to show the sequence of states an object can have during its lifetime. So it is appropriate for developing test cases at the class level. Figure 3.1 depicts a UML state diagram for a network reconfiguration protocol.

A UML state diagram comprises of states and transitions. A state is a condition or situation during the life of an object in which it satisfies some conditions, performs some activities and waits for some events. States can be classified as either simple states or composite states. Composite states contain other states as sub-states. UML state diagrams are based on the paradigm of a hierarchical state machine. A composite state can be further classified as a sequential state (*or*-state) or a concurrent state (*and*-state). In Figure 3.1, the root state *NetworkStatus* is a sequential state that consists of the direct sub-states *Init, Electing, ErrorDetected* and *LeaderElected.* When a sequential state is

active, only one of its .direct sub-states can be active at a time. However, when a concurrent state is active, all of its direct sub-states are active simultaneously. State *Electing* is an example of a concurrent state.

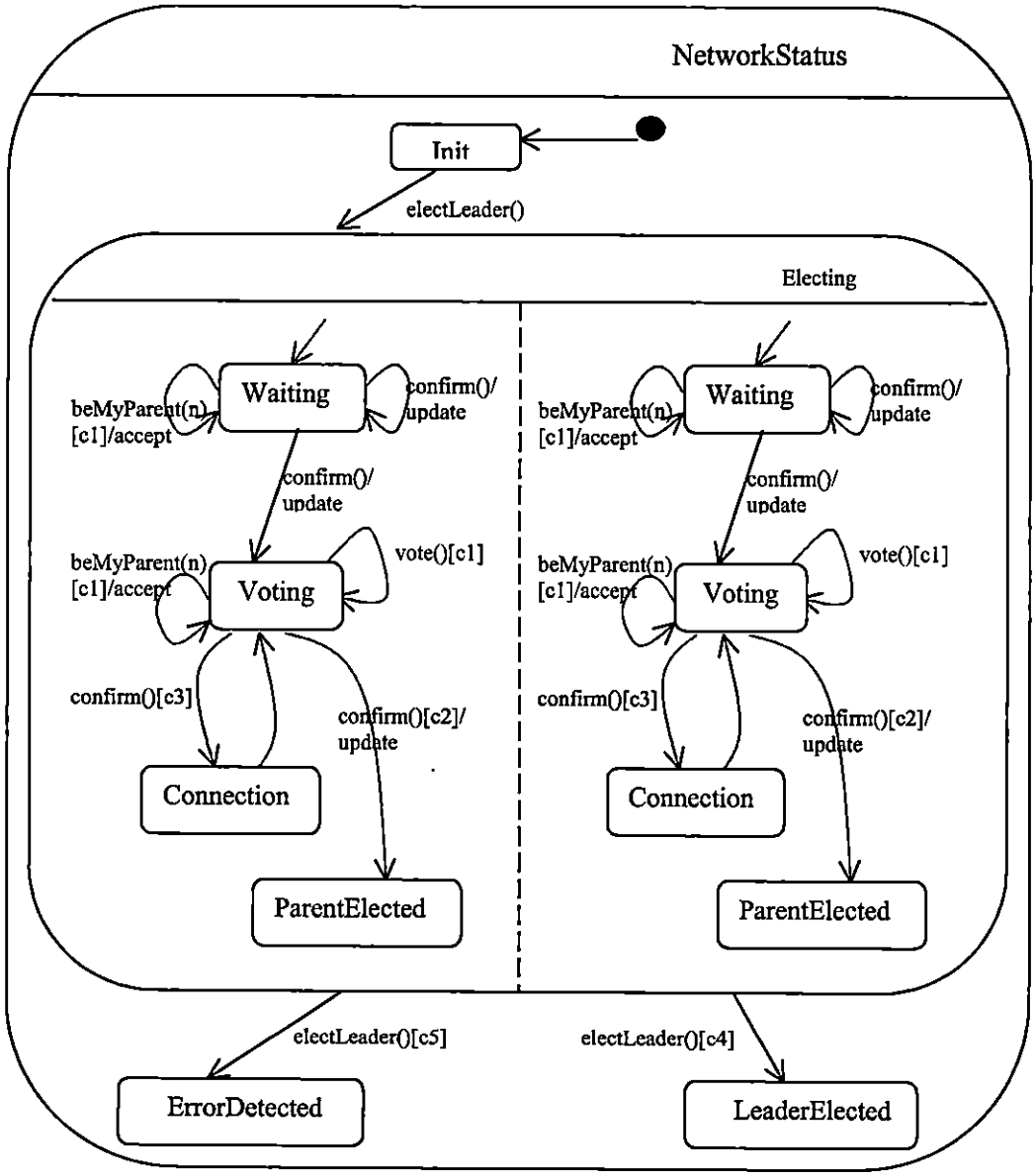


Figure 3.1 UML State Diagram for a Network Reconfiguration Protocol

A transition is a relationship between two states indicating that an object in the first state (source state) can enter the second state (target state) when a specified event occurs and some guard conditions are satisfied. An event is an occurrence usually corresponding to a message call that can trigger a state transition. For example, in Figure 3.1, the event *electLeader()* may trigger either of two different target states, *ErrorDetected* and *LeaderElected*, from the composite source state *Electing*. If the guard condition *c4* is satisfied, the object will enter the state *LeaderElected*. Otherwise, if the guard condition *c5* is satisfied, the object will enter the state *ErrorDetected*. A self-transition is a transition in which the source and target state are the same. The transition *beMyParent* is an example of self-transition.

3.2 Rigorous Semantics for UML Statechart

A formal semantics for UML statecharts using PVS [28, 29] has been suggested by Traore [30], and implemented in the PrUDE tool. The whole PVS semantic model for a given UML statechart diagram consists of three generic PVS theories named *AbstractSyntax*, *WellFormedness* and *FormalSemantics* that describe respectively the abstract syntax, the well formedness rules, and the actual semantics. The detailed definition of the three theories is given in [30]. The proposed semantic model defines a set of predicates for the formal representation of the primitive elements involved in a statechart diagram. More specifically, states and actions are precisely defined by predicates function of the instance variables of corresponding class. The predicate associated with a state corresponds to a condition that must hold for the state to be active. The predicate of an action corresponds to the action postcondition. Guard and state predicates are defined as function of a record type, named *V*, whose fields correspond to the attributes of the class related to the statechart diagram:

T1, T2, ...,Tn : TYPE

V : TYPE = [# a1: T1, a2: T2, ..., an: Tn #]

where a1, a2, ..., an are state variables in a class with respective types T1, T2, ..., Tn. The predicate associated with an action corresponds to its postcondition, which is defined as a function of another record type, named *VC*, which combines both the current and future state information.

VC : TYPE = [# current: V, next: V #]

A guard condition is naturally defined as a predicate expression of the instance variables.

The semantics of a UML statechart diagram is defined as a pair consisting of a root state and a set of transitions. A transition is a five tuple consisting of a source state, a target state, an event, a guard condition and an action. To fire a transition, two steps are needed: a transition is firstly enabled, and then, the enabled transition is fired. If the event instance generated matches its trigger, its guard condition is true and its source state is active, a transition is enabled, and eligible to fire. Firing a transition will activate its target state and execute its action. These steps are formalized in [30] by defining two predicates named *enabled* and *fired* as follows:

v, v1: VAR V; vc: VAR VC ; e: VAR Event; tr: VAR Transition

enabled(e,tr,v): bool = pred (source(tr))(v) AND
(trigger(tr) = e) AND
pred(guard(tr))(v)

fired(tr,v,v1): bool = pred(target(tr))(v1) AND
pred(effect(tr))(vc) AND
where vc = (#current:= v, next:= v1#)

3.3 Transition Test Strategy

3.3.1 Test Expressions

A UML statechart can be used in practice to describe the sequence of states through which an instance of a class (i.e. an object) evolves during its lifetime, as well as the sequence of messages it sends and/or receives. The messages exchanged during that lifetime correspond to method calls, i.e. events, and are associated to transitions between states. Therefore the actual execution of a method is closely related to the firing of a corresponding transition in the statechart specifying the class behavior. Since a method can be executed under various circumstances, several transitions can be associated with a method, each corresponding to a particular condition or circumstance. A natural test model at the class level may thereby consist of the set of transitions involved in the

associated statechart diagram. Using that test model, we can define a transition-based test strategy that we'll call the transition test strategy in the rest of this thesis.

The transition test strategy is a class-based test strategy that focuses on the test of the methods involved in a class. It allows the generation of test cases at the method and class level. The activation of a transition involves two predicates, *enabled* and *fired*, as defined above. Predicate *enabled* defines the enabling condition for the transition, and predicate *fired* defines the outcome of firing the transition. These pair of predicates can be used to define a pre-post condition pair for the transition. Given the transition depicted in Figure 3.2, the pre-condition for method m() consists of the conjunction of the source state predicate, and the guard condition:

pre-m: pred (S1) ∧ pred (c)

the post-condition for method m() consists of the conjunction of the target state predicate, and the action predicate:

post-m: pred (S2) ∧ pred (a)

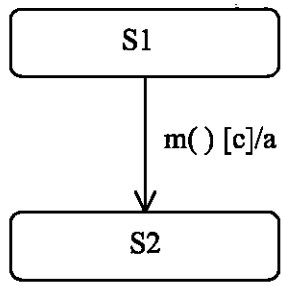


Figure 3.2 Transitions

The pre/post-condition pair derived from a transition is actually a partial pre/post-condition for the method involved. The global pre/post-condition for a method corresponds to the combination of the pre/post condition pairs associated with all the transitions involving that method. The pre and post conditions identified are functions of the attributes of the class under test and of the method parameters, some of which may be of a complex data type.

3.3.2 Refinement and Decomposition

Next, we need to convert the expressions obtained into executable test assertions by refining them. The preconditions are broken into disjunctive normal form (DNF). The decomposition approach used is similar to the one adopted in [50]. The purpose of this decomposition is to eliminate the disjunction (e.g. $\vee$) operator from the expression of a precondition. For example, given a relation $(a \wedge b) \vee (p \wedge q)$, the decomposition consists of transforming this expression into two disjoint cases: $a \wedge b$ and $p \wedge q$. Each DNF expression is analyzed separately using the domain testing technique in order to generate the test cases, which can be collected using a domain matrix. Difficulties arise mainly when DNF expressions involve object or complex variables, in which case we build a decision tree based on the attributes structure of the object variables involved (see Section 3.4).

Test execution starts using individual methods (e.g. m()) corresponding to transitions in a statechart diagram. We create a fresh object, and set its source state using the test values generated. Then, after the method execution, we observe the outcome of the target state. Object state can be set and observed by using mutator and accessor methods (e.g. set/get methods). Class testing is conducted by testing all the methods involved in the class.

3.3.3 Example

To illustrate our testing approach, we consider a small banking application implemented in Java. The application is kept quite simple in order to ease the understanding of the approach. Figures 3.3 and 3.4 depict respectively the class diagrams and statechart diagrams extracted from the UML specification. The complete Java program is provided in Appendix A. There are only two classes named *Account* and *Customer* in this class diagram (see Figure 3.3). The *Account* class provides methods for making various banking operations, such as deposit, withdraw and transfer. An account is characterized by the average income of its owners, its balance, and the credit line allocated by the credit manager. An account may be owned by one or several *Customer* objects. The UML statechart diagram in Figure 3.4 describes the behavior of an *Account* object in terms of

the messages it sends and receives. Transitions between states take the form *Event[Guard]/Actions*, all of which are optional. An account object is initialized in the *Credit* state. It may remain in that state or shift to the *Debit* state according to the balance of the account and the transactions performed.

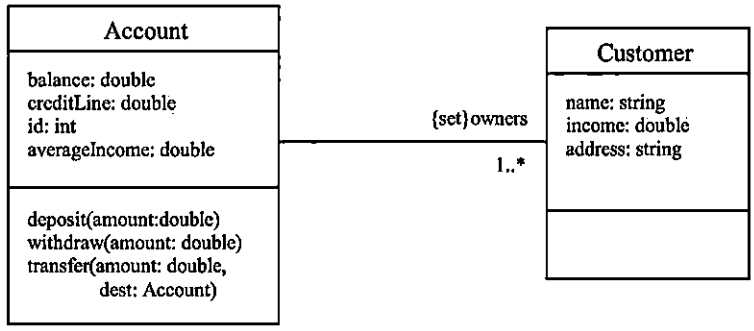
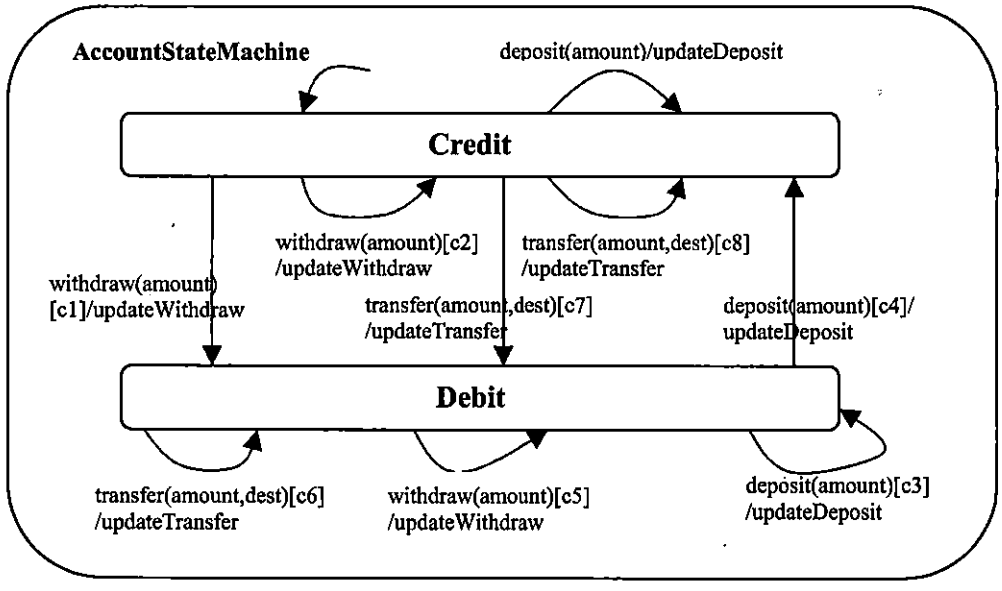


Figure 3.3 Class Diagram for the Banking Application



Guard conditions:
forall acc: Account
c1 = (amount-acc.balance<=acc.creditLine) AND (acc.balance<amount) AND (0<=amount)
c2 = (amount=<acc.balance) AND (0<=amount)
c3 = (acc.balance+amount<0) AND (0<=amount)
c4 = (0<=acc.balance+amount) AND (0<=amount)
c5 = (amount-acc.balance<=acc.creditLine) AND (0<=amount)
c6 = (amount-acc.balance<=acc.creditLine) AND (0<=amount)
c7 = (amount-acc.balance<=acc.creditLine) AND (acc.balance<amount) AND (0<=amount)
c8 = (amount=<acc.balance) AND (0<=amount)

Figure 3.4 Statechart Diagram for the Banking Application

We illustrate our approach in the sequel by considering the transitions involving method *withdraw*. There are three transitions that involve method *withdraw*. Each transition gives rise to a pre-post condition pair. Let us define, for instance, the pre-post condition pair corresponding to the withdraw transition that originates from state *Credit* and arrives at state *Debit*. Firstly, the predicates associated with the state, guard condition and action are defined as follows (where *balance'* is the value of *balance* after the execution of the method):

(States)

pred(Credit) = $0 \leq$ balance

pred(Debit) = balance < 0

(Guard conditions)

pred(c1) = (amount – balance $\leq$ creditLine) $\wedge$

(balance < amount) $\wedge$

($0 \leq$ amount)

(Actions)

pred(updateWithdraw) = ((balance' < 0) $\wedge$

(balance = 0.99*balance'-amount)) $\vee$

(($0 \leq$ balance') $\wedge$

(balance = balance'-amount))

Then, the pre/post-condition pair corresponding to method *withdraw* can be derived as follows:

pre $_{withdraw}$ = pred(Credit) $\wedge$ pred(c1)

post $_{withdraw}$ = pred(Debit) $\wedge$ pred(updateWithdraw)

By replacing the predicates involved by their respective expressions, we obtain the following expressions:

pre$_{withdraw}$ = ($0 \leq$ balance) $\wedge$

(amount – balance $\leq$ creditLine) $\wedge$

(balance < amount) $\wedge$

($0 \leq$ amount)

$post_{withdraw}$ = ((balance < 0) ∧
(balance' < 0) ∧
(balance = 0.99*balance'-amount)) ∨
((balance < 0) ∧
(0 ≤ balance') ∧
(balance = balance'-amount))

The next step towards test case generation is the decomposition of the preconditions into DNF. In this case, $pre_{withdraw}$ doesn't require any further decomposition because it is already in disjunctive form. However, we do need to break it into elementary conjuncts prior to performing domain analysis. The four conjuncts involved in the expression are given below:

conj1 = (0 ≤ balance)

conj2 = (amount – balance ≤ creditLine)

conj3 = (balance < amount)

conj4 = (0 ≤ amount)

We define test cases by analyzing the domain and by studying the boundary conditions for each conjunct. The test cases are identified and organized using the domain matrix depicted by Table 3.1. There are in total eight test cases based on the above boundary conditions. Only four of these test cases make the preconditions TRUE and are valid test cases.

Boundary			Test Case							
Variable	condition	type	1	2	3	4	5	6	7	8
balanced	balance>=0	on	0							
		off		-0.001						
	amount-balance<=creditLine	on			0					
		off				-0.001				
	balance<= amount	on								
		off								
	typical	in					0.001	2	50	50
amount	amount>=0	on					0			
		off						-0.001		
	amount-balance<=creditLine	on								
		off								
	balance<= amount	on								
		off								
	typical	in	100	100	100	100			100	100
creditLine	amount-balance<=creditLine	on							50	
		off								49.99
	typical	in	1000	1000	100	100	100	100		
Expected Results			TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE

Key: TRUE=IUT accepts this value, FALSE=IUT rejects this input

Table 3.1 Test Cases for Method Withdraw

3.4 Analyzing Object Predicates Using Decision Tree

As we mentioned earlier, domain testing is a systematic test case generation approach based on assertions. The key idea of it is to define boundary conditions or constraints for the input variables, which may correspond to message parameters and instance variables (attributes) in a class. However, the most challenging aspect of this technique is that it would be relatively easy and straightforward if all input variables were primitive data types, but what about variables of complex types, such as object variables?

It is essential to consider this situation because most object-oriented implementations define instance variables or method parameters with complex data types. In this case, we cannot directly generate test cases from the predicates of the defined states of an object. Instead, we can define a decision tree model, which mimics the attribute structure of the object variables involved to assist us.

In this section, we describe an approach for creating a decision tree model for a predicate with object variables in order to generate test cases using domain analysis. In the sequel, we call a predicate involving only primitive variables a simple predicate and a predicate involving at least one object variable an object predicate.

3.4.1 Decision Tree

A decision tree is a directed acyclic graph used to address problems with hierarchical structure. More formally, a decision tree T is a pair (D, E) where D is a set of nodes and E is a set of edges. D contains only one *root* node, zero or more *internal* nodes and zero or more *leaf* nodes (also called terminals). An edge is a connection between nodes, every node has exactly one incoming edge and zero or more outgoing edges except that root node has no incoming edges and leaf nodes have no outgoing edges. Figure 3.5 shows a typical decision tree representation.

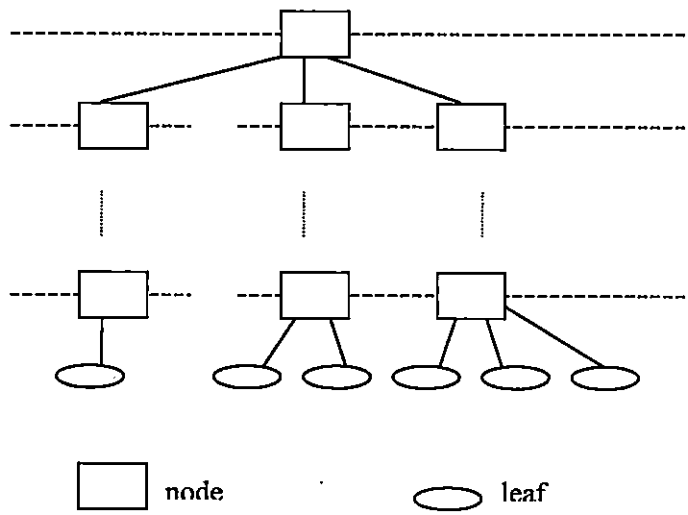


Figure 3.5 A General Form of Decision Tree

Decision trees have been successfully used for complex problem solving in a wide range of areas and applications. The rationale behind using a decision tree is that sometimes complex tasks of decision-making, which are difficult to solve directly, can be broken down to simpler ones. Then, based on some decision rules, the final complex

decisions can be made indirectly. Russell says that *a decision tree takes as input an object or situation described by a set of properties, and outputs a yes/no decision. Decision trees therefore represent boolean functions. Functions with a larger range of outputs can also be represented...* [11].

There are two frequently used approaches to decision tree construction: *top-down* and *bottom-up*. Whichever way is chosen, a well-designed decision tree having the following features is always the main concern:

- clear definition of the problems,
- simple and concise terminology and expressions,
- good split rules for internal nodes, and
- leaf nodes sufficiently pure and easily usable for decision-making.

Splitting rules are criteria used to distinguish leaf nodes from internal nodes. Stopping rules are criteria for stopping the decomposition process for internal nodes. Both of them determine the structure of a decision tree. Typically, the design of a decision tree using the top-down approach boils down to three tasks: (1) the selection of splitting rules, (2) the definition of the stopping rule, and (3) the labeling of the terminal nodes [12].

3.4.2 Construction of Object Decision Tree

Given an object predicate, we construct a corresponding decision tree, that we call an object decision tree, by navigating through the attribute structure of the object variables involved. The nodes of the tree correspond to the variables involved in the object predicate. More specifically, the combination of all the primitive variables involved in the predicate is represented as a single leaf node, and each object variable is mapped to a unique non-terminal node. Each non-terminal node is then refined downward following the local attribute structure of the corresponding object variable and by applying the previous rule: the combination of all of the primitive variables at a given level is mapped to a single leaf node, and object variables are mapped to non-terminal nodes.

The root node in the tree may correspond to an instance of the class under test. We may give an arbitrary name to that instance and start the construction from there. Each edge in the decision tree corresponds to predicates of the child node in the edge. Given two nodes a and b where a is the parent of b, we denote by $Pa \rightarrow b$ the predicate

associated with edge (*a, b*); $Pa \rightarrow b$ is a function of *b*. If *b* is a leaf node, $Pa \rightarrow b$ resolves to a predicate that can be analyzed using the classical domain analysis method.

The decision tree must be non-overlapping (e.g., the conjunction of any two paths must evaluate to false) and exhaustive (e.g., the disjunction of all paths must evaluate to true).

To better understand how an object decision tree is generated, the previous banking application example is still used, but modified by adding one extra class named *Money*, and reducing the number of attributes and methods to keep the tree simple (see Figure 3.6). *Money* encapsulates the value and currency of the amount involved in banking transactions. Predicates such as those associated to *Debit* and *Credit* become object predicates because they involve object variables such as *balance*. Therefore, object decision trees should be constructed based on the attribute structure of every object involved in these predicates. Figure 3.7 depicts the object decision tree for the banking application where *acc* is an object variable of type *Account*, *balance* and *income* are object variables of type *Money*, and *c* is an object variable of type *Customer*. Alternatively, in practice the rectangle with the broken line could be added in the tree for a node representing a set of objects, but the decision outcome (e.g., predicates) will depend on each individual object in the set, rather than the set itself. For example, in this case, there is a set of objects of class *Customer* named *owners* for which an object variable *c* is used to represent its individual elements and evaluate associated predicates.

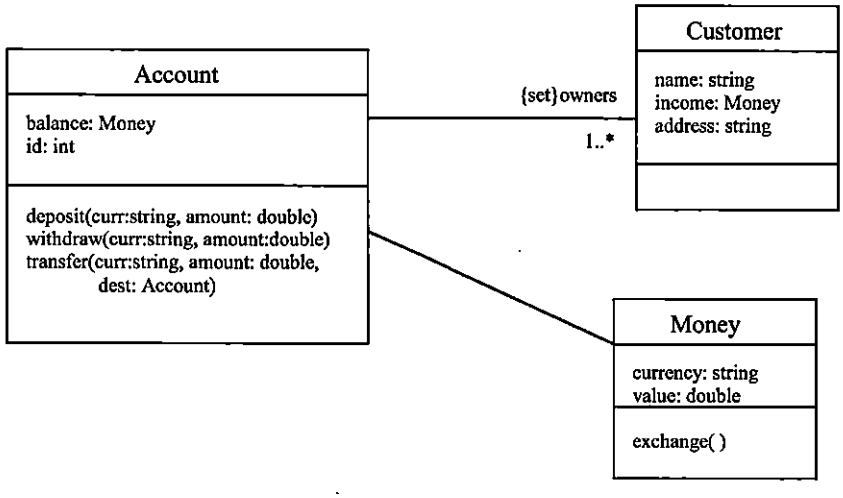


Figure 3.6 Modified Class Diagram for the Banking Application

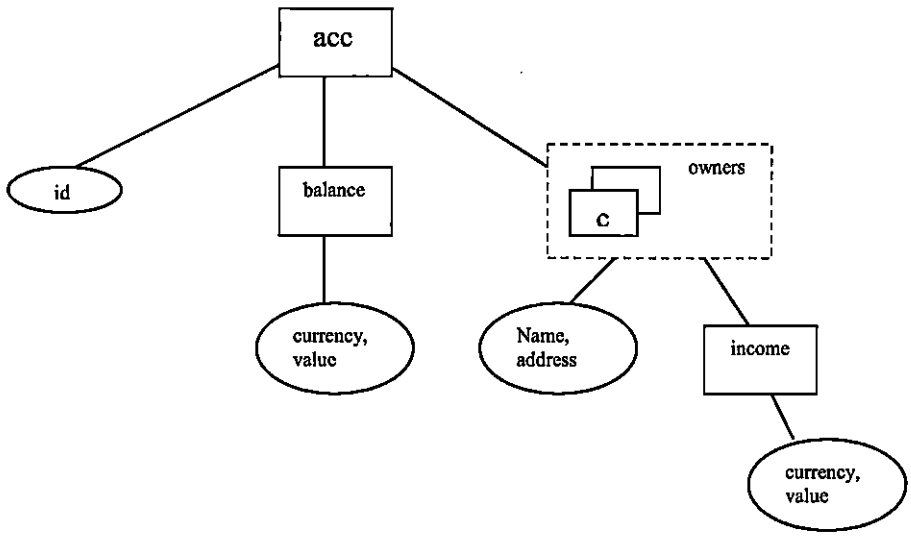


Figure 3.7 Object Decision Tree For the Banking Application

Test case generation starts by analyzing the simple predicates (related to the leaf nodes) using classical domain testing. Instances of the object variables corresponding to parent nodes are then constructed accordingly. By iterating bottom-up and reusing the previously built instances, classical domain analysis is applied level by level up to the root node.

3.4.3 Test Case Generation Algorithm

The test case generation algorithm is straightforward. It computes a collection of instances for the object variables involved in the class under test based on the given object predicates. Figure 3.8 shows the major steps of this algorithm.

(1) Before starting the process of test case generation, decompose each complex DNF expressions involved in the preconditions into a set of simple expressions
(2) Compute the test values based on each simple expression in this set locally
(3) Construct the corresponding instances for the object variables based on the object decision tree bottom-up until reaching the instance of the class under test.
(4) Repeat steps 1 to 3 for all DNF expressions in the precondition

Figure 3.8 Test Case Generation Algorithm

The test case generation algorithm is described by applying it to the withdraw method in the banking application example. Before starting to describe the algorithm, it is necessary to specify how the object predicates are defined, and how the related complex constraints are expressed and recorded. We express the constraints using a subset of Object-Constraint Language (OCL) [64]. For instance, the fact that financial transactions must take place in US dollars is expressed as follows:

a.m.currency = "US"

where *a* is an object variable of type *Account*, and *m* is an object variable of type *Money*. By extending the previous predicates of method withdraw with that new constraint, we get the following expressions:

(States)

$\forall$ acc: Account, m:Money

pred(Credit) = (acc.m.currency= "US") $\wedge$ (0 $\leq$ acc.m.value)

pred(Debit) = (acc.m.currency= "US") $\wedge$ (acc.m.value < 0)

(Guard conditions)

$\forall$ acc: Account, m: Money

pred(c1) = (acc.m.currency = "US") $\wedge$

(acc.m.value < amount) $\wedge$

$(0 \leq amount)$

(Actions)

$\forall$ acc: Account, m: Money

$pred(updateWithdraw) = ((acc.m.currency = \text{“US”}) \wedge$
$(acc.m.value' < 0) \wedge$
$(acc.m.value = 0.99*acc.m.value'-amount)) \vee$
$((acc.m.currency = \text{“US”}) \wedge$
$(0 \leq acc.m.value') \wedge$
$(acc.m.value = acc.m.value'- amount))$

(pre/post condition pair)

$pre_{withdraw} = ((acc.m.currency = \text{“US”}) \wedge$
$(0 \leq acc.m.value) \wedge$
$(acc.m.value < amount) \wedge$
$(0 \leq amount)$

$post_{withdraw} = ((acc.m.currency = \text{“US”}) \wedge$
$(acc.m.value < 0) \wedge$
$(acc.m.value' < 0) \wedge$
$(acc.m.value = 0.99*acc.m.value'- amount)) \vee$
$((acc.m.currency = \text{“US”}) \wedge$
$(acc.m.value < 0) \wedge$
$(0 \leq acc.m.value') \wedge$
$(acc.m.value = acc.m.value'- amount))$

The precondition in this case is already in normal form, so it is ready to be used for domain analysis.

During the first step of the algorithm, complex expressions are decomposed into a set of elementary expressions by eliminating the AND operator. In order to apply domain analysis, the obtained expressions are replaced by equivalent expressions in which referenced variable names are replaced by corresponding primitive variable names. That is essentially a syntactic operation whose sole goal is to ease the manipulation of

expressions during domain analysis. For example, the previous precondition can be decomposed into the following simple conditions: c1 = {currency = "US"}, c2={value ≥ 0}, c3={amount ≥ 0} and c4={value<amount} for the *Money* object. Test values that satisfy these simple conditions are easily identified for the *Money* object. Suppose that the input values for method parameters *currency* and *amount* are respectively "US" and 200, respectively, the instances *m1* and *m2* shown in Table 3.2 (a) can be used for possible test case generation because the values for attributes *currency* and *value* satisfy the above four simple conditions.

The next step of the algorithm is to construct the object variables accordingly based on the object decision tree bottom-up and level by level until getting the instances of the class under test, which will be treated as the object values used for object domain analysis. Table 3.2 (b) shows the construction of instances for object Account. The final test cases are also represented in the domain matrix shown in Table 3.3. If there is more than one DNF expression in the precondition, repeat the same process from steps one to three.

Table3.2 (a) Construction of Instances For Class Money (m)

Instances	Instance Variables	
	currency	value
m1	US	100
m2	US	150
m3	US	300
m4	US	-100
m5	CA	100

Table3.2 (b) Construction of Instances For Class Account (acc)

Instances	Instance Variables	
	balance	id
acc1	m1	1234
acc2	m2	1234

Table 3.2 Construction of All The Instances Involved

Boundary			Test Case				
Variables	condition	type	1	2	3	4	5
acc	acc.m.currency==curr=="US"&& acc.m,value>=0&& acc.m.value<=amount	on					
		off					
	typical	in	acc1	acc2			
curr	curr="US"	on	US	US			
		off					
	typical	in					
amount	amount>=o	on					
		off					
	typical	in	200	200			
Expected Results			TRUE	TRUE			

Table 3.3 Domain Matrix for Object Variables

3.5 Summary

The selection of test data is a crucial step of program testing. Domain analysis is a straightforward and effective way for selecting test values. We have presented in this chapter a test strategy at the class level based on the set of transitions involved in the statechart describing the dynamic behavior of the class. The proposed strategy, named the transition test strategy, combined an extended form of the conventional domain analysis technique in order to cope with predicates involving object data types as well as primitive data types.

Chapter 4

Testing Based On UML Class Diagrams

This chapter presents an overview of UML class diagrams. Two types of the most important relationships between classes in object-oriented paradigm, *association* and *inheritance*, are described. Issues on testing class *association* and *inheritance* are also discussed.

4.1 Overview of UML Class Diagrams

UML class diagrams give a static overview of a system by showing its classes and the relationships among them. The main constituents in a class diagram are classes and relations. Figure 4.1 shows an example of a UML class diagram for a small banking application.

A class is an abstraction of a concept or a physical thing; it may have attributes and operations associated with it. An attribute is a named property of a class that describes a range of values that instances of the class may exhibit. An operation is the implementation of a service that can be requested from any object of the class, and that may affect its behavior. Graphically, a class is rendered as a rectangle, which can be further divided into three parts. The top part lists the class's name; the middle part lists attributes, if any; the bottom part lists operations, if any. In Figure 4.1, for example, the class named *Account* consists of attributes – *accNo* and *balance*, and operations – *deposit*, *withdraw* and *transfer*. However, the class *SavingAccount* has only one attribute named *interestRate* and no operations.

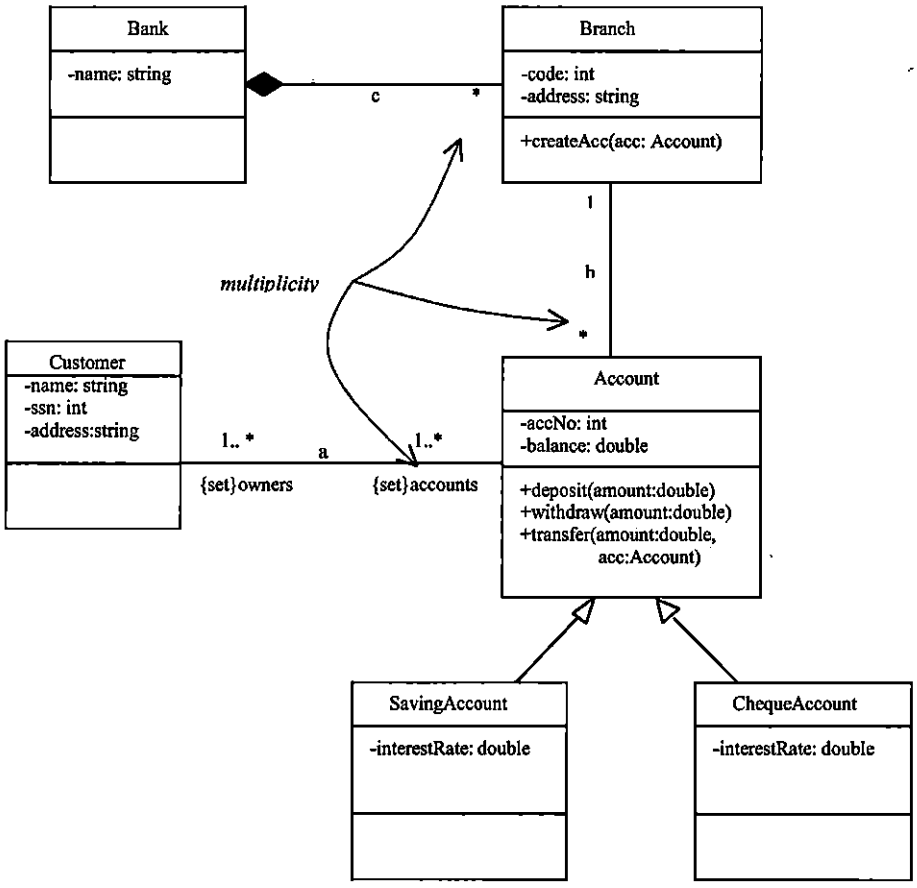


Figure 4.1 An Example of UML Class Diagram

A relationship between two classes is drawn as a line. Our class diagram Figure 4.1 shows three kinds of relationships, namely association, aggregation and generalization (e.g. inheritance), which are important in UML. We give a brief description of these relations as follows:

- **Association**: a structural relationship specifying that instances of two classes are related. An association represented by a joining line shows peer-to-peer visibility for instances of the two classes.
- **Aggregation**: a special kind of association in which one class belongs to another class or is part of another class. An aggregation is described using a line with diamond end pointing to the containing class.
- **Generalization**: characterizes a relationship between a general concept or thing and a more specific one. A generalization is described by a directed line with an open arrowhead pointing from the specific class to the general class.

Multiplicity parameters may appear at the end of the association line: numbers, letters and symbols indicate the number of instances of each class that will be involved in the relationship. In our example, every customer has at least one account and every account is owned by at least one customer. Table 4.1 depicts some examples of multiplicities.

Multiplicities	Meaning
1	one and only one instance of the class is involved in the relation
1..n	the number of instances of the class varies from 1 to n
* or **0..***	the number of instances of the class is arbitrary
none	no restriction on the number of the instances of a class

Table 4.1 Symbols of Multiplicities in UML

4.2 Class Association and Inheritance

Classes seldom stand alone in object-oriented systems. Classes involved in an object-oriented program collaborate and work together to form a system that fulfills some required tasks. In general, classes can be used to compose new classes in two ways, or in other words, using two kinds of primary connections: *association* and *inheritance*. Using *association*, one class may contain instances of another class. Most object-oriented language implements this through attributes of the objects. *Inheritance* allows the representation of one class to be defined in terms of the representation of one or more existing classes. The new class (child class) inherits its parent's state variables (attributes) and methods. The mechanisms described above are among the most important features of object-oriented programs.

4.3 Issues on Testing Classes and Relationships

Since association and inheritance play an important role for OO design and programming, it is necessary to ensure that they are correctly modeled, and are consistent with the requirements. There are three main issues underlying testing class association and inheritance; we review them briefly in the sequel.

Firstly, a UML class diagram provides structural information of classes and relationships such as the attributes of a class, what classes are involved in a relationship, or the multiplicity of an association. Hence, from the simplest testing point of view, syntax testing should be performed to verify that the class diagram contains correct and proper information.

The second issue concerns the scope of integration testing. An object-oriented program is usually made of a collection of classes to solve problems. Objects interact through their methods or associations with other objects. However, such relationships could be within a system, a subsystem, or a cluster of classes. The interactions between objects could also take place within one class, a class hierarchy, or a client and server environment. Thus, the scope for integration testing of class relationships must be taken into account. A UML class diagram models a cluster of related classes within a package corresponding to a software component or a subsystem; hence testing relationships of classes within a cluster and a subsystem seems feasible.

Thirdly, dependency is another important issue for testing the relationships of classes as well as testing classes since classes often depend on each other in many ways. Among the most typical examples of class dependency are super and sub classes in inheritance, one class may be used to define the instance variables of another class, objects used as method parameters and so on. Therefore, dependency analysis becomes a key step for integration testing of classes and their relationships. As a matter of fact, dependency analysis can also result in the decision of test order that impacts the construction of test drivers or stubs, and the preparation of test cases. Most approaches to cluster testing adopt dependency analysis to support bottom-up testing [03]. UML class diagrams, as a graph-based model, can provide some basic dependency information for various classes, but the testing order for these classes is not addressed.

In short, the test approaches may vary due to their intrinsic differences although integration testing can be applicable for testing class association and inheritance. In this context, we present in detail in Chapter 5 and Chapter 6 some strategies for testing association and inheritance, respectively.

4.4 Summary

Relations in UML class diagrams are important as well as classes. UML class diagrams provide a static view for not only a set of classes but also the relationships between these classes. Few classes are separated from other classes in an object-oriented system. Integration testing of classes requires a thorough examination of the relationships involved.

Chapter 5

Testing Class Association

This chapter presents a relational test strategy for class associations. The proposed test strategy is an integration testing strategy. The test model used consists of the collection of association and aggregation relations involved in a class diagram.

5.1 Properties of Association in UML Class Diagrams

An association in UML is a structural relationship specifying that one object is logically connected to another object. Given an association connecting two classes, one can navigate from an object of one class to an object of the other, and vice versa. Consider for example a *Person* object who is employed in a *Department* object of a company. The association "is employed" could be implemented through attributes of *Person* and *Department* objects. That is, each *Person* object has an attribute *Department* pointing to appropriate department in the company. Each *Department* on the other hand has an attribute *Person* pointing to a set of persons who are working in that department. In UML, aggregations are associations with special semantics representing "whole-part" or "is part of" relations between objects. Hence, testing associations is identical to testing class aggregation.

On the other hand, from the perspective of testing, it is often considered that identifying the existence of the relations of two objects doesn't seem to be so significant. Instead, the primary concern is to ensure the definition of properties that relations must strictly hold. The following are two important properties for an association worth testing:

- *Referential integrity* means that an object referenced from other objects should exist. For instance, for two related objects *Person* and *Department*, we may require that a *Person* has to work at or belong to a *Department* of the company. In other words, if a *Person* is located or found, its corresponding *Department* must exist or be known.
- *Multiplicity* denotes associations having cardinality such as one-to-one, one-to-many or many-to-many. For example, in a relation *"is employed at"* between object *Person* and *Department*, a *Person* may belong to only one *Department*, but the *Department* may possess an unrestricted number of *Person* objects as members. In this case, the property of multiplicity for the relation *"is employed at"* is one-to-many.

5.2 A Relational Test Strategy

The graph structure of a UML class diagram lends itself to relational testing. We present a relational test strategy for testing associations. The relational test strategy developed herein is based on concepts and techniques from a formal relational model and is useful in testing the above properties of associations between classes. It focuses on the test of special methods or sets of attributes associated with a class such as constructors, vectors, or hashtables.

A UML class diagram D can be formally defined as a pair consisting of a set of classes C and a set of relations R, and denoted $D = \{C, R\}$. The set of relations R may contain several subsets including a set of associations RA or a set of generalizations RG etc., $RA, RG \subset R$. Our relational test strategy focuses mainly on testing the collection of associations in RA including all *UML associations* and *aggregations*. A class can be naturally associated with a type. A relation can be associated with a predicate taking the name of the relation and having as parameters the elements involved in the relation. Given an association L between two classes A and B, of respective multiplicities m and n (see Figure 5.1), we define the corresponding predicate as follows:

$$L : T_A \times M \times T_B \times M \rightarrow \text{bool}$$

where T_A and T_B are the types associated respectively with classes A and B, and M is the set of multiplicities. Multiplicities correspond to constraints on the number of instances of classes involved in associations.

The relational test model that we use to test class associations consists of the set of association relationships (e.g. *RA*) associated with the class diagram (e.g. *D*). Each association is a 5-tuple consisting of the name of the association, the classes involved, and the corresponding multiplicities. The strategy is to first test each single association in the set of associations. Then, in order to test a collection of associations relating several classes, graph matrices corresponding to class diagrams are defined and reduced using the node-reduction algorithm defined by Beizer [01].

Given an association L involving classes A and B with respective multiplicities m and n, as depicted in Figure 5.1, corresponding predicate can be modeled as follows:

pred_L : bool = [L, A, m, B, n]

suppose that a and b are respectively instances of classes A and B. Applying predicate L to the tuple <L, a, B.size(), b, A.size()> should evaluate to true if A.size( ) and B.size( ) are within the ranges defined by the multiplicities (e.g. m, n). Otherwise, that indicates the existence of a bug in the implementation of the association. Either a or b being null or nonexistent will result in the failure of creating the corresponding data set, or either B.size( ) or A.size( ) being out of scope of m or n can falsify the predicate. This would imply that association properties such as *referential integrity* or *multiplicity* are violated.

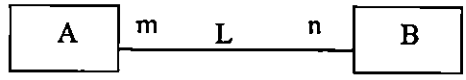


Figure 5.1 Associations

5.3 Test Case Generation for An Association

To illustrate our test case generation approach for association, we extend the banking application example. One of the basic requirements is that each account must have at least one account owner (i.e. customer), but a customer does not have to possess an account. For instance, the bank may still keep records for those former customers who have already closed their accounts. The UML design class diagram used is shown in Figure 4.1 of Chapter 4. The partial class diagram depicting the relationship between the class *Customer* and *Account* is shown in Figure 5.2.

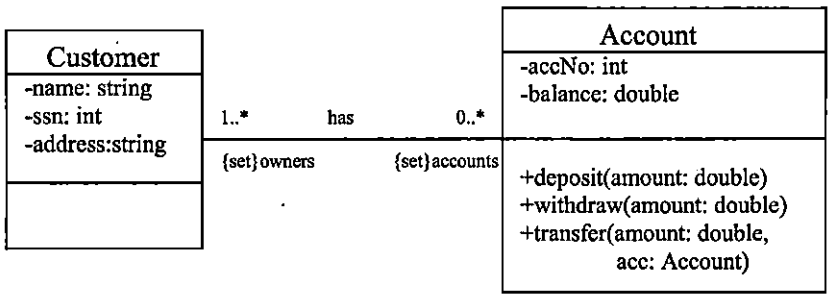


Figure 5.2 Subset of the Class Diagram for the Banking Application

First, the association *has* can be defined as a predicate as follows:

$\forall$ c: Customer, acc: Account

pred_A : bool = [has, c, X, acc, Y]

where X is the set *accounts* containing a set of instances of class *Account*, and Y is the set *owners* containing a set of instances of class *Customer*. Test cases generated are based on this predicate.

The first property of the association we want to test is referential integrity. Based on the requirement mentioned above, the referential integrity of the association *has* requires that if an instance of class *Account* exists, a corresponding instance of class *Customer* must be available. In order to test that, the instances of class *Customer* are treated as input data, and whether the corresponding instances of class *Account* are successfully created or not will be observed by assigning non-null or null instances for class *Customer*. Test cases for testing referential integrity of the association *has* are shown in Table 5.1 in which Table 5.1(a) shows the set of instances for class *Customer*, and Table 5.1(b) is the domain matrix representing generated test cases.

Construction of Instances for Object Customer			
Instances	**Instance Variables**		
	name	**ssn**	**address**
c1	John	100001	88 Queen St.
c2	Null		
c3	Smith	100002	168 King St.
c4	Null		

(a)

Boundary			Test Case			
Variables	condition	type	1	2	3	4
accNo		on				
		off				
	typical	in	98-12345	98-12345	76-27456	76-27456
balance		on				
		off				
	typical	in	1000	1000	2000	2000
Customer	c != null	on	c1		c3	
		off		c2		c4
	typical	in				
Expected Results			TRUE	FALSE	TRUE	FALSE

(b)

Table 5.1 Test Cases for Referential Integrity

Another important constraint we want to test is the multiplicity property of this association, which implies how many instances of the class can be connected with the instance of another class. For example, the multiplicity 1..2 for class *Customer* corresponding to the set Y in this association *has* means that every account must have at least one or up to two account holders (*Customers*). Likewise, the multiplicity 0..* for class *Account* corresponding to the set X in the association *has* means each customer can have zero or more accounts, in other words, unrestricted number of accounts. Test cases are designed to ensure that the size of the set Y is within the scope of the multiplicity constraints defined in the class diagram. Table 5.2 shows a number of instances of class customer constructed. Test cases for the multiplicity of the set Y in the association *has* are shown in Table 5.3. In this case, there is no restriction for the set X.

Construction of Instances for Object Customer			
Instances	Instance Variables		
	name	ssn	address
c1	John	10001	1 Queen St.
c2	Mary	10002	2 Queen St.
c3	Smith	10003	3 Queen St.
c4	Tom	10004	4 Queen St.
c5	Carter	10005	5 Queen St.

Table 5.2 Construction of the Instances for Class Customer

			Test Cases				
Variables	Condition	Type	1	2	3	4	5
Customers		On					
		Off					
	Typical	In	{c1}	{c2}	{c1, c2}	{c1,c2,c3}	{c1, c2, c3, c4,c5}
Number of added customers to the account	1=< Y.size()		1	1	2	3	5
Expected result			Accept	Accept	Accept	Accept	Accept

Table 5.3 Test Cases for the Multiplicity of the Association

5.4 Testing Multiple Associations

Associations can be "joined". For instance, a joined association can be expressed as *has• belong• own* or *{Customer, Account, Branch, Bank}* where *has* is the association between object *Customer* and *Account*, *belong* is the association between object *Account* and *Branch*, and *own* is the association between object *Branch* and *Bank* as shown in Figure 5.3. Referential integrity ensures the joined association will work. For instance, if a *Customer* is successfully added to a *Bank*, not only does the *Bank* object have to exist, but also the corresponding *Account* and *Branch* objects must be available as well. Hence, the main concern of testing multiple associations consists of testing referential integrity, which is the property of a joined association.

Most testing activities based on graph specifications are likely to deal with path processing. The relational test strategy developed for testing multiple associations herein is no exception. To test referential integrity for a joined association, tracing the path expressions involving several associations and classes in a UML class diagram is crucial. However, as the number of nodes and edges becomes larger, the path expression will become complex. Such tasks as tracing paths are sometimes tedious and error prone; matrix operation can make this simple. A relation matrix, which represents all the classes involving path tracing and the node reduction algorithm can be used to get the expected path expression of any two nodes.

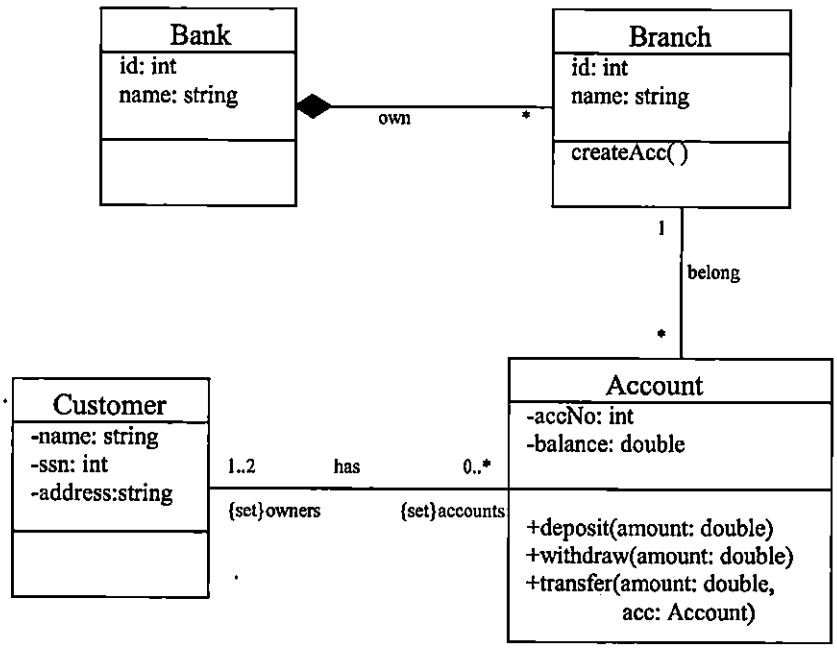
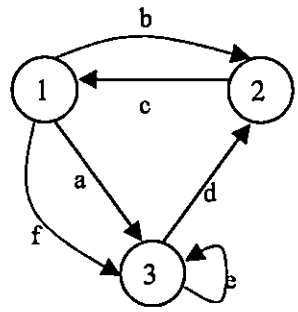


Figure 5.3 Modified Class Diagram for the Banking Application

5.4.1 Graph Matrix Representation of A Class Diagram

Since the test model we use to test class association consists of a set of associations, and sometimes class associations do involve several associations and classes, we need to determine a set of independent paths related to multiple associations and then prepare test cases that will force execution of each path in the set. In order to do so, graph matrices can make the process of determination of a set of paths simple, precise and automatic.

A graph matrix is a square array with one row and one column for each node in the graph. Each row-column combination corresponds to a relation between the node in the row and the node in the column. The relation, for example, could be denoted simply using the link name [01]. Figure 5.4 shows an example of graph and its corresponding graph matrix. For instance, the connection from node 1 to node 2 in Figure 5.4 (a) is labeled b, and in the corresponding matrix Figure 5.4 (b), it is located in the cell corresponding to the first row and second column. Similarly, for a UML class diagram, classes are modeled as nodes, and the relation between classes corresponds to the edges of the graph.



(a)

	1	2	3
1		b	a+f
2	c		
3		d	e

(b)

Figure 5.4 A Graph and Its Matrix Representation

5.4.2 Node Reduction Algorithm

The node reduction algorithm is an efficient way to get the path expressions from one node to another node, typically from the entry node to the exit node through processing a matrix. Compared with other methods, it is the most methodical and may be the quickest way to get the resultant of path expressions between two specified nodes [01]. The main steps of this algorithm are as follows:

(1) Before the initiation of the node reduction procedure, the entry node and exit node are determined. They should be located in the first and second position in the row and column of the matrix. If not so located, the nodes can be relocated by interchanging the corresponding rows and columns in the matrix without changing the underlying graph.

(2) Select one node for removal every time based on the rules. Usually start by removing the last node.

(3) Continue the node-reduction process until the two nodes of interest remain.

The node reduction algorithm procedure is described in the following paragraphs.

The first step and the most crucial and complex step of the algorithm eliminates a node and replaces it with a set of equivalent links. The previous graph example can be used to better describe this algorithm. The whole process for one node reduction is depicted in the changes to the matrix representation in Figure 5.5. Suppose we want to get the path expression from node 1 to node 2 in the matrix shown in Figure 5.5 (a). The algorithm will start reducing one node, for instance node 3, at a time. If any loop terms had occurred at this point (e.g. node 3), we should eliminate the loop term first by premultiplying every term in that row by the loop term starred (see Figure 5.5(b)).

If there is no loop term at this point, the node is ready to be removed by combining the elements in the column of this node (or last column in a matrix) with elements in the row of this node (last row in a matrix), and then putting the result into the entry at the corresponding intersection. In this case, the term $a+f$ in the position (1,3) will combine with the term de^* in the position (3,2) to yield the (1, 2) term $b+ade^* +fde^*$ shown in Figure 5.5 (c), which is the value after node reduction. If the reduction of more than one node is needed, continue the process until the final two nodes are left.

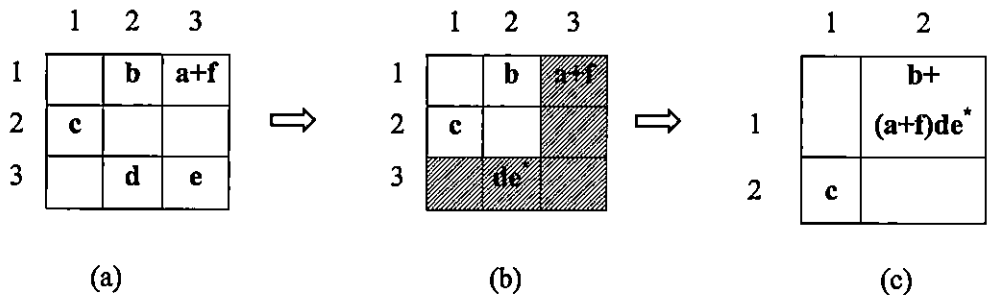


Figure 5.5 A Node Reduction Process

5.4.3 Example

To illustrate how to generate test cases for multiple associations, we still consider the above simple banking application example shown in Figure 5.3. Suppose a Person can become the customer of a bank by opening an account at any branch of the bank, and a joined association (i.e. multiple associations) between objects *Customer* and *Bank* is to be tested. We assume each single association in the diagram has already been tested without finding errors before testing the joined association. Test case generation steps are described in the following paragraph.

First, the matrix representation of the class diagram is generated in order to obtain the path expression from object nodes *Customer* to *Bank*. In this case, objects *Customer* and *Bank* are assigned respectively to column/row 1 and 2 in the matrix; for the rest of the object nodes, the number can be arbitrarily chosen. The matrix is shown in Table 5.4.

	Customer	Bank	Account	Branch
Customer			has	
Bank				
Account				belong
Branch		own		

Table 5.4 The Matrix Representation of the Class Diagram

Secondly, the node reduction algorithm is applied to get all possible path expressions between objects *Customer* and *Bank*. Figure 5.6 shows this process in more detail. Since there is no loop in the graph, nodes are eliminated one after the other until the two nodes *Customer* and *Bank* are left.

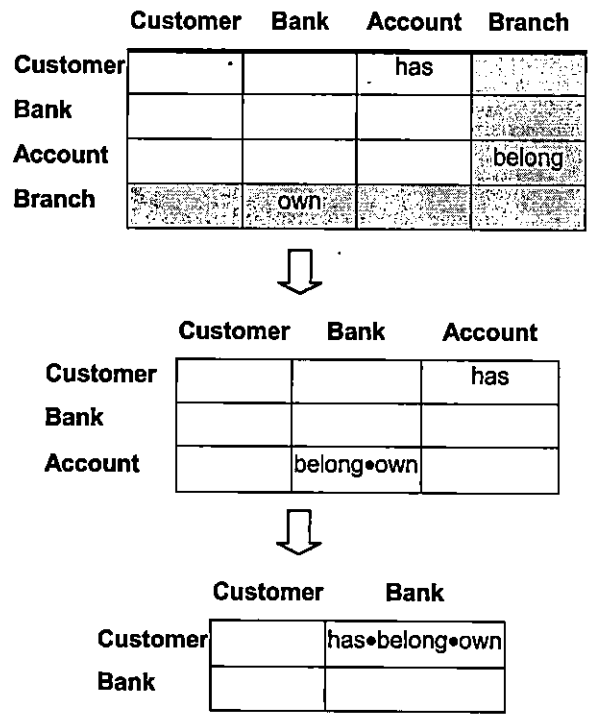


Figure 5.6 Node Reduction Process

Thirdly, test cases are generated based on the final path expression, which describes the joined association. For example, the path expression between Customer and Bank is *has• belong• own*. In order to test referential integrity for the joined association,

test cases involving null or non-null value of the objects *Bank*, *Account* and *Branch* are generated, and used to evaluate whether or not object *Customer* is successfully created. Tables 5.5 (a, b, c) show the construction of corresponding instances for *Bank*, *Branch* and *Account*. The domain matrix of test cases generated for the referential integrity of the joined association is shown in Table 5.6.

Tab. 5.5 (a) Construction of Instances for Object Bank

Instances	Instance Variables	
	id	name
bk1	001	CIBC
bk2	null	
bk3	001	CIBC
bk4	001	CIBC

Tab. 5.5 (b) Construction of Instances for Object Branch

Instances	Instance Variables	
	id	name
brch1	23	Bay&King
brch2	45	Johnson&Quadra
brch3	null	
brch4	56	Tillicum Center

Tab. 5.5 (c) Construction of Instances for Object Account

Instances	Instance Variables	
	accNo	balance
acc1	34547	2000.00
acc2	29605	456.92
acc3	87132	2300.00
acc4	null	

Table 5.5 Construction of Instances of Bank, Branch and Account

Boundary			Test Case			
Variables	condition	type	1	2	3	4
Bank	bk != null	on	bk1			
		off		bk2		
	typical	in			bk3	bk4
Branch	brch != null	on	brch1			
		off			brch3	
	typical	in		brch2		brch4
Account	acc != null	on	acc1			
		off				acc4
	typical	in		acc2	acc3	
Expected Results			TRUE	FALSE	FALSE	FALSE

Table 5.6 Test Cases for the Referential Integrity of Multiple Associations

5.5 Summary

In this chapter, a new technique for testing the associations between classes in a UML class diagram is presented. The goal of testing an association is to verify its properties, more specifically, multiplicity and referential integrity. The approach is based on basic concepts and techniques from the formal relational model. Approaches for testing both single and multiple associations are also discussed. When testing multiple associations, the graph matrix representation of a UML class diagram is first modeled. Then, nodes are removed by using a node-reduction algorithm in order to be able to get the path expression between any two nodes in a convenient way. The approach is believed to enhance the capture of errors in class association.

Chapter 6

Testing Class Inheritance

This chapter gives a brief overview of the features of class inheritance in object-oriented languages. Then specific kinds of errors related to class inheritance hierarchies are discussed and corresponding testing approaches are presented. The test model used is for integration testing and consists of only inherited classes in a UML class diagram, among which are a base class and its subclasses.

6.1 Inheritance in Object-Oriented Development

Inheritance is perhaps the most significant innovation in object-oriented programming. One of its most important features is that it provides a mechanism for reuse. A class is both a module and a type: as a module, it encapsulates a set of data members and member functions offered to other classes; as a type, it describes a number of run-time objects – its instances. Thus, inheritance implies two different things: *subclass* and *subtype*. Further, there are two kinds of reuse: module reuse and type reuse. For instance, if a class A is inherited by a class B, as module reuse, that means all the attributes and methods in class A can be inherited and directly referenced within class B. On the other hand, as type reuse, the class B is also a subtype of the class A, and any instance of class B can be freely used whenever an instance of class A is expected, this is sometimes referred to as "substitutability" [01, 02]. From a testing perspective, errors related to subclass inheritance can mostly be checked by a language compiler or processor. Most errors, such as inheritance and polymorphism faults, are caused by misuse of subtype, hence, we will restrict our attention to the investigation of anomalies related to subtype inheritance.

Polymorphism is the capability to treat an object as belonging to more than one type and is often bound up with inheritance. Two forms of genuine polymorphism are the most familiar in object-oriented languages: (1) *Inclusion polymorphism* can refer to subtyping; it specifies that an instance of a subclass can be used whenever an instance of a super class is expected. (2) *Parametric polymorphism* refers to types (or objects) that are parameters to functions and types (or objects). For example, array is a parametric type which allows instances of array[int], array[boolean] and so on. When polymorphism is combined with method overriding, the same call can result in the execution of different methods. This is called *dynamic binding*, which implies the method that is actually executed cannot be known statically and must be determined based on the type of the object during runtime when the call is executed.

Three other issues for class inheritance of concern are:

(1) There may exist multiple inheritance among classes, that is, a subclass may have several parent classes with faults that may be inherited.

(2) Due to the anomalies in a super class that can be also inherited by all its descendants when we use the powerful mechanism of inheritance, testing subclasses should be based on the strong assumption that the super class is anomaly free.

(3) Accessibility to the super class facilities depends strongly on programing language: in Java, for example, only public or protected attributes in the super class can be accessed from its subclasses although all the attributes in the super class are inherited. The following fragment of Java code shows that the private attribute *id* in class *Account* cannot be accessed from its subclass *SavingsAccount*; the error will be uncovered (see Figure 6.1) when compiling the subclass.

```
class Account
{
        public String name;
        private int id;
        protected double amount;

        Account()
        { }

}
class SavingsAccount extends Account
{

        SavingsAccount()
        {
```

```
            name="John";
            id=1001;
            amount=2456.00;
        }

}
```

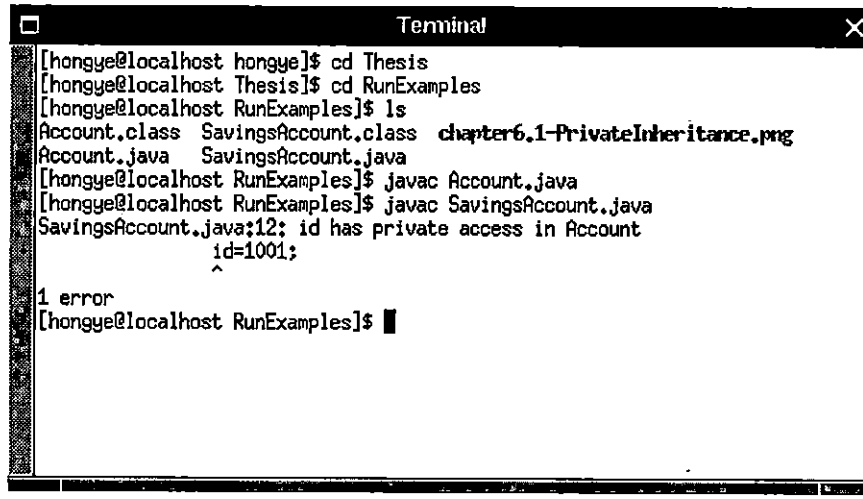


Figure 6.1 A Screen Snapshot for Compiling the Class SavingsAccount

Hence, unless otherwise noted, this thesis makes three assumptions regarding class inheritance:

- the strategies for the testing of class inheritance are only for the mechanism of single inheritance, which can be used to model most features of modern object-oriented systems.
- the base class has been thoroughly tested at least at the module level and found to have no errors.
- all the state variables (attributes) used in a super class can also be accessed from its subclasses.

6.2 Liskov Substitution Principle (LSP)

LSP is a principle that regulates inheritance and polymorphism in object oriented design. The purpose of LSP is to make object-oriented code more reusable and maintainable. The formal statement of this principle is as follows [04]:

> *If for each object o1 of type **S** there is an object o2 of type **T** such that for all programs **P** defined in terms of **T**, the behavior of **P** is unchanged when o1 is*

substituted for o2 then S is a subtype of T.

More simply, this means that modules that use references to base types (e.g. super class) must be able to use references to derived types (e.g. sub class) without knowing the difference. Under this substitution principle, the methods in a subclass must obey the following rules [02]:

- the preconditions for the overriding methods in a sub class must be the same or weaker (less restrictive) than those in its super class
- the post-conditions for the overriding methods in a sub class must be the same or stronger (more restrictive) than those in its super class
- the class invariant must be the same or stronger, that is, add constraints

Suppose a design involving inheritance and polymorphism as shown in Figure 6.2. Class C inherits from class B, and each class has a different implementation for the method m(), which can be denoted and distinguished as B::m() or C::m(). An instance of class T is considered sending messages to an instance of class B. When writing programs, polymorphism allows the instance of C to be substituted for the instance of B. But, the substitution should follow the rules regarding pre and post conditions for each inherited operation as mentioned above. For instance, the method call() in T must satisfy the preconditions of the m() operation of B in order for the method B::m() to execute. If an instance of C is to be replaced, the preconditions for C::m() must not add any new conditions to those of B::m(). In other words, class T knows only preconditions of the m() method of the base class.

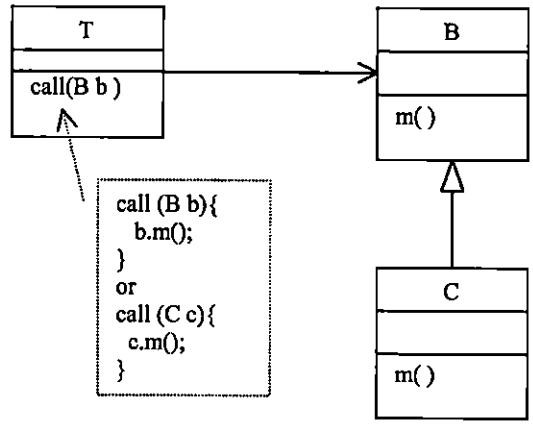


Figure 6.2 A Design Involving Inheritance and Polymorphism

6.3 Some Problems With Class Inheritance and Polymorphism

The power that inheritance and polymorphism bring to the expressiveness of object oriented programming, also creates the potential for a number of new faults. For software testing, identifying potential sources of errors is critical for successful development of suitable test strategies. Basically, as indicated by Binder [03], there is a variety of errors related to class inheritance; most of these errors come from some degree of misuse of polymorphism and overridden methods, dynamic binding and so on. We review more closely, some of the faults related to polymorphism and overridden methods in the class inheritance hierarchy.

6.3.1 Inconsistent Overriding

An incorrect overriding method in a descendent class may break the functionality of methods in its super class, especially the method for making copy operations. To get a better understanding of this error, let us consider an example related to abstract data types. The code fragment given in Figure 6.3 implements inheritance hierarchy between the class *List* and *Set*. The class *List* encapsulates a data structure that allows duplicating elements and the copying of a list to another list. The class *Set* has only non-duplicated elements. Since one may think of a *Set* as a list that contains no duplicates, the *Set* class could inherit from the *List* class in order to reuse existing code. The only change would be to override the *addElement* method to provide the non-duplicated element requirement. This seems to be fine, but it may be possible to break the functionality of the method *copyTo* because the *copyTo* method may still work error-free as long as a List is copied to another List, a Set is copied to another Set, or even a Set is copied to a List. However, errors can occur when copying a List to a set. For instance, if a given List *la* in the class *TestCopy* (see Figure 6.3) contains elements {a, b, c, a, b} and is copied to another List *lb*, there will be no problem; but, if the List *la* is copied to a new Set *s1*, the set will just contain {a, b, c} because it doesn't allow duplicated elements. Figure 6.4 shows the printed results of the list *lb* and the set *s1* after executing the class *TestCopy* in which the list *lb* is completely the same as the given list *la*, but the set *s1* is not. Clearly, this is not exactly what the

method *copyTo* should have done: The LSP (Liskov Substitution Principle) has been violated.

```
public class List
{
   Vector   list;
    List()
    {
       list=new Vector();
    }

   //Adds an element to the end of the list
   public void addElement(Object element)
   {
      list.add(element);
   }

   //Gets the i-th element in this list
   public Object getElementAt(int i)
   {
      return list.get(i);
   }

   // Returns the number of elements in the list
   public int getLength()
   {
       list.size();
   }

   //Returns true if element is contained in this list
   public boolean hasElement(Object element)
   {
      return list.contains(element);
   }

   //Removes all the elements for this list
   public void clear()
   {
      list.clear();
   }

   // Copies the content of one list into dest list
   public void copyTo(List dest)
   {
     dest.clear();
     int n = getLength();
     for(int i = 0; i < n; i++){
        dest.addElement(list.getElementAt(i));


     }

   }

}//end of class List
```

```
public class Set extends List
{

   // Adds element to this set eliminating the duplicated one
   public void addElement(Object element)
   {
     if(!hasElement(element))
     {
        super.addElement(element);
     }
   }

}//end of class Set


public class TestCopy
{
  public static void main(String[] args)
  {
    List la = new List();
    la.addElement("a");
    la.addElement("b");
    la.addElement("c");
    la.addElement("a");
    la.addElement("b");

    System.out.println("The list la is :"+la);

    List lb= new List()
    la.copyTo(lb);
    System.out.println("The list lb is :"+lb);
    for(int i=0;i<lb.getLength();i++)
    {
      System.out.println("One of elements in the list lb is
                          :"+lb.getElementAt(i));
    }

     Set s1 = new Set()
     la.copyTo(s1);
     System.out.println("The set s1 is :"+s1);
     for(int i=0;i<s1.getLength();i++)
     {
       System.out.println("One of elements in the set s1 is
                           :"+s1.getElementAt(i));
     }

   }//end of main method

}//end of class TestCopy
```

Figure 6.3 Code Fragment Showing the Inheritance of Class List and Set

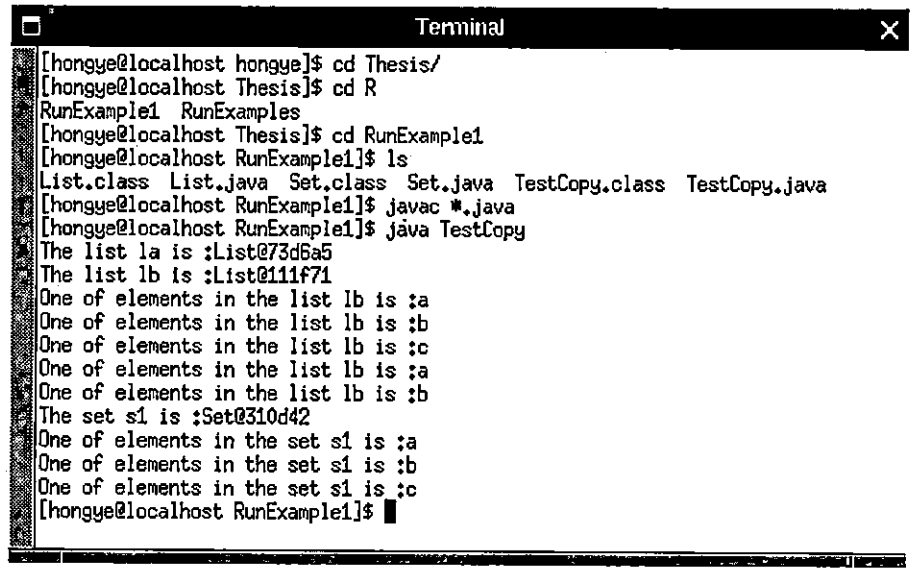


Figure 6.4 A Snapshot for the Results of Running the Class TestCopy

The LSP violation is established by carefully analyzing the pre and post conditions of the overriding and overridden methods as follows:

pre and post conditions for the overridden method *addElement* in the class *List*,

pre: element != null

post: this.getLength() = = old.getLength() + 1 **and**

this.contains(element) = = true

pre and post conditions for the overriding method *addElement* in the derived class *Set*,

pre: element != null **and** this.contains(element) = = false

post: this.getLength() = = old.getLength() + 1 **and**

this.contains(element) = = true

When replacing a method's implementation through overriding, the pre and post conditions of the overriding methods in a derived class might also replace those of the base method. But, when doing so, one must only replace preconditions with weaker preconditions and post conditions with stronger post conditions. In this example, the pre condition of the overriding method in the derived class is stronger than that of the base method in its super class although the post conditions for both methods are the same, therefore, the LSP is violated and the inheritance between these two classes might be flawed.

6.3.2 Data Flow Anomaly

A data flow anomaly refers to an inaccurate use of data. In general, a data flow anomaly could happen in various ways in class inheritance, and becomes even more frequent as the use of polymorphism and overridden methods increases. Two common examples that may involve potential data flow anomalies are described as follows:

(1) *State variable confusion*. The introduction of a new state variable in the subclass having the same name as an existing variable in the super class may easily result in a data flow anomaly. Suppose a state variable x is defined in the super class A, and another state variable with the same name x is defined in the subclass B. The corresponding code fragment is given below:

```
public class A
{
    int x;
    int y;

    public A()
    {
      x=25;
      y=15;
    }

   public void m()
   {

   }
}//end of class A
```

```
public class B extends A
{
    int x;
    int z;

    public B()
    {
       x=35;
       m();
    }

    public void m()
    {
        int w=x;//suppose this x is from B
        z=x+y+w;//suppose this x is from A
    }

    public int getZ()
    {
       return z;
    }
}//end of class B
```

```
public class Test
{
  public static void main
  (String[ ] args)
  {
        B b =new B();
        b.m();
        int v = b.getZ();
        System.out.println("The
        result is: " + v);

  }
}//end of class Test
```

In the case that both state variables x from classes A and B are considered to be referenced, and used in an overriding method *m()* of the subclass B, however, the value of all these two state variables might come from the descendent class's B if the overriding method is called in practice. Hence, by running class *Test*, the printed outcome of variable z shown in Figure 6.5 is 85 (i.e. 35+15+35=85), which is supposed to be 75 (i.e. 25+15+35=75). In this case, a data flow anomaly will occur unless these two state variables are explicitly indicated, such as *super.x* or *this.x*. After rewriting code in the *m()* of the subclass B as follows:

```
……
public void m()
  {
      int w=this.x;//this x is from B
      z=super.x+y+w;//this x is from A
  }
……
```

then, recompiling class B and running class *Test* again, the calculated outcome for variable *v* is 75 this time, which is also shown in Figure 6.5.

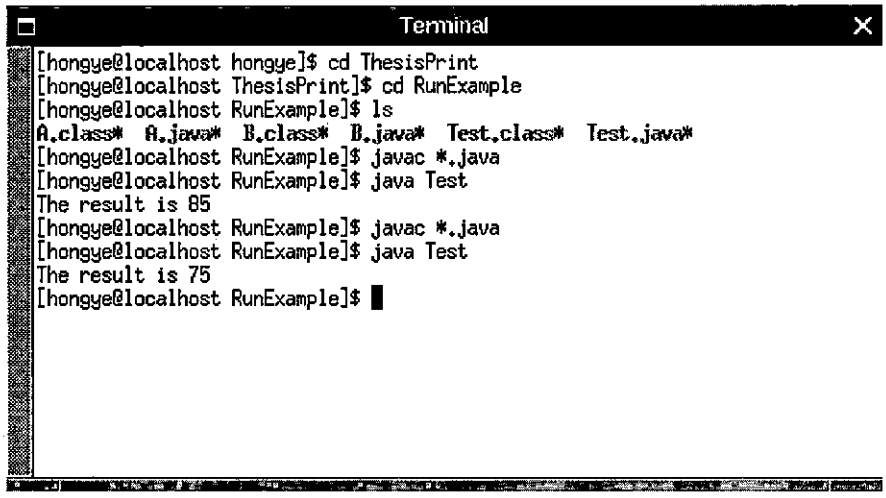


Figure 6.5 A Snapshot Showing Two Different Outcome for the Variable z

(2) *State variable accessibility*. This kind of data flow anomaly is a function of the depth of inheritance trees. Consider an inheritance hierarchy involving three classes, *A*, *B* and *C*, with polymorphic method *m()*. *A* is the root class, *B* is the subclass of class *A*, and *C* is the subclass of class *B* as depicted by Figure 6.6. The state variable *x* in the ancestor class *A* is declared as private and defined by the polymorphic method denoted as *A::m()*. Because *A::x* is private, it is impossible for the subclasses to directly define the state variable *A::x*; instead, they can define *A::x* by using the overriding methods. Suppose that we want to define *A::x* from class *C*, and accordingly we provide an overriding method in class *C* denoted as *C::m()*. At that time, there is no overriding method in class *B* (see Figure 6.6(a)). So, *C::m()* can call *A::m()* directly to modify *A::x*. Later, suppose that class *B* also adds an overriding method *m* denoted as *B::m()* as shown in Figure 6.6(b). In this circumstance, for *C::m()* to properly define *A::x*, method *C::m()* needs to call *B::m()* first, then from

B::m() to call *A::m()*. Otherwise, the data flow anomaly may occur if method *C::m()* still tries to call *A::m()* directly.

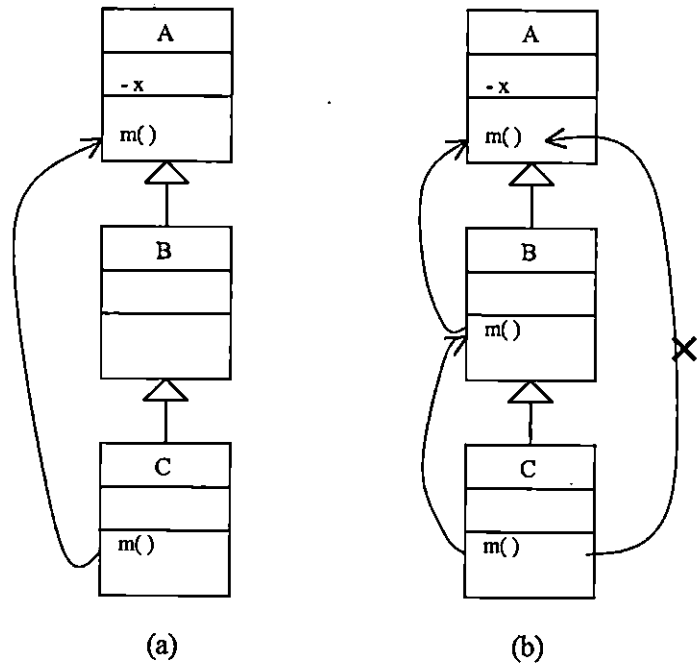


Figure 6.6 State Variable Accessibility Anomaly

6.4 Testing Inconsistent Overriding

Testing inconsistent overriding is intended to test class inheritance designed for LSP compliance. In this approach, test cases are defined in particular for those overridden methods in a class inheritance hierarchy because it is believed that misuse of method overriding in derived classes is most likely to result in LSP violation. Thus, it is strongly suggested that not only should any overridden and overriding methods in the class inheritance hierarchy be tested separately, but also additional test cases especially for these methods must be generated.

To generate test cases for the overridden and overriding methods, first, we may still use the approach presented previously to define the pre and post condition pairs for each overriding and overridden method. Then, the weakest precondition and the strongest post condition among those pre and post conditions are selected to form a new pre and post condition pair, which will be used for test case generation. Test execution is based on the interactions between objects instead of the individual

method. As we know, an object interaction is a request from one object often called the *client* to another object often called the *server* to perform some operations (*the services*) provided by the receiver [02]. In most object-oriented languages, this can be achieved by message calls between objects. A focus of the object interaction is whether the client object meets the preconditions of the method in a server object.

In order to verify that an overriding implementation in a class inheritance hierarchy works properly, we can simulate object interaction by creating a client object (test driver class) that will send a message call to the overridden and overriding methods associated with different contexts of the object involved in class inheritance. Each successful execution of an overridden or overriding method of the corresponding object must satisfy the precondition of the overridden method in the base class that should be weaker than that of any of its derived classes. Then, the action for each overridden and overriding method must lead to a state corresponding to the strongest post conditions among them.

6.5 Data Flow Testing

A computer program contains two essential elements: *code*, the sequence of computer instructions, and *data*, the information on which the instructions operate. This is particularly important in object-oriented programs because code and data are merged into a single entity, i.e. an object. In procedural programs, code and data have strong independence. Code can still be executed without values being assigned for some variables. However, in object-oriented programs, missing data values may result in a failure and interrupt the execution of a program. Data flow testing is used to test how values of data that are associated with variables can affect the execution of programs. Anomalies that data flow testing can reveal are usually that the data values are not available when they are supposed to be.

Data flow testing takes into account variable occurrences in the program as Beizer and Huang described in their respective data flow anomaly models [01, 08]. Each variable occurrence can be classified as either a definition or a use occurrence, usually denoted by lower case letters d and u; d denotes that the value of a variable is defined, and u denotes that the value of a variable is used. Use of a variable can be further classified as either a computation use or a predicate use. If the value of a variable is used for computation, and the variable usually appears on an assignment statement,

the use of the variable is called a *computation use (c-uses)*. Otherwise, if the value of a variable is used to decide the outcome of a predicate statement for directing the control flow, the use of the variable is called a *predicate use (p-uses)*.

Data flow testing is usually used to track the definition and use of a variable (*def-use* pair), in which abnormal actions could happen, during program execution. In general, three important kinds of data flow anomalies can be distinguished based on the occurrence of a def-use variable [01]:

(1) Using a variable (c-use or p-use) which is not yet defined

(2) Redefining a variable before it is used (define a variable twice)

(3) Defining a variable without using it

Data-flow oriented test selection criteria proposed by most researchers are *all-du paths, all-u, all-p-use/some-c-use and all-c-use/some-p-use, all-d* etc, among which *all-u and all-d paths* are the strongest ones [01, 09, 10]. These criteria are intended to select a set of paths that cover all occurrences for all the definitions and uses of the given variables. Test data should cause the execution of the implementation under test along the selected set of paths. Detailed information on these criteria will be provided in a later section dealing with test coverage.

When doing data flow testing, data flow graphs are often not used, rather, control flow graphs are typically used to show what happens to the variables of interest. A well-designed control flow graph is essential for effective data flow testing.

6.6 Application of Data Flow Testing to Class Inheritance Testing

This section presents an approach to test data flow anomalies in polymorphic and overridden methods during object interaction in a class inheritance hierarchy. Data flow testing, as its name indicates, is abled for revealing data flow anomalies. As mentioned earlier, performing data flow testing requires the construction of a control flow graph. In fact, any algorithm or way for selecting nodes and links between nodes may lead to a corresponding test strategy.

The method call graph shown in Figure 6.7 can be built and used as the control flow graph. The method in each class can be modeled as a node, and a call from a calling method to a called method can be modeled as a link with the arrow pointing to the called method. Due to the existence of overridden and overriding methods

involved in class inheritance, the name of method in the graph should have a prefix indicating the context of an object with which the method should be associated in order to distinguish it. For example, in Figure 6.7, B::m() means the method m() in the class B. The data flow graph can be generated based on specification and implementation.

Each node may be associated with some state variables either defined or used. After the call graphs are defined, we can list all the variables associated with the specified method (du-pairs) in terms of test requirements or specifications as depicted in Table 6.1. Test case generation is based on static analysis of the method call graph. In this case, the strongest test data selection criteria, all possible du-paths criteria, are used to generate two executed set of paths, each of which includes a set of methods call. An assumption is made that before the first method in each object is called, the corresponding object will be first instantiated and some variables may be initialized. For example, {B::g(), B::h(); B:m(); B::n()} and {B::g(), C::h(); C::m(); B:m(); B::n()}. From the first path, we can deduce this path covers the du-pairs for the given variable x and y since the path includes method B::h() which defines x and y, and method B::n() which uses x and y. But, the second path does not cover du-pairs for the x and y because of missing the method B::h(), this probably will cause data flow faults. In order to avoid this error, the x and y must be defined before the execution of the method B::n(), for example, if we define the x and y in the method of the class C::h() or C::m(), this anomaly could be corrected.

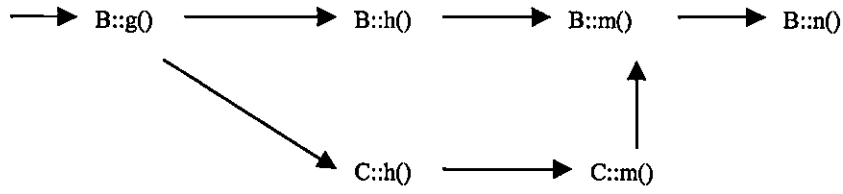


Figure 6.7 Methods Call Graph

Variables	Definition	use
x	B::h()	B::n()
y	B::h()	B::n()
u	C::h()	C::m()

Table 6.1 DU-Pairs

The test case execution for data flow testing may require a test driver concurrently simulating the method calls according to the def-clear paths, or instrumentation in source code to detect the correct value of a variable.

6.7 Summary

Based on "substitutability" in subtype inheritance, an instance of the parent class occurring in a test suite ought to be replaced by an instance of each of its subclasses. Although in principle this is very useful and we can rerun a super-class test suite against any instance of its subclasses, testing the interaction between the super-class and the subclass methods is far from what these test suites can achieve because object-oriented programs so often use inheritance combined with polymorphism and overridden methods. Hence, we need to define new test techniques and develop additional test cases in order to deal with them. To do so, two test strategies for testing class inheritance have been presented: (1) an object interaction test strategy for the test of LSP-conformance, and (2) a data flow testing strategy for revealing data flow anomalies involved in object interaction.

Chapter 7

Test Coverage Analysis

This chapter presents test coverage analysis along with the definition of adequacy criteria. The purpose of the adequacy criteria is to facilitate the generation of complete test sets achieving higher code coverage. Code coverage metrics are also discussed in order to evaluate the effectiveness of test cases during code execution.

7.1 Test Coverage Overview

Test coverage analysis is a process commonly used to create adequacy criteria and increase the confidence that an implementation has been thoroughly tested. For example, test cases can be generated by means of a test strategy, but in the later execution of programs, some code may never be executed due to the possibility of missing test cases. At this stage, one hardly knows how effective test cases are until one finds what code is, or isn't executed. This dilemma may be avoided by defining efficient adequacy criteria based on test requirements and/or analytical techniques. So far, there is a large variety of test coverage models or tools available, both commercially and academically. The three basic criteria commonly used are *Statement Coverage*, *Decision Coverage* and *Condition Coverage*. It has been suggested that the combination of these three methods can achieve 80-90% or more coverage [34].

Even though available criteria may achieve a high coverage percentage, we must admit that there is still no absolute guarantee of error free software. Hence, one should never rely solely on test coverage to devise test cases. Most researchers argue that the use of test coverage in combination with the test strategy, can play an important role

because it can mitigate the impact of uncovered code during software testing, and especially help the tester find some rational points at which to stop testing. Interested readers are referred to Binder [03] for the correlation between when to stop testing and test coverage.

7.2 Coverage Criteria for State-Based Test Strategies

7.2.1 The Transition Coverage Criterion

The transition coverage criterion is defined in terms of the state diagram of a class. At a minimum, a tester should test every transition in the state diagram at least once. Transition coverage is analogous to statement or branch coverage. Basically, a class, the basic unit in object-oriented programs, contains five kinds of methods namely constructor and destructor methods, mutator (e.g. *set* methods) and accessor methods (e.g. *get* methods), and user-defined methods. All of them except accessor methods can change the state of an object, however, not all of them may appear on the state diagram, for example, mutator methods (*set* methods) are usually not defined in a statechart diagram since they don't reflect the functionality of a class, and testing of these transitions can be achieved after the code is implemented by simply selecting input/output values. Therefore, for the transition-based test strategy proposed in this thesis we can define the transition coverage criterion as follows:

> ***Transition Coverage Criterion:***
> *For each transition in a state diagram, there must exist a corresponding set of test cases for it.*

7.2.2 The Precondition Coverage Criterion

The precondition coverage is defined in terms of a precondition in the pre-post condition pair of a transition. This coverage is similar to condition coverage. To better convey our idea for this kind of coverage, we define the following terms:

(1) *Boolean Operators* are AND, OR and NOT.

(2) A predicate is a boolean function

(3) A DNF (Disjunctive Normal Form) expression is a boolean expression that is composed of literals and only AND operators.

(4) A literal is a simple boolean expression (in which there is no operator involved)

A pre or post condition is composed of predicates. A pre condition can be decomposed to several DNF expressions. Test cases are generated based on each DNF expression separately. The test cases generated at least should cover every DNF expression in the pre condition, that is, make each DNF expression true at least once without considering other DNF expressions. This coverage criterion can be finally defined as follows:

Precondition Coverage Criterion:

For each DNF expression in the precondition on a transition, there must exist a corresponding test case for it.

7.2.3 The DNF Coverage Criterion

The DNF coverage is an extension of the precondition coverage. As mentioned previously, a DNF is composed of conditions. The DNF coverage criterion is based on the rationale that each condition should be tested independently without interference from other conditions. In other words, each condition in a DNF must independently affect the outcome of the DNF. Effectively, each unique combination of conditions must be tested.

However, finding the combination of all conditions that satisfy the DNF coverage is sometimes laborious and errors are common since the number of combinations will increase exponentially as the number of conditions increase. In order to obtain possible combinations of conditions, perhaps the simplest and most straightforward way is to use a decision table. Consider a DNF involving two conditions, A and B, there will be four possible unique combinations of the two conditions which determine the outcome of the decision table, each of which is monitored, and must be tested, for satisfying the coverage. Table 7.1 shows the four combinations. Given n conditions involved in a DNF, the number of such unique combination of conditions is 2^n.

In fact, the number of unique combination may be less than 2^n in practice due to the mechanism of *short circuiting* logic operators provided by some languages, such as

Conditions		DNF Outcome
A	B	AB
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Table 7.1 Combinations of Conditions for A DNF

Java and C++. That is, evaluating their second operand depends on the outcome of evaluating the first operand. For example, for a piece of Java code,

```
if (A&&B){
        ..... statement 1 .....
}
else
{
        ..... statement 2 .....
}
```

if the condition A resolves to false, the program will directly execute the *statement 2* without evaluating the value of condition B. Therefore, a large number of conditions may significantly reduce the number of combinations. Table 7.2 shows the refined combination of conditions for the above example in which the number of combinations is reduced from four to three.

Conditions		DNF Outcome
A	B	AB
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	Don't Care	FALSE

Table 7.2 Refined Combinations of Conditions for A DNF

Now, the DNF coverage criterion can be defined as follows:

DNF Coverage Criterion:

For each DNF, there should be test cases that make all the conditions involved true and each of them false at least once.

7.3 Coverage Criteria for Class-Based Test Strategies

7.3.1 The Association Coverage Criterion

Association coverage is defined for the relational test strategy based on a class diagram, and is similar to the transition coverage criterion. The main difference is that they are based on different models. One is based on state diagram and the other is based on a class diagram. The association coverage criterion is defined as follows:

Association Coverage Criterion:

For each association in a class diagram, there must exist a corresponding set of test cases for it.

7.3.2 The All-Paths Coverage Criterion For Associations

In most cases, testers will address this situation, that is, they will want to test the relation of two classes that are indirectly connected. The above relation coverage criterion tests a relation independently, but does not test sequence of relations; thus, some faults may not be adequately revealed. Consider the following three related classes as shown in Figure 7.1. To investigate these kinds of errors, path coverage requires that complete paths between any two classes should be taken, and can be defined as follows:

Path Coverage Criterion:

For each sequence of relations (i.e. path) from class A to C, there must exist a corresponding set of test cases for the path.

For the above example, testing relations between A and B requires inputs that satisfy the six following sequences of relations: <R1, Ri>, <R1, Rii>, <R1, Riii>, <R2, Ri>, <R2, Rii> and <R2, Riii>.

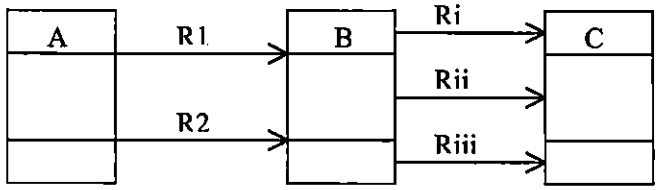


Figure 7.1 Implicit Relations Between Classes

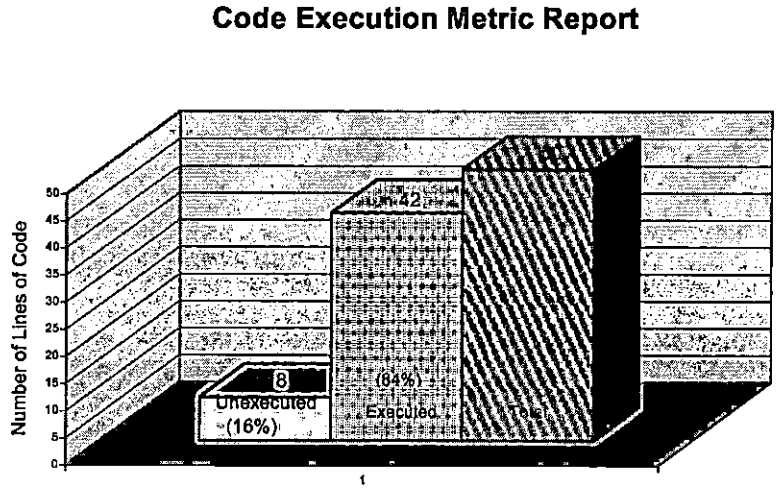
7.4 Code Execution Metrics

Code execution metrics provide measures of whether or not a code segment has been covered during the execution of a given test case. It is often expressed in terms of the ratio of the number of code segments executed or evaluated to the total number of code segments. However, in the course of code execution, some code segments may be executed several times (e.g. loops); if every execution is counted for them, the actual ratio of code execution coverage may not be correctly reflected. Hence, code execution metrics for code coverage usually refers to the unique code coverage. To measure how comprehensive a test execution is, we can use the code execution coverage metric that can be expressed as a percentage as follow:

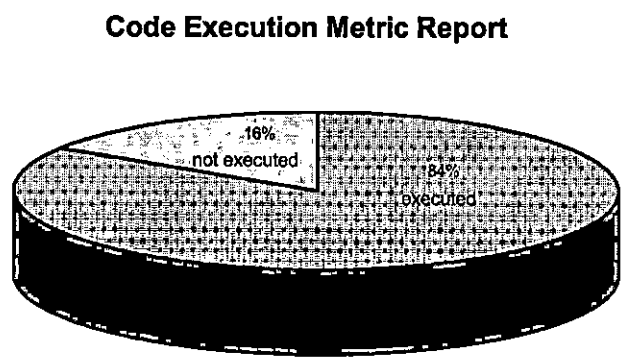
$$\text{Code execution coverage} = \frac{\text{Number of lines of code executed once}}{\text{Total number of lines of code}} * 100\%$$

Code execution metrics can help testers or managers know what code isn't covered, and then determine what additional test cases need to be developed. This process can help find "dead code", which will never be executed. Code execution metrics are also suitable to show statement coverage, branch coverage or path coverage. In addition, code execution metrics and the adequate coverage criteria complement each other. Coverage criteria are used as guidance to generate more complete and efficient test cases, but how complete the test cases really are can be reflected through code execution metrics. Thus, the combined use of both is believed to bring good test coverage.

The results of code execution measurements are usually reported concisely by using graphs such as bar charts, or pie charts. Figure 7.2 shows samples of bar chart and pie chart report styles for reporting code coverage metrics.



(a) Bar Chart



(b) Pie Chart

Figure 7.2 Samples of Code Coverage Metric Report Charts

7.5 Summary

Test coverage analysis is a structural testing technique that helps eliminate gaps in a test suite. Adequate criteria are traditionally defined based on code. This chapter presents several adequacy criteria, such as the transition coverage criterion, the pre-

condition coverage criterion, DNF coverage criteria and the relation coverage criterion, and the all-paths coverage criteria. These criteria are defined in terms of specifications/requirements, and in more precise manner to evaluate test cases generated especially from either the state-based transition test strategy or class-based relational test strategy in this thesis, which are based on UML state diagram or class diagrams, respectively. Code execution metrics are considered among the most effective ways to evaluate the efficiency of test cases applied to a software application.

Chapter 8

PrUDE

Test automation tools are needed to help testers improve productivity. This chapter presents a prototype software verification and validation (V&V) tool named PrUDE which is being developed by the ISOT group at the Department of Electrical and Computer Engineering, University of Victoria (http://www.isot.ece.uvic.ca). The tool supports test case generation and execution for object-oriented programs. In particular, the implementation in PrUDE of the test strategies presented in this thesis is discussed.

8.1 Overview of PrUDE

PrUDE (The Precise UML Development Environment) is an integrated UML-based verification and validation (V&V) framework. One of the functions of the PrUDE tool suite is to automate, either fully or partially, some of the most important phases of the testing process including test case generation, test execution and test result reporting. Test cases are generated based on a valid UML specification. The UML specification is rigorously verified using consistency-checking, proof-checking or model-checking. Model checking and proof checking are based on the PVS toolkit, and can be run at the back end. This allows users to deal with only their familiar design notations (e.g. UML) without handling too many formal artifacts. Figure 8.1 gives an overview of the V&V strategy underlying the PrUDE platform.

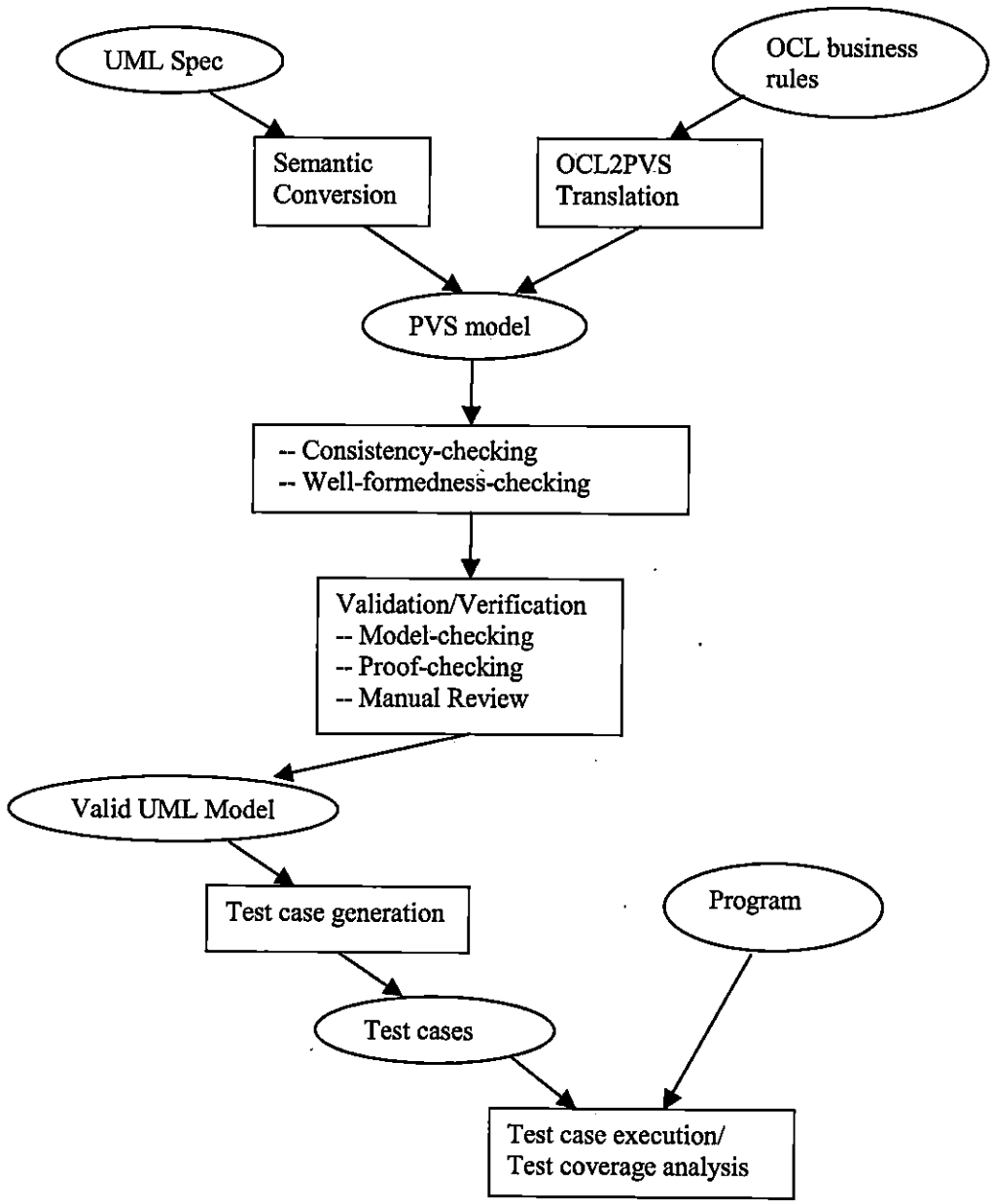


Figure 8.1 V&V Strategy Using the PrUDE Platform

In practice, users can build a UML model using their favourite UML tools, such as Rational Rose or ArgoUML, and then submit the model to PrUDE for verification. The model must first be converted to the XMI format. Most UML tools support XMI as a model exchange format. For tools such as Rational Rose, the UML to XMI converter

needs to be installed with the UML tools. For tools such as ArgoUML, the conversion is done automatically when the model is saved. Then, using PrUDE, a PVS model representing the formal semantics of the UML model, which serves as a basis for the formal validation of the UML model can be automatically generated. In principle, the UML model may need to be augmented by expressing business rules using an assertion language such as the Object Constraints Language (OCL). Corresponding expressions are converted in PVS and integrated to the semantic model. The PVS toolkit can then be invoked in order to check the model and the system properties.

The test component of PrUDE allows the generation of suitable test cases from a valid UML model. Test cases are derived from various constraints related to the model, such as pre and post conditions, and invariants. Once the test cases have been generated, PrUDE also provides a test execution component that can be invoked to execute the test cases for Java implementations. After testing is completed, a test report can be generated using PrUDE.

8.2 Main Functionalities of PrUDE

The main functionalities of PrUDE are carried out by four key components namely the *Specifier*, *Analyzer*, *Generator* and *Tester*, each of which is briefly reviewed in the following paragraphs.

Specifier. The *Specifier* component is mainly used to generate the PVS formal specification for the UML model and add the complementary definition into the generated PVS semantics. An XMI file that contains the UML model is first imported and parsed by the *Specifier* component. Then, the PVS semantics are generated automatically.

Analyzer. The *Analyzer* component is used to conduct the major verification tasks. It is also used to edit the PVS semantics file previously generated from the *Specifier* component, and if needed, to integrate the system properties into the PVS semantics. The PVS semantics, including the system properties for the UML model, are now ready for proof checking. During the verification process, the PVS proof-checking system can be invoked directly from this component, and will run at the back end either in batch mode

or interactively. The final verification results can be displayed in the reporting text box of PrUDE.

Generator. The *Generator* component is used for test case generation based on constraints collected from valid UML specifications. It implements in the current version two test strategies: the transition test strategy for a UML state diagram, and the relational test strategy for a UML class diagram. The transition test strategy is used for method testing at the class level, so it is a unit testing strategy. The relational test strategy focuses on testing of the relations between classes, and is used thereby for integration testing. The *Generator* uses a mix of automation and heuristics to generate test cases. The selection of test values is eventually contained in the domain matrix.

Tester. The *Tester* component is used for test case execution of Java implementations. The *Tester* uses the Java reflection API to directly access and modify the internal states of objects without needing accessor and mutator methods (e.g. get and set methods). The tester also provides support for test execution reporting.

8.3 Implementation and Operational Features of PrUDE

The detailed presentation of all the features of PrUDE is beyond the scope of this thesis. Implementations related to the transition test strategy, including test case generation and execution, is described in detail in this section.

8.3.1 The Main Interface of PrUDE

PrUDE has a user-friendly interface as shown in Figure 8.2. At the top of the window are the menu and tool bar providing various operations such as create, save or exit a project. The subjacent area of the toolbar is divided into four major sections as displayed. The left part, used to browse the tree structure of a project created by users, includes two sections named Specification and Program, which are used respectively for specification verification and program testing. In both cases, users can select a node in a tree and open a popup menu by right-clicking, and then conduct the corresponding V&V or testing activities by choosing a submenu in the popup menu. The upper right part consists of four

tabbed panels, each of which is related to a different component (e.g. tester, specifier). For example, the *Specifier* tab is used to edit the generated PVS file, and the *Tester* tab can edit the source code of a program under test. The lower right part is a text area used for displaying the run-time status and the log messages.

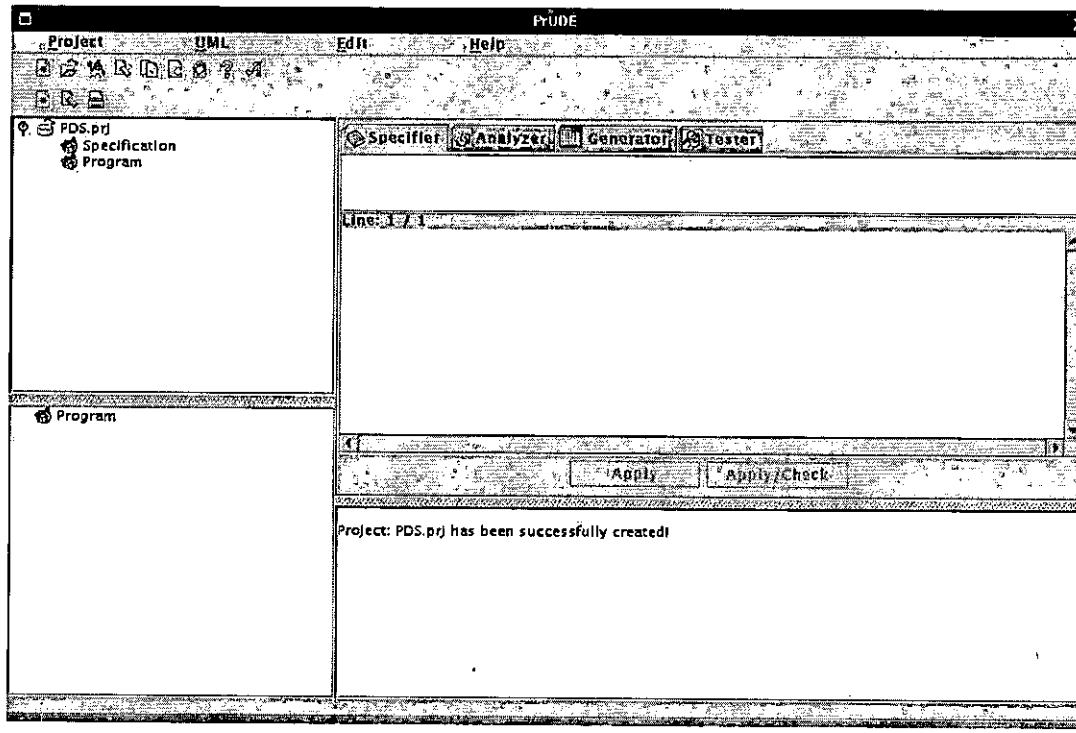


Figure 8.2 A Snapshot of PrUDE Main Interface

8.3.2 Importing Program Under Testing

To conduct the testing activities, the program under test (PUT) is initially imported into PrUDE. One of the classes, *TestInterface* in PrUDE using the Java class loading mechanism and Java reflection, has been implemented for this purpose. Given the specified URL for classes and resources (e.g. the directory including the Java source and class files), all the Java files underlying that URL can be searched and found by overriding the method *findClass* of *java.lang.ClassLoader*. Once a class is located, the information related to the class such as class name, the names of the methods of the class, or package name, if applicable, can be easily obtained by using Java reflection. In

PrUDE, PUT is represented hierarchically in a tree structure as shown in Figure 8.3. For example, this tree shows that there is a method named *getRight* in a class named *vvot.test.SecurityProfile.*

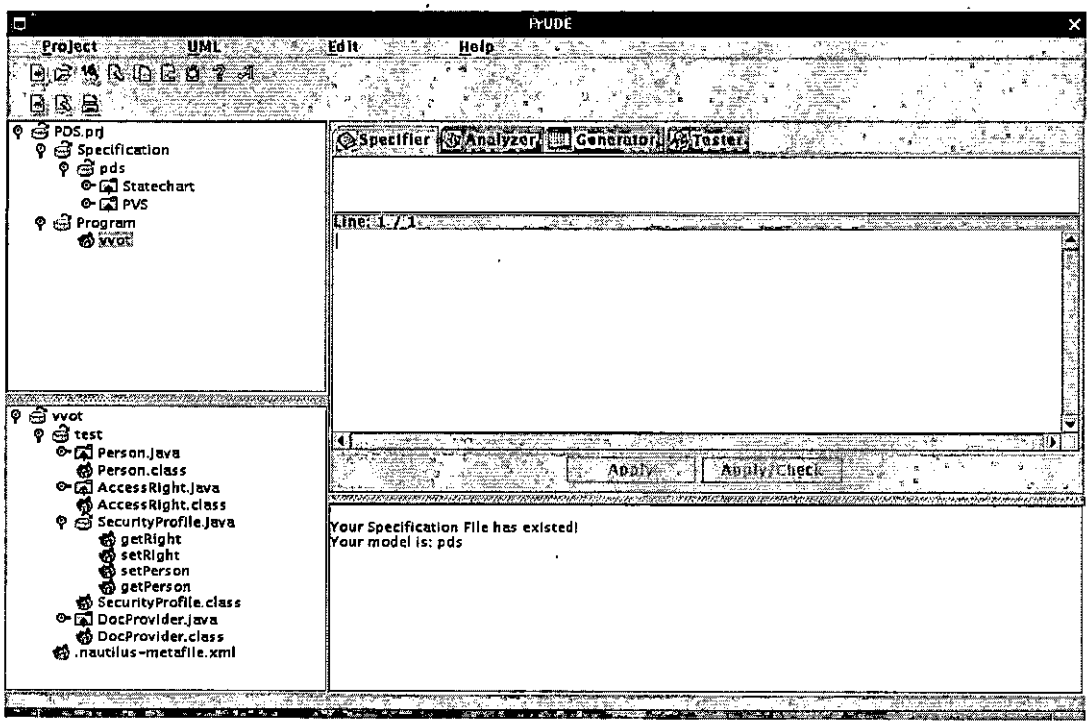


Figure 8.3 Import Programs Under Test

8.3.3 Test Case Generation

Test cases generation is based on preconditions in the transition test strategy. There are two different approaches for test case generation, one for primitive variables and another for object variables, which have also been implemented in two different software components; both use heuristics to guide users in the process.

The key difference between the two approaches is how domain analysis is used to generate test cases. Both approaches start by choosing a method under test corresponding to a transition in a UML statechart diagram, then, based on the states, guard conditions and actions involved in the transition, pre and post condition pairs are generated. If there are only primitive variables involved in the pre and post conditions, the traditional

domain analysis is applied by identifying the boundaries of the variables. Figures 8.4 shows a domain matrix in PrUDE representing the generated test cases for primitive variables. However, in the case where there are object variables involved in the preconditions, domain analysis cannot be used directly. Instead, the object decision tree first needs to be constructed. Then the domain analysis for object variables is conducted by reusing the object instances defined in the decision tree. Figure 8.5 shows a domain matrix representing test cases generated for object variables in PrUDE. The left part of the window displays the hierarchy of the tree structure, the lower right part of the window shows the construction of the corresponding instances in the decision tree, and the upper right part of the window shows the domain matrix in which some variables are assigned object values.

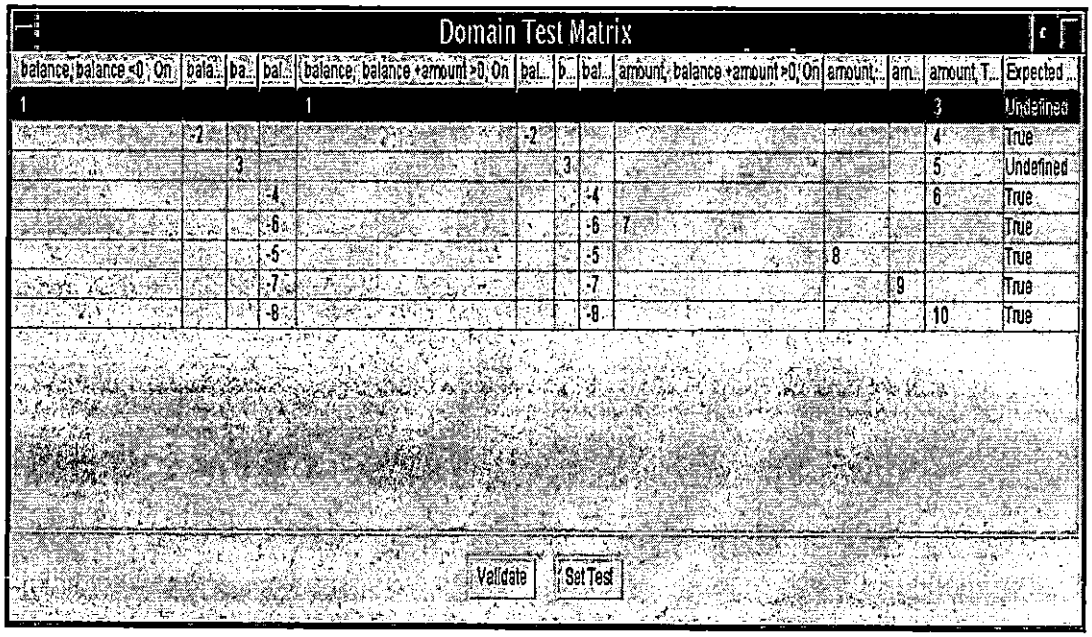


Figure 8.4 Domain Matrix for Primitive Variables

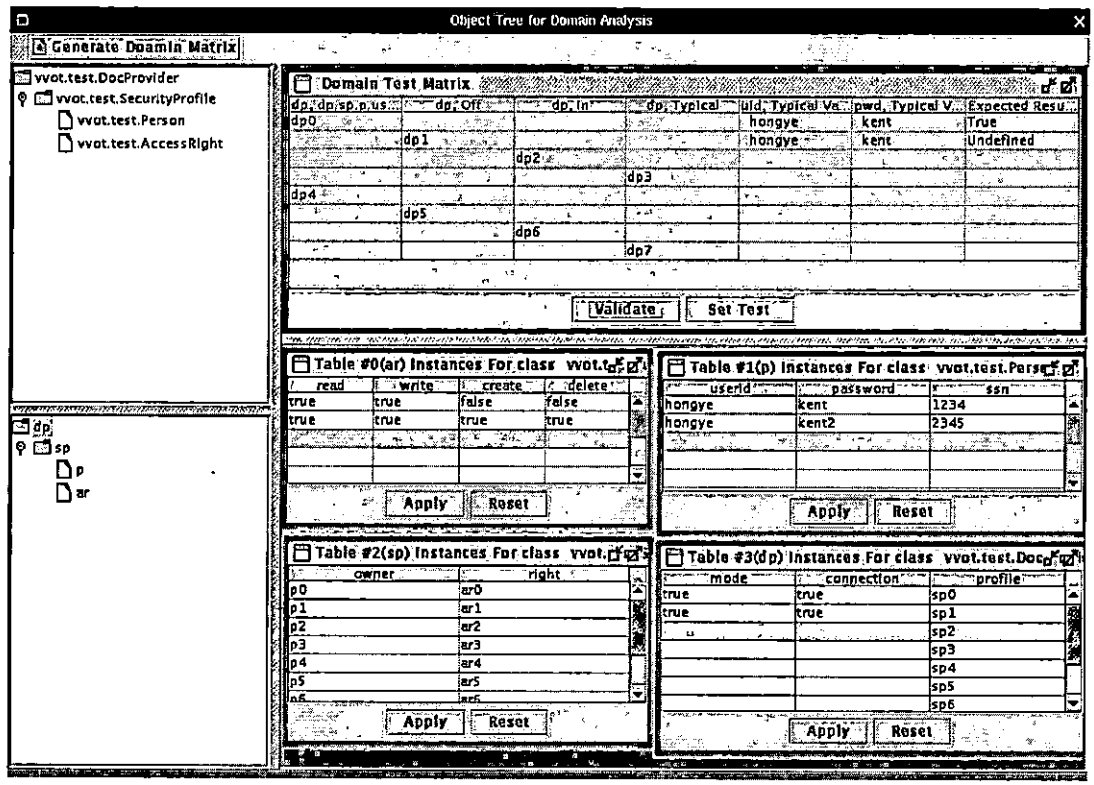


Figure 8.5 Domain Matrix for Object Variables

8.3.4 Test Execution

After test case generation, PrUDE also provides text execution for Java programs. One of the classes, *TestRun.java* in PrUDE, has been implemented for this purpose by mainly using the Java reflection API that can access internal information of an object and invoke a method of the object directly at run time so that building test drivers or writing test scripts is unnecessary. Test execution starts by creating a fresh instance of the class containing the method under test, then the initial values of the state variables are set based on the test cases. After execution, test results are evaluated based on post conditions. Figure 8.6 shows the result of a test execution in PrUDE.

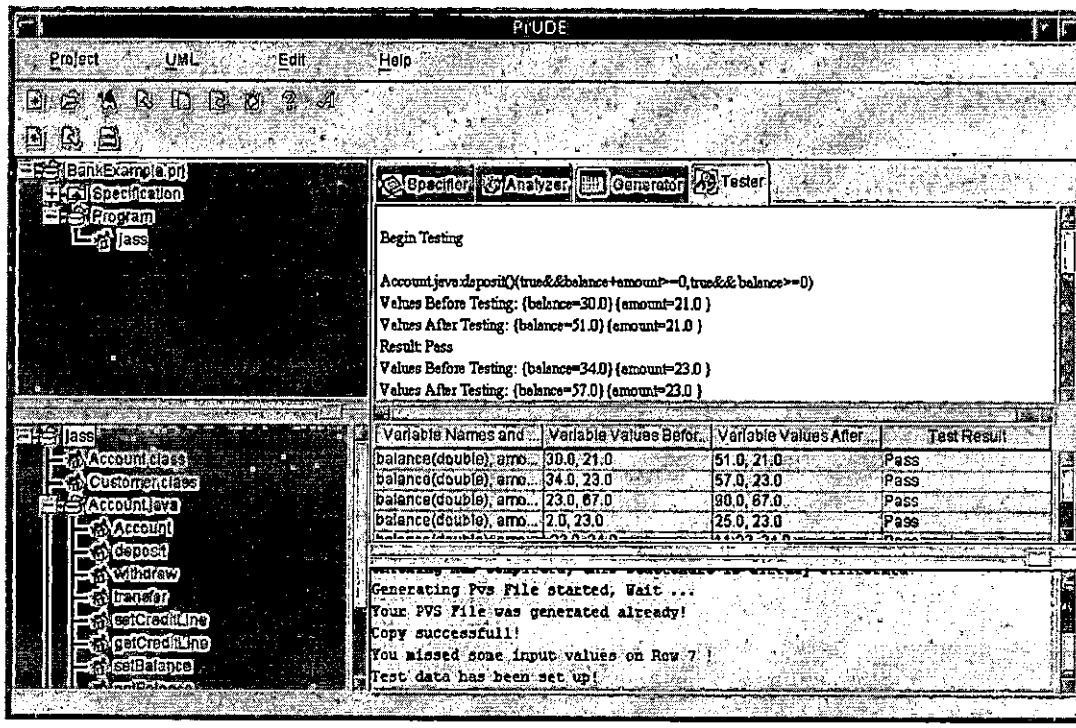


Figure 8.6 Test Results Displayed in PrUDE after Test Case Execution

8.4 Summary

Formal verification and validation may significantly improve software quality during software development. However, using abstract formal languages and notations to design software systems is not likely to be widely used in the software industry since most designers and developers lack background in formal methods. PrUDE provides, in this context, an automated environment that bridges the gap between formal verification techniques such as proof-checking and semi-formal verification techniques such as testing.

Chapter 9

Example – A Patient Document System (PDS)

In order to illustrate our approach, this chapter describes a case study on a critical system that provides a secure patient document service. The process of test case generation based on the state transition strategy is mainly discussed. Test result and coverage analysis are also presented.

9.1 Functional Requirements

Binkadi Life, an insurance company needs to rapidly create a health care marketplace. The central and initial component of the marketplace would be a patient document service (PDS) that provides support for the company's 1,000,000 care providers, benefit coordinators and agents. The initial version of the PDS will only maintain secure patient medical records and make them available to authorized persons worldwide. Subsequent versions are expected to expand the basic functionality with several new services.

The main function of the PDS system is to provide secure access to patient medical records worldwide. The system must provide special protection features dealing with suspicious users and disclosure of unauthorized information. The actors involved in this system are the patients, patients' relatives and friends, doctors, and site administrators. The main resources to be secured are the medical records of patients. A patient may choose a unique family doctor who is automatically granted the right to read and modify his medical records. Only authorized doctors can read or modify a medical record. Every

doctor is solely responsible for the modifications that he makes to a medical record, and the system is expected to enforce this responsibility. An authorized doctor is a registered doctor that a patient has chosen either as his family doctor or as a "guest" doctor, e.g. a specialist, or for travel reasons or unavailability of the family doctor. The patient is the only person that is allowed to choose his own doctor. A patient may have read access to his own medical record, but he cannot modify it. He may grant read access to his friends and family members. The site administrator is the only person who can create, delete, read and modify a patient record. The system is required to be secure, i.e. it must ensure that authenticity, integrity, confidentiality, and authorization are always preserved.

9.2 UML Design Specification

The UML specification consisting of a variety of diagrams describes the PDS system from different view. We present in the following some of these diagrams.

9.2.1 Class Diagram

The class diagram provided in Figure 9.1 depicts the structural components of the system described above. Potential users of the system are represented by the *Patient*, *Doctor*, *Administrator*, and *Friend* classes. These classes are subclasses of the *Person* class that describes a set of common attributes. The *DocProvider* class manages the access to and delivery of medical records, which are described by the *MedicalRecord* class. The *SecurityProfile* of a user is defined as a set of *AccessRight* associated to the *Person* class.

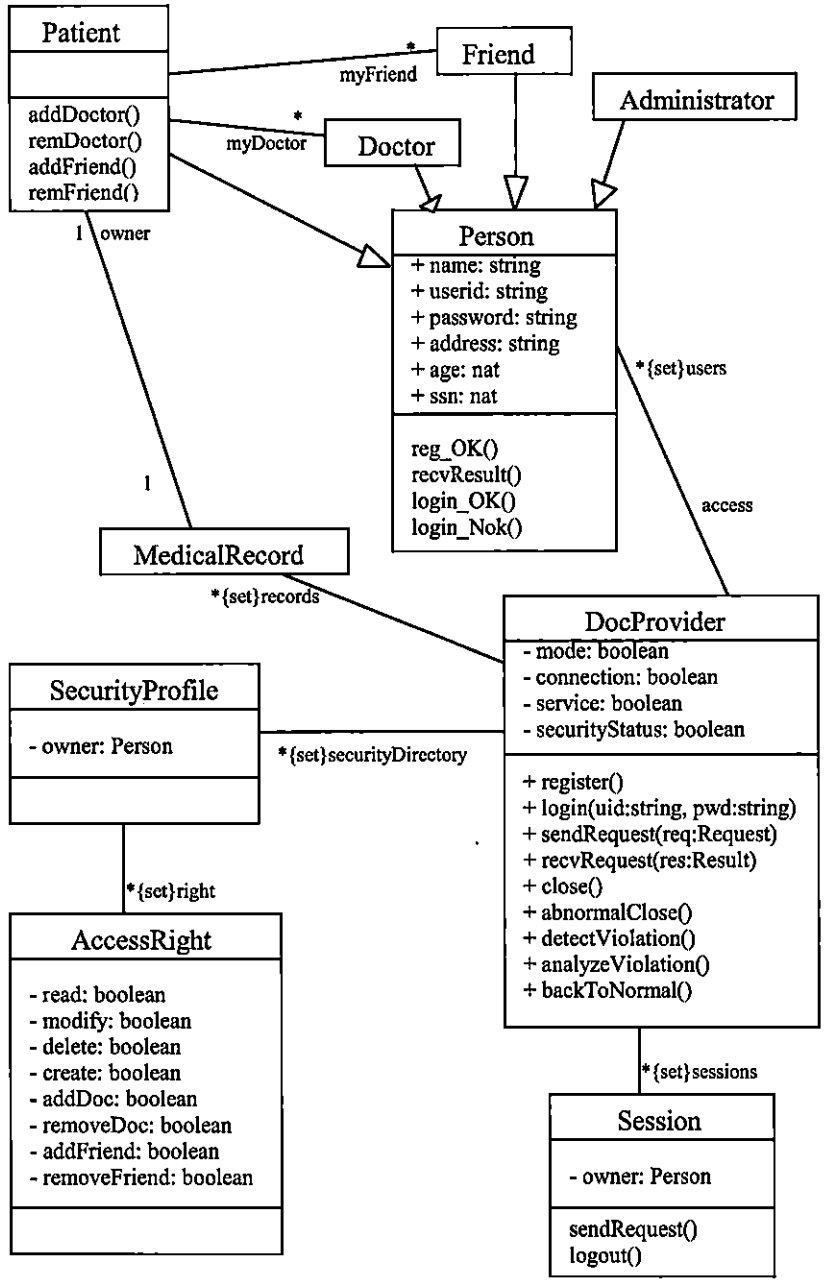


Figure 9.1 Class Diagram of Patient Document Service

9.2.2 State Diagram

The statechart diagram shown in Figure 9.2 describes the dynamic behavior of the *DocProvider* class. The system starts in an initial state where security parameters are initialized. Then, it moves to its basic operating state *NormalOperation* in which it waits

for or processes requests from users. When a request is received, the security profile of the user is checked and the request is either served or rejected. *NormalOperation* is defined as a concurrent state where connection requests and other services are dealt with simultaneously.

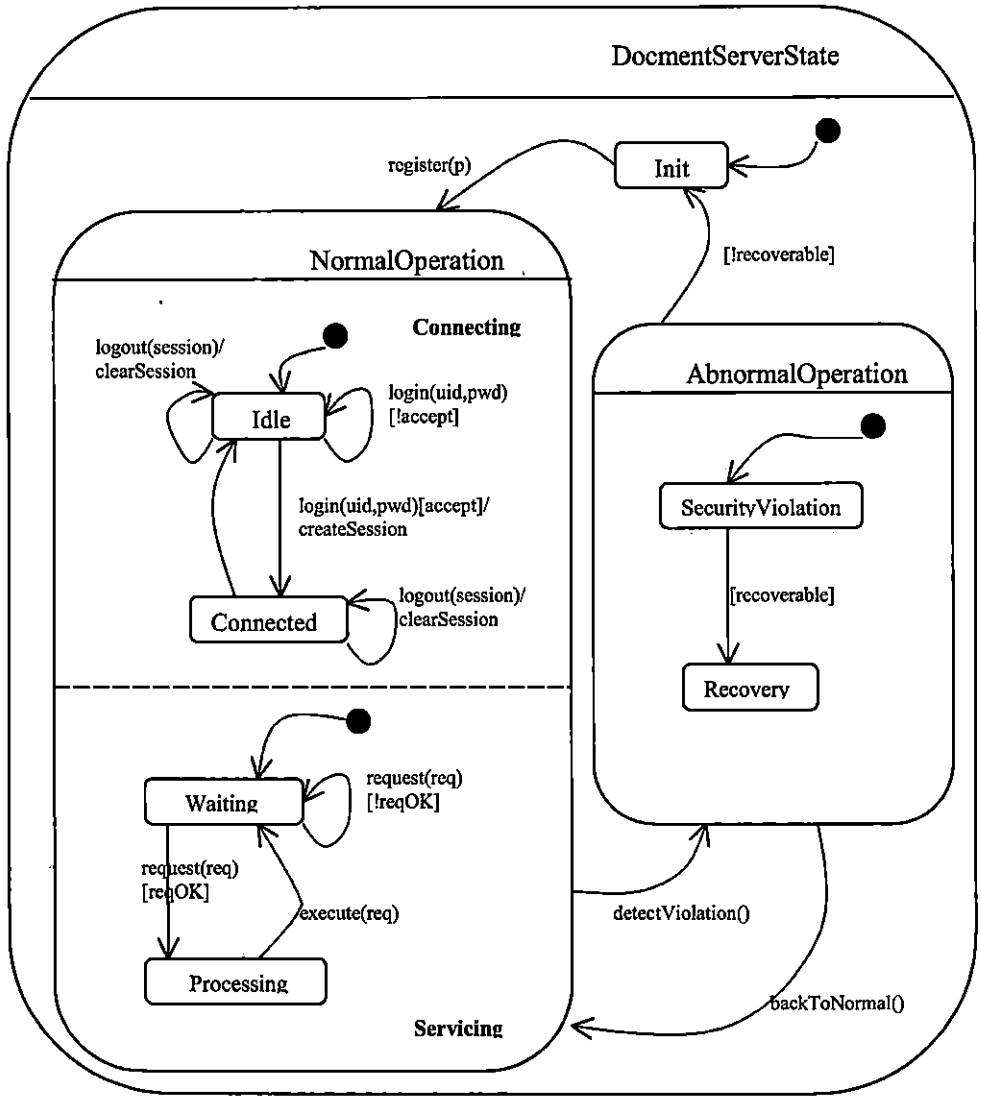


Figure 9.2 State Diagram of the Class *DocProvider*

9.2.3 Sequence Diagram

The sequence diagram shown in Figure 9.3 describes a scenario in which a user registers on a secured patient document server represented by *DocProvider*, then logs in to perform specific operations such as create, read, modify, and delete on patient's medical records depending on his access level. If the login is successful, a session object carrying the user data is created and will perform the operations on behalf of the user during the session. The session object is automatically destroyed after the user logout. That is one of several scenarios that take place during the system life.

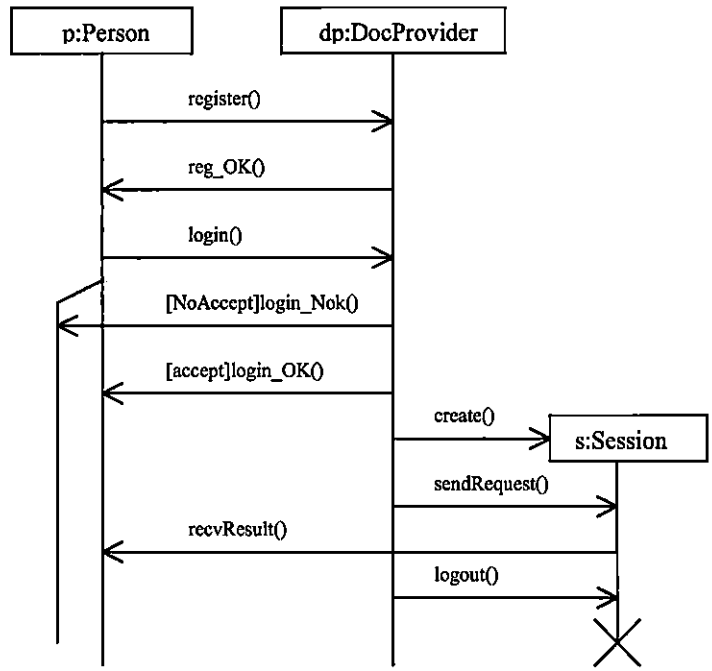


Figure 9.3 Sequence Diagram of a Login Scenario

9.2.4 Complementary Semantics

Since our goal is to use the UML specification as the basis for our testing activities, we need to augment it by improving precision and removing ambiguities. The standard UML notation provides only a partial specification of the system. The UML specification produced needs to be extended by providing complementary semantics for the elementary

features (e.g. state, actions, and conditions) and properties involved using languages such as OCL or any other mathematical or textual languages. We define in the following the complementary semantics of the states, guard conditions and actions for the statechart in Figure 9.2 using OCL. The context of the expressions is the *DocProvider* class with which the statechart is associated. For example, *predConnected* characterizes the state *Connected,* and specifies that state *Connected* is active when *DocProvider* is in its normal operation, has just established a connection, and has at least an active user. *PredAccept* corresponds to the guard condition *accept*, and ensures that for a *login* operation there is a security profile in the security database that matches the profile of the requesting user. Predicate *predCreateSession* corresponds to the outcome of the action *createSession*, which can create a new session and update the total number of current sessions in the context of *DocProvider* once the request from a user is accepted.

Context DocProvider

Predicates associated to states

predIdle(): Boolean

self.mode = true and self.connection = false

predConnected(): Boolean

self.mode = true and self.connection = true and self.users → notEmpty

predWaiting(): Boolean

self.mode = true and self.service = false

predProcessing(): Boolean

self.mode = true and self.service = true and self.sessions → notEmpty

predSecurityViolation(): Boolean

self.mode = false and self.securityStatus = false

predRecovery(): Boolean

self.mode = false and self.securityStatus = true

Predicates associated to guard conditions

```
predAccept(sp:SecurityProfile): Boolean
  exists (sp | self.securityDirectory.includes(sp) and
              sp.owner.userid=uid  and
              sp.owner.password = pwd )
```

Predicates associated to actions

```
predReqOK(sp:SecurityProfile, ac:AccessRight, req:Request): Boolean
  exists ((sp, r, req) | self.securityDirectory.includes(sp) and
                         sp.owner=req.source   and
                         sp.right.includes( ac) and
                         ac=req.action)
predCreateSession(): Boolean
  exists (self.sessions→size = self.sessions→size + 1)
```

9.3 The Process of Test Case Generation for PDS

Due to space limitations, we present in the sequel only results related to one test strategy, namely the transition test strategy.

9.3.1 Test Case Generation

The complete Java program for PDS is provided in Appendix B. At the implementation level, test cases are collected and generated based on the constraints and invariants involved in the UML and OCL specifications. In this thesis, we present in the sequel an example of test case generation involved in object domain analysis for method *login()* of the class *DocProvider* (see Figure 9.2) based on the transition test strategy. There are 2 transitions involving method *login()*: a transition that originates from state *Idle* and arrives in state *Connected*, and a self transition that loops in state *Idle*. Based on the

predicates associated with the elements of each transition, we identify the two pre-post condition pairs associated to method *login()* as follows:

DocProvider::login(uid:string, pwd:string):true

pre1: predIdle( ) **and** predAccept( )

post1: predConnected( ) **and** predCreateSession( )

DocProvider::login(uid:string, pwd:string):false

pre2: predIdle( ) **and not** predAccept( )

post2: predIdle( )

Due to the object variables involved in the pre and post condition pairs, the object domain analysis technique is used for test case generation. Having replaced the predicates involved with their respective referenced expressions based on the rules mentioned previously, we obtain the following expressions:

$\forall$ dp: DocProvider, $\exists$ sp $\in$ dp.securityDirectory, owner: Person

pre1 = (dp.mode = = true $\wedge$ dp.connection = = false) $\wedge$

~(dp.sp.owner.userid = uid $\wedge$ dp.sp.owner.password = pwd)

post1= (dp.mode = = true $\wedge$ dp.connection = = true) $\wedge$

(dp.sessions.size()' = dp.sessions.size()+1)

$\forall$ dp: DocProvider, $\exists$ sp $\in$ dp.securityDirectory, owner: Person

pre2 = (dp.mode = = true $\wedge$ dp.connection = = false) $\wedge$

(dp.sp.owner.userid = uid $\wedge$ dp.sp.owner.password = pwd)

post2= (dp.mode = = true $\wedge$ dp.connection = = false)

where dp.securityDirectory is a set of *SecurityProfiles* and dp.owner is a *Person* object. After that, we need to break the preconditions into DNF expressions, but, in this case, the preconditions are already in normal formal.

Test cases can be defined by analyzing the domain of the object variables involved. The instances of these objects are first built based on the object decision tree, and then, our extended domain matrix technique is used to identify and organize the test cases.

Table 9.1 from (a) to (d) shows the construction of the instances for objects *Person*, *AccessRight*, *SecurityProfile* and *DocProvider*, respectively. Table 9.2 shows the generated test cases for the pre-post condition pair 1 of the method *login*, and Table 9.3 shows the generated test cases for the pre-post condition pair 2 of the method *login*. We obtain in total eight potential test cases for both pre-post condition pairs. But only three of the eight test cases, which make the postcondition true (indicated by a "TRUE" in the *Expect Results* row), correspond to effective test cases. The remaining potential test cases falsify the preconditions, so we can't conclude anything after executing them.

Table 9.1(a) Instances For Class Person

No.	Object Var.	Instance Variable					
		name	userid	password	address	ssn	age
1	p1	Alex	alex	camry	40 Bay St.	1234567	20
2	p2	Alex	alex	camry	40 Bay St.	1234567	20
3	p3	Alex	alex	camry	40 Bay St.	1234567	20
4	p4	Alex	alex	camry	40 Bay St.	1234567	20
5	p5	Alex	alex	camry	40 Bay St.	1234567	20

Table 9.1 (b) Instances For Class AccessRight

No.	Object Var.	Instance Variable					
		read	modify	create	delete	addFriend	addDoctor
1	ac1	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
2	ac2	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
3	ac3	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
4	ac4	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
5	ac5	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE

Table 9.1 (c) Instances For Class SecurityProfile

No.	Object Var.	Instance Variable	
		owner	right
1	sp1	p1	ac1
2	sp2	p2	ac2
3	sp3	p3	ac3
4	sp4	p4	ac4
5	sp5	p5	ac5

Table 9.1 (d) Instances For Class DocProvider

No.	Object Var.	Instance Variable				
		mode	connection	service	securityStatus	securityDirectory
1	dp1	TRUE	FALSE	False	False	sp1
2	dp2	TRUE	FALSE	False	False	sp2
3	dp3	TRUE	FALSE	False	False	sp3
4	dp4	TRUE	FALSE	False	False	sp4
5	dp5	TRUE	FALSE	False	False	sp5

Table 9.1 Construction of the Instances for Object Variables

Domain Matrix For method login() in Class DocProvider (for pre/post pair 1)

Boundary			Test Case			
Instance Var.	condition	type	1	2	3	4
dp	dp.mode==true&& dp.connection==false&& dp.sp.p.userid=uid && dp.sp.p.password=pwd	on	dp1			
		off		dp2	dp3	dp4
	typical	in				
uid		on				
		off				
	typical	in	alex	alex	smith	smith
pwd		on				
		off				
	typical	in	camry	honda	camry	honda
Expected Results			TRUE	FALSE	FALSE	FALSE

Table 9.2 Test Cases for the Pre and Post Condition Pair 1 of Method Login

Domain Matrix For method login() in Class DocProvider (for pre/post pair 2)

Boundary			Test Case			
Instance Var.	condition	type	1	2	3	4
dp	dp.mode==true&& dp.connection==false&& dp.sp.p.userid!=uid	on			dp3	dp4
		off	dp1	dp2		
	typical	in				
uid		on				
		off				
	typical	in	alex	alex	smith	smith
pwd		on				
		off				
	typical	in	camry	honda	camry	honda
Expected Results			FALSE	FALSE	TRUE	TRUE

Table 9.3 Test Cases for the Pre and Post Condition Pair 2 of Method Login

9.3.2 Test Case Execution

Test execution starts at the class level by testing the individual methods involved in the class. Individual methods are tested by creating an instance of the class and setting the test values (i.e. an initial state) using the reflection API. After calling the method on the modified instance, we get the new state of the object still using the reflection API, and then finally evaluate the post conditions. The general approach to do so consists of writing test drivers or scripts. However, in our approach as mentioned earlier, the Java reflection mechanism is applied to directly modify and access object internal states. Also, this has been implemented in the PrUDE toolkit for executing a Java program automatically. Figure 9.4 depicts the results generated for the execution of the test cases shown in Table 9.2 for method *login*.

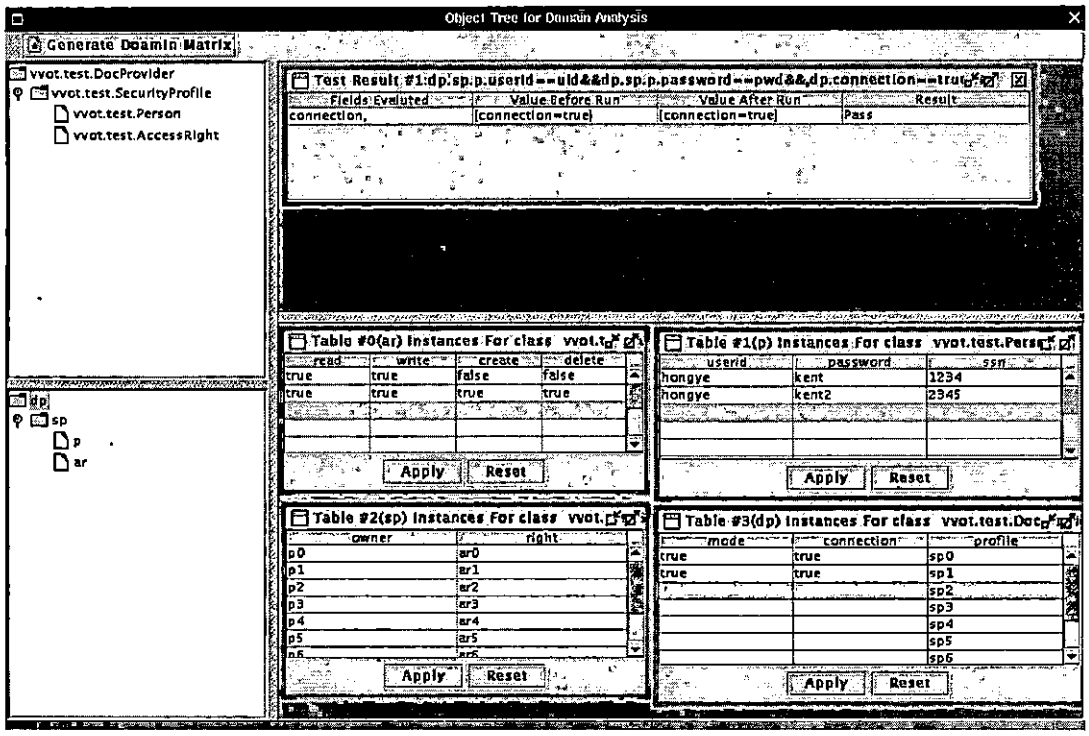


Figure 9.4 Test Execution Using the PrUDE Toolkit

9.4 Test Results and Analysis

To evaluate the effectiveness of the transition test strategy, an empirical study has been undertaken to demonstrate the feasibility of the three state-based test coverage criteria defined previously. The goal is to show that these coverage criteria are effectively used, and help us achieve good code execution coverage and completeness of test case generation.

The implementation of PDS used herein is a moderate size program with approximately eight hundred lines of Java code including ten classes and fifty-nine methods. Twenty faults are created manually, each of which may be inserted into a separate version of the program. Most of these faults are functional errors that are inconsistent with specifications, and they are well distributed in the different methods corresponding to the transitions defined in the state diagram. The measurement for the

test cases generated focus mainly on two aspects: the ability to detect faults and the code execution coverage.

The first measurement applies to an individual method. The method *login* of class *DocProvider* is selected for the evaluation. Six test cases are generated for the pre-post condition coverage criterion and the DNF coverage criterion. As a result, two faults in the method login are detected, and the six test cases cover all the code within the method *login*. Table 9.4 shows the result of detected errors and coverage for the method *login*.

Class	vvot.test.DocProvider
Method	login
Number of Errors Detected	2
Total Errors	2
Number of Lines of Code Executed	28
Total Lines of Code	28
Code Execution Coverage	100%

Table 9.4 Error Detection Result and Code Execution Coverage for Method Login

The other measurement is for the methods of a class. In this case, the class *DocProvider* is chosen as an example. There are thirty-four test cases that are generated for the transition coverage criterion. By executing all thirty-four test cases, sixteen functional faults are detected, and four faults are not revealed. The percentage of the detected faults and code execution coverage for the class *DocProvider* are shown in Figure 9.5. The correlation between the overall code execution coverage and the number of test cases generated for the class *DocProvider* is also shown in Figure 9.6.

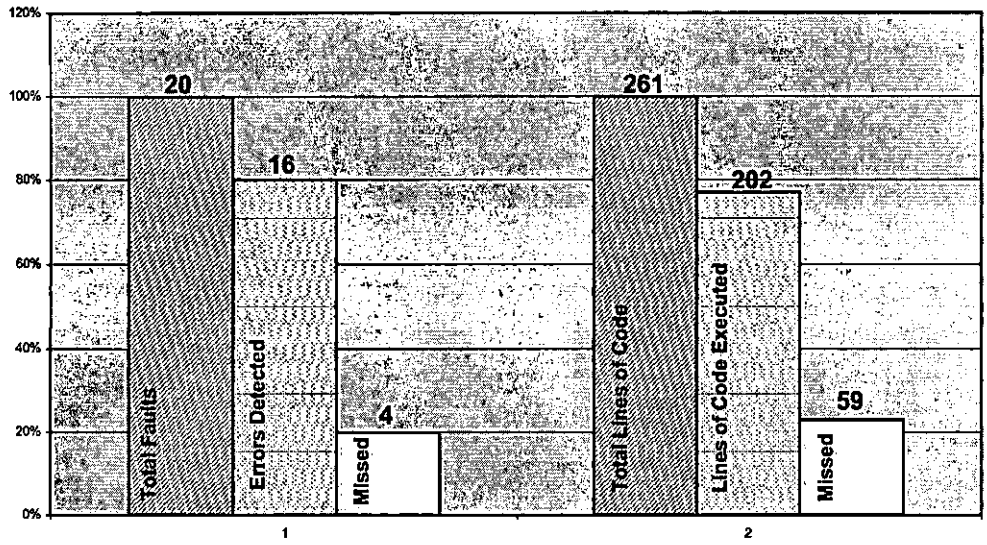


Figure 9.5 Error Detection Result and Code Execution Coverage for Class DocProvider

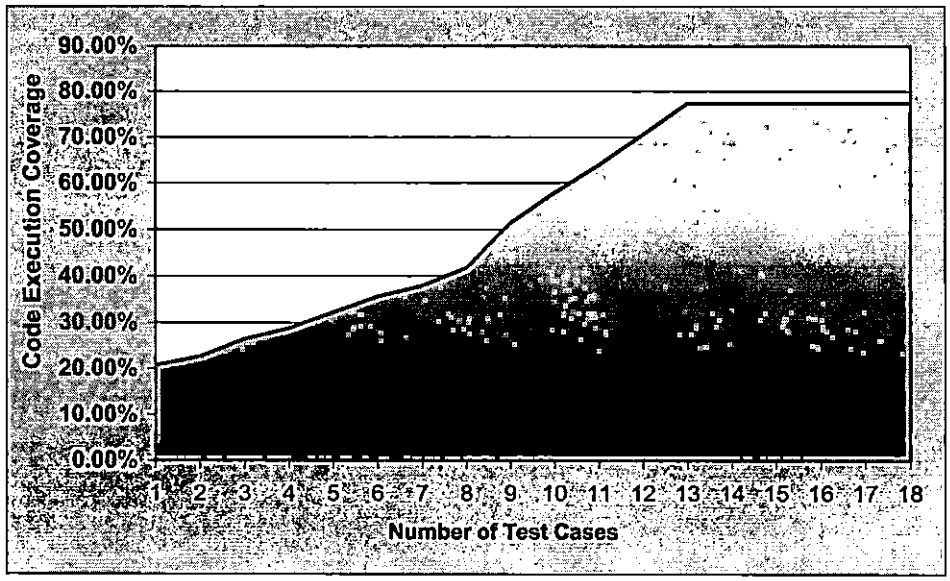


Figure 9.6 Test Case Quantity Report for Class DocProvider

Through detailed analysis of the faults, we learned that the four undetected faults are all naturally occurring faults made in the mutator or accessor methods, which could never been found by using this test methodology. The reason is that mutator or accessor

methods are usually not defined in a statechart diagram. However, complementary test cases for testing these methods are easily developed based on white-box techniques, i.e. code. For example, test data development for a mutator method that actually change one of the object's states into another state could still use the transition strategy, but the main difference is that the definition of state predicates will be code-based, rather than specification-based. Most mutator and accessor methods (e.g. set or get methods) are also not covered by executing our test cases in that the Java reflection mechanism is used to set and access the values of state variables in the object. This seems not to have significant effect on detecting the functional errors inserted although code execution coverage is decreased. In addition, the test case quantity report shown in Figure 9.6 presents a good snapshot of the fact that the total test cases generated are enough to achieve the general goal of code coverage.

9.5 Summary

This case study shows that the transition test strategy is particularly useful to deal with most functional faults in object-oriented programs. Meanwhile, the coverage criteria defined works well with the transition test. The case study demonstrates that the transition test strategy and the coverage criteria defined are effective in finding functional faults, which is just the focus of this research.

Chapter 10

Related Work

We review briefly, in this section, some of the work related to the research presented in this thesis.

10.1 Testing Based on Formal Specifications

A lot of research has been done using specifications for testing of object-oriented programs. Most approaches suggested are based on formal notations, such as Object-Z (an extension of Z), Algebraic Specification, and Contract Specification [38, 39, 40].

Chang and colleagues suggest in [38] that test data can be generated from formal specifications expressed in Object-Z [37]. They model the dynamic behaviour of a system by constructing so-called *enhanced state transition diagrams* (ESTDs). An enhanced state transition diagram includes information about inputs and outputs, parameters, functions and pre-post conditions. A probability associated with each transition is provided to indicate its usage frequency. The pre- and post conditions are obtained from an Object-Z specification, and can be interpreted by testers. Test cases can be eventually generated based on the information provided. The strength of their framework is the testing of more probable and important scenarios (e.g. a sequence of transitions) in a system determined using the probability weight. Unfortunately, their approach is currently program dependent and only targeted at testing programs written in C++.

Doong and Frankl propose the ASTOOT approach [39] to test object-oriented programs by using algebraic specifications. They recommend heuristic guidelines on the use of equivalent ground terms as class level test cases. However, relatively little work has been conducted for integration testing. Chen improves the ASTOOT approach of Doong and Frankl, and presents a unified methodology called TACCLE for object-oriented software testing at the class and cluster levels [40]. Although algebraic specifications for classes were still used in TACCLE, nonequivalent ground terms have been used to generate test cases at class levels using state-transition diagrams that troubled Doong and Frankl in using ASTOOT. Chen also introduces Contract specifications [41] used for integration testing at the level of a cluster of classes. Two automatic tools, GCS and ESI, have recently been developed to support cluster-level testing.

In short, formal specifications can be used to support a comprehensive testing of object-oriented programs. The common advantage is that test data is generated based on precise specifications. Thus, inconsistency between implementations and requirements is reduced. However, the complexity of the models along with strong mathematical demands on testers may present a serious barrier for wide scale adoption.

10.2 State-Based Testing

State-based testing techniques rely on the construction of a finite-state machine (FSM) or state-transition diagram that represents the change of states of the program under test. Object-oriented state testing is an important aspect of object-oriented testing, and is widely used for testing object-oriented programs.

Kung [42] presents an approach in which state machines are constructed from source code by combining reverse engineering and symbolic execution methods. The reverse engineering method depends on symbolic execution to retrieve the information of states for data members and the effects of the member functions against the states of the data members. A transition can then be constructed between the states of a data member based on the information acquired. Eventually, the transitions are used to construct the test model, which consists of a state-transition diagram called *object state diagram* (OSD) that represents state dependent behaviours of objects. The strength of the approach is that

the test model is a hierarchical state machine that resembles the concepts of aggregation and inheritance in object-oriented development. Hence, test cases can be generated for the state behaviour of a class and interclass interaction. However, the difficulty and costs of the approach may be much higher than those of exploring such errors at unit level testing.

Both Marlon and Jean's approaches to specification-based testing use UML statechart diagrams as the underlying specification [43, 44], similar to our approach. In Marlon's approach for testing Java programs, transitions, states and specific state of the object in a statechart are directly mapped to Java class operations, Java class attributes and range of attribute values, respectively in programs, which can be regarded as test-ready information. Then, test drivers and test scripts incorporating test cases and test oracles are constructed based on the information. This approach has also been implemented in a prototype tool called DAS-BOOT (**D**esign **A**nd **S**pecification **B**ased **O**bject-**O**riented Testing). Jean showed in their approach that several user-defined UML statecharts from a commercial tool, such as Rational Rose, could be used to construct a global behaviour model. In this global model, the significant properties, i.e. the behaviours of individual state machines are preserved. Test cases are then generated based on the finite global state machine using the *Category-Partition Method.* A UML specification is a semiformal specification that is popular in modeling object-oriented systems. UML statecharts are state transition diagrams that can be used to directly generate test cases. Hence, the approaches based on UML statecharts greatly reduce the cost to construct a finite state machine. However, both approaches indicated herein are used for integration testing of component interactions; the construction of a global state machine may be subject to the state explosion problem and become unmanageable. Whether or not the dynamic behaviour of individual classes in a software component is correct cannot be guaranteed either.

10.3 Class-Based Testing

One of the most important problems that arises when testing classes is the complex dependencies that may exist between the classes due to inheritance, association and

aggregation relationships. Hence, a number of papers have provided strategies and algorithms to derive test orders from dependencies [45] [46] [47] [48] [49].

Tai and Daniels propose a strategy that focuses mainly on the problem of cycle breaking [45]. In their approach, every class in a class model called an *Object Relation Diagram (ORD)* [48, 49] that is produced from code by means of reverse engineering is assigned a level number consisting of a major level number and a minor level number. A major level number is based on inheritance and aggregation relations only. For classes with the same major level number, each class is assigned a minor level number according to associations between these classes. For example, a class is said to have level numbered i-j, if its major number and minor level number are i and j, respectively. If there are no cycles in ORD, the test order can be obtained from ORD by a topological sort[1]. If there are cycles in the ORD, their algorithm suggests the deletion of more edges to break cycles. However, the computation of major level numbers does not take into account all association relationships. Thus, where the cycles are taking place is not considered. Also, their approach relies on the assumption that a stub developed for one class can be reused in any association relationship involving that class. This assumption is debatable.

Labiche and colleagues present a very different strategy to deal with dependencies by exploiting a class model produced during design phases, for example, using class diagrams in UML [46]. In their approach, two sets of classes and a boolean function are associated with each class *x*. The set of classes on which this class depends statically is denoted D1. D2 denotes the set of classes on which this class depends either statically or dynamically or both. The boolean function indicates whether or not this class may be dynamically dependent on at least one class of D2 due to polymorphism. Then, the test levels can be deduced from these two sets and the boolean functions. The whole approach is implemented in a prototype tool called TOONS (**T**esting level generator for **O**bject-**O**rie**N**te**d** **S**oftware), which can automatically perform dependency analysis, define test levels and order them. Their approach considers not only dynamic dependency (polymorphism), but also abstract classes that cannot be instantiated. The final result of the test order is represented by a *test order graph* for visualization. However, their

[1] A topological sort is an ordering of a directed graph in which all predecessors of every node are listed before the node itself.

approach does not take into account cycles, and assumes that cycles in class diagrams have already been broken.

10.4 Domain Testing

White and Cohen first proposed the domain testing technique [07], which has been modified by Clarke, Hassell and Richardson [35]. This technique can be mainly used to catch path selection errors or missing path errors in a program [36]. White and Cohen have shown their strategy to be effective for detecting domain errors and straightforward for selecting test data. However, their approach is limited to programs containing only linear predicate expressions and programs with variables defined over continuous domains. In addition, complexity and costs are very high requiring a relatively large number of test cases. For example, consider a program containing 10 variables. The White and Cohen's N × N strategy requires 10 On points, plus one Off point for each inequality predicate, and 10 On points, plus two or three OFF points for each equality predicate. If there are 10 decision statements in a path, their strategy requires between 110 to 130 test cases in order to test that path.

Jeng and Weyuker improved the White and Cohen strategy by exploring each of the limitations mentioned above and proposing a new strategy called the *Simplified Domain-Testing Strategy* [13]. They introduced a new approach to detecting domain errors indirectly by using a border shift, which was equivalent to directly detecting domain errors. By sampling a potential displaced area that is the expression of a pair of On-Off points, domain errors can frequently be detected. In their approach, it doesn't matter whether or not borders are linear. Therefore, the restrictions on programs containing only linear predicates and variables defined over continuous domains are removed. Another strength of their approach is that they define only two points, one On and one Off point, for each inequality border to expose a border shift. The On point can be located anywhere on the given border. For an equality border, the approach requires one On point and two Off points to assure that the correct border is not an inequality border. Hence, the number of test cases selected is significantly reduced.

The domain testing strategies mentioned above are particularly useful for test case selection. However, they are only applicable for variables of primitive types in programs, and do not take into account complex variables (e.g. classes).

10.5 Summary

There is a lot of research on testing of object-oriented programs. Since object-oriented programs involve many unique features that do not appear in traditional programs such as encapsulation, object instantiation, inheritance, polymorphism and overridden methods, a variety of test techniques to respond to these unique features have been developed. Examples of these methodologies are specification-based testing, code-based testing, formal or semiformal techniques, state-based and class-based testing. Each approach addresses specific issues, and therefore reveals specific kind of faults. However, no single model is able to cover all kinds of errors in object-oriented programs.

Chapter 11

Conclusion

Software testing is widely used as a means for engineers to develop high quality systems. A variety of test strategies and models for the testing of object-oriented programs has been developed due to the popularity of object-oriented technology in recent years. Some are specification-based, and others are code-based; both have advantages and disadvantages. The main goal of our research was to investigate how efficient test strategies for object-oriented systems can be defined by using UML specifications.

This thesis proposes transition and relational test strategies covering different test levels, which deal with problems introduced by new features of object-oriented languages such as encapsulation, inheritance, and polymorphism. The transition test strategy is defined based on a statechart diagram of a class, and is mainly used to test methods of a class at the unit level. Inputs and outputs in this approach are not independently treated as traditional unit testing, and they must be associated with states of classes. The relational test strategies based on class diagrams focus mainly on testing relationships between classes. They are mostly used for integration testing of a cluster of classes. More specifically, the properties of each association in a class diagram, such as multiplicity and referential integrity are tested. Strategies for testing class inheritance investigate mainly subtype substitution rules and data flow anomalies between a super and a sub class due to the use of polymorphism and overridden methods.

Domain testing in combination with other test strategies for test case selection is optimized and extended. Domain testing now can be used for object variables in object-

oriented programs. In addition, adequate criteria are defined for test coverage based on specification or requirements. These adequate criteria can be used to measure the effectiveness of the test strategies and the thoroughness of code execution.

The PrUDE tool suite that supports the platform is designed in order to allow the integration of new test strategies. The tool provides heuristics to users, and automates test case generation and execution.

In short, the main contributions from this thesis can be summarized as follows:

- transition test strategy for testing errors of operational behavior in classes
- relational testing strategy for testing errors of class association
- strategies to test class inheritance for testing mainly errors of conformance rules and data flow anomalies brought by the use of overridden methods between super and sub classes
- adequate criteria for test coverage analysis which can help to increase confidence that code is executed
- implementation of the transition test strategy

In this thesis, the test strategies for test case generation are mainly presented based on two sub-models of UML, i.e. state and class diagrams. In future work, we plan to investigate other UML diagrams as well. Firstly, two other diagrams are worthy of note: use case diagrams identify the functionality provided by the system (use cases), and describe behavior of the system. Use case diagrams may be helpful for detecting test scenarios at the system testing level. Sequence diagrams describe interactions between different objects. Hence, a trace-based testing strategy may be adequate for integration testing. Secondly, component and deployment diagrams provide information about the overall structure of the system and the interconnections among modules. They may be used to investigate possible test case generation for distributed testing.

Bibliography

[01] Boris Beizer. "Software Testing Techniques" (Second Edition), Van Nostrand Reinhold, New York, 1990, ISBN 0-442-20672-0

[02] John D. McGregor, David A. Sykes. "*A Practical Guide to Testing Object-Oriented Software*" (object technology series), Addison-Wesley, 2001, ISBN0-201-32564-0

[03] Robert V. Binder *" Testing the object-oriented system: Models, Patterns and Tools*", Addison-Wesley Object Technology

[04] The Liskov Substitution Principle, http://www.objectmentor.com/resources/articles/lsp.pdf

[05] Glenford J. Myers. "The Art of Software Testing", WILEY-INTERSINCE 1979 ISBN 0-471-04328-1

[06] Michael Liu, Hong Ye and Issa Traore, *"Using Formal Methods In Security Engineering: Case Study of a Patient Document Service"*, Technical Report No. ECE01-3, May 2001, University of Victoria

[07] White, L J. and Cohen, E.I. 1980. "*A Domain Strategy for Computer Program Testing*" IEEE Trans. Software Engineering. SE-6, 5(May), 247-257

[08] Huang, J. C. "Detection of data flow anomaly through program instrumentation." *IEEE Transactions on Software Engineering* 5:226-236(1979).

[09] P. Frankl and E. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. SE-14, no. 10, 1988, pp. 1483-1498.

[10] S, Rapps and E.J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 4, 1985, pp. 367-375.

[11] Stuart J. Russell and Peter Norvig, "Artificial Intelligence: A Modern

[24] I. Traore, K. Stolen, "Formal Development of Open Distributed Systems: Towards An Integrated Framework" Research Report No.273, University of Oslo, Norway, August 1999. Submitted to OOSDS'99, Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviors, Paris, France, Sep. 27, 1999

[25] R. B. France, A. Evans, B. Rumpe, *The UML As A Formal Modeling Notation*, Computer Standards & Interfaces, 19(1998), P.325-334

[26] A. Evans, *UML Class Diagrams –Filling The Semantic Gap (Draft)*, Technical Report, York University, 1998

[27] Latella, I. Majzik, M. Massink, "Towards A Formal Operational Semantics of UML Statechart Diagrams" Proc. FMOOD'99, Feb. 1999, Florence, Italy.

[28] Ricky W. Butler, *An Elementary Tutorial on Formal Specification and Verification Using PVS 2*, September 1993

[29] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, Mandayam Srivas, *A Tutorial Introduction to PVS*, presented at WIFT'95: Workshop on Industrial -Strength Formal Specification Techniques, Boca Raton, Florida, April 1995

[30] Issa Traore. "An Outline of PVS Semantics for UML Statecharts" Journal of Universal Computer Science, 6(11) 2000

[31] D.B. Aredo, I. Traore and K. Stolen. "An Outline of PVS Semantics of UML Class Diagrams". *In the Proc. Of The 11th Nordic Workshop on Programming Theory NWPT'99*, Uppsala, Sweden, October 6-8, 1999

[32] D.B. Ardeo. "Semantics of UML Sequence Diagrams in PVS (extended abstract)". *In the Proc. Of the Workshop on dynamic Behavior in UML Models, at UML2000*, York UK, October 2-6 2000

[33] Grady Booch, James Rumbaugh and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, Inc. 1999, ISBN: 0201571684

[34] Steve Cornett, Bullseye Testing Technology. http://www.bullseye.com/coverage.html

[35] Clarke, L. A., Hassell, J., AND Richardson, D. J. A Close Look at Domain Testing. IEEE Trans. Softw. Eng. SE-8, 4 (July 1982), 380-390

[36] Chou, C.-S., AND Du, M.-W. Improved domain strategies for detecting path selection errors. In Proceedings of the Conference on Software Maintenance (Sept.

1987). IEEE Computer Society, Los Angeles, 1987, pp. 165-173.

[37] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Boston, 2000.

[38] Kai H. Chang, Shih-Sung Liao, Richard Chapman and Chun-Yu Chen. Test Scenario Generation Based On Formal Specification and Usage Profile. *International Journal of Software Engineering and Knowledge Engineering*. Vol. 10, No. 2 (2000) 00-00

[39] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[40] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: A Methodology For Object-Oriented Software Testing At the Class and Cluster Levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[41] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[42] D.C. Kung, N. Suchak, J. Dao and Ping. Hsia. "On Object State Testing." *In Proc. Of IEEE COMPSAC'94 Conference*. Feb. 26, 1994

[43] Marlon E. Vieira, Marcio S. Dias, and Debra J. Richardson. "Object-Oriented Specification Based Testing Using UML Statechart Diagrams" ICSE2000 Workshop on Automated Program Analysis, Testing, and Verification. June, 2000, Limerick, Ireland

[44] Jean Hartmann, Claudio Imoberdorf and Michael Meisinger. "UML-Based Integration Testing", *Proc. Of International Symposium on Software Testing and Analysis*, August 21-24, 2000, (ISSTA2000) Portland, Oregon, USA

[45] K. C. Tai and F. J. Daniels. "Interclass Test Order for Object-Oriented Software" *Journal of Object-Oriented Programming*, vol.23 (2), pp. 95-109, 1993

[46] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.-H. Durand, "Testing Levels for Object-Oriented Software," *Proc. 22 nd IEEE International Conference on Software Engineering (ICSE'2000)*, Limerick (Ireland), pp. 136-145, June, 2000.

[47] L. Briand, Y. Labiche and Y. Wang, "Revisiting Strategies for Ordering Class

Integration Testing in the Presence of Dependency Cycles," *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE'2001)*, Hong Kong, pp. 287-296, November 27-30, 2001.

[48] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Class Firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8 (2), pp. 51-65, 1995.

[49] D. Kung, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, "On Regression Testing of Object-Oriented Programs," *Journal of Systems Software*, vol. 32 (1), pp. 21-40, 1996.

[50] J. Dick, A. Faivre, *Automating the Generation and Sequencing of Test Cases From Model-Based Specifications*, Proc. FME'93, Odense, Denmark, April 1993, LNCS'670

[51] F. C. Hennie, "Finite-State Models for Logical Machines", John Wiley & Sons, 1968.

[52] C. D. Turner, and D. J. Robson, "State-Based Testing and Inheritance", Technical Report TR1/93, University of Durham, England 1993

[53] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, "A Test Strategy for Object-Oriented Systems", Proceedings, The Nineteen Annual International Computer Software and Applications Conference. August 1995, IEEE Computer Society Press, Los Alamitos, Calif. 239-244.

[54] B. Tsai, S. Stobart, N. Parrington, "A Method for Automatic Class Testing (MACT) Object-Oriented Programs Using A State-based Testing Method", EuroSTAR '97, November 1997.

[55] Issa Traore, "Making the UML More Precise: A Formal Framework for UML Statechart", TchnicalReport No. ECE 00-4, Dept. of Electrical and Computer Engineering, U. of Victoria, October 2000

[56] William Perry. "Effective Methods for Software Testing" (2nd Edition), John Wiley & Sons, Inc 2000, ISBN 0-471-35418-X

[60] Dr. Linda H. Rosenberg, Larry Hyaat *"Applying and Interpreting Object-Oriented Metrics"*, Internet resources, http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply.pdf

[61] El-emam, K. "Object-Oriented Metrics: A Review of Theory and Practice" National Research Council Canada, ERB-1085, March 2001

[62] Robert M. Poston, "Automating Specification-Based Software Testing" IEEE Computer Society Press, 1996, ISBN: 0-8186-7531-4

[63] J. B. Warmer and A. G. The Object Constraint Language: Precise Modeling with UML" Addision Wesley Longman Inc, 1999

[64] The OMG, "Object Constraint Language (OCL)", OMG standard document http://www.omg.org/docs/ad/01-08-28.doc

Appendix A

The Java Program for the Banking Application

Account.java

```
package bank;

import java.lang.*;
import java.util.*;

public class Account
{
        private int accNo;
        private double balance;
        private double creditLine;
        private int id;
        private Vector owners  = new Vector( );//list of account holders (customers)

        public Account(int accNo, double balance,  Customer c)
        {
                this.accNo = accNo;
                this.balance = balance;
                creditLine=getAverageIncome();
                if(c != null)
                {
                        owners.add(c);
                }
                else
                {
                        //return message
                        System.out.println( "A valid customer is needed here");
                }
        }

        public void deposit(double amount)
        {
                if(balance>=0)
                {
                        balance += amount;
                }
                else
```

```
        {
                //calculate fees
                double  fees =  (double)balance*0.01;

                //update fees
                balance = balance + amount – fees;
        }

}

public void withdraw(double amount)
{
        if(balance>=0)
        {
                balance -= amount;
        }
        elese
        {
                //calculate fees
                double  fees =  (double)balance*0.01;

                //update fees
                balance = balance - amount – fees;
        }
}

public void transfer(double amount, Account dest)
{
        if(0 <= balance – amount + creditLine)
        {
                withdraw(amount);
                dest.deposit(amount);
        }
        elese
        {
                System.out.println("Transfer Denied: insufficient fund");
        }
}

public int getAccNo( )
{
        return accNo;
}

public void setAccNo(int i )
{
```

```
            accNo=i;
    }

    public int getBalance( )
    {
            return balance;
    }

    public void setBalance(double d )
    {
            balance=d;
    }

    public double getCreditLine( )
    {
            return creditLine;
    }

    public void setCreditLine(double d )
    {
            creditLine=d;
    }

     //attribute averageIncome is not represented directly in the program, so the
     // programmer needs to provide a method that allows its retrieval
    public double getAverageIncome()
    {
            Enumeration enum = owners.elements();
            int size = owners.size();
            double average = 0 ;

            while(enum.hasMoreElements())
            {
                    average=average+((Customer)enum.nextElement()).income/size;
            }

            return average;
    }

}//end of class Account
```

Customer.java

```
package bank;

import java.lang.*;
import java.util.*;

public class Customer
{
        private String name;
        private int ssn;
        private String address;
        private Vector accounts  = new Vector( );//list of opened accounts

        public Customer(String name, int ssn, String address)
        {
                this.name=name;
                this.ssn=ssn;
                this.address=address;

        }

        public String getName( )
        {
                return name;
        }

        public void setName(String s )
        {
                name = s;
        }

        public int getSsn( )
        {
                return ssn;
        }

        public void setSsn(int i )
        {
                ssn = i;
        }

        public String getAddress( )
        {
                return address;
        }
```

```
    public void setAddress(String s )
    {
        address = s;
    }

}//end of class Customer
```

Appendix B

The Java Program for Patient Document Service

DocProvider.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.lang.Class;
import java.io.*;
import java.lang.reflect.*;

public class DocProvider
{
        private boolean mode;
        private boolean connection;
        private boolean service;
        private boolean securityStatus;
        private Hashtable vprofile = new Hashtable();
        private Hashtable vsession = new Hashtable();
        private Hashtable vmedrecord = new Hashtable();

        /**initialize the security parameters*/
        public DocProvider(boolean b1,boolean b2, boolean b3, boolean b4)
        {
                mode=b1;
                connection=b2;
                service=b3;
                securityStatus=b4;

        }

        public void setMode(boolean b)
        {
                mode=b;
        }

        public boolean getMode()
        {
                return mode;
        }
```

```
public void setConnection(boolean b)
{
        connection=b;
}

public boolean getConnection()
{
        return connection;
}

public void setService(boolean b)
{
        service=b;
}

public boolean getService()
{
        return service;
}

public void setSecuirtyStatus(boolean b)
{
        securityStatus=b;
}

public boolean getSecurityStatus()
{
        return securityStatus;
}

public Hashtable getVprofile()
{
        return vprofile;
}

public Hashtable getVsession()
{
        return vsession;
}

public Hashtable getMedRecord()
{
        return vmedrecord;
}
```

```
/**create the profile in SecurityProfile for a new registered person */
public void register(Person user) throws  java.lang.ClassNotFoundException
{
        Class cpatient = Class.forName("Patient");
        Class cfriend = Class.forName("Friend");
        Class cdoctor = Class.forName("Doctor");
        Class cadmin = Class.forName("Administrator");

        if(cpatient.isInstance(user))
        {
                String tmpuid=user.getUserid();

                if(!vprofile.containsKey(tmpuid))
                {
                        AccessRight ar = new AccessRight
                        (true,false,false,false,true,true,true,true);
                        SecurityProfile sp = new SecurityProfile(user,ar);

                        vprofile.put(tmpuid,sp);

                        //the action of a successful registration
                        mode=true;
                }
        }

        if(cfriend.isInstance(user))
        {
                String tmpuid=user.getUserid();

                if(!vprofile.containsKey(tmpuid))
                {
                         AccessRight ar = new AccessRight
                        (true,false,false,false,false,false,false,false);
                         SecurityProfile sp = new SecurityProfile(user,ar);

                        vprofile.put(tmpuid,sp);

                        //the action of a successful registration
                        mode=true;
                }

        }

        if(cdoctor.isInstance(user))
        {
                String tmpuid=user.getUserid();
```

```
                if(!vprofile.containsKey(tmpuid))
                {
                        AccessRight ar = new AccessRight
                        (true,true,false,false,false,false,false,false);
                        SecurityProfile sp = new SecurityProfile(user,ar);

                        vprofile.put(tmpuid,sp);

                        //the action of a successful registration
                        mode=true;

                }
        }

        if(cadmin.isInstance(user))
        {
                String tmpuid=user.getUserid();

                if(!vprofile.containsKey(tmpuid))
                {
                        AccessRight ar = new AccessRight
                        (true,true,true,true,true,true,true,true);
                        SecurityProfile sp = new SecurityProfile(user,ar);

                        vprofile.put(tmpuid,sp);

                        //the action of a successful registration
                        mode=true;
                }
        }

}

/**check the username and password of the user*/
public void login(String uid,String pwd)
{

        if(mode==true&&connection==false)
        {

                String tempuid=
                (((SecurityProfile)vprofile.get(uid)).getPerson()).getUserid();
                String temppwd=
                 (((SecurityProfile)vprofile.get(uid)).getPerson()).getPassword();
```

```
                if(uid.equals(tempuid)&&pwd.equals(temppwd))
                {
                        //the action of the successful login
                        connection=true;

                        Person tp = ((SecurityProfile)vprofile.get(uid)).getPerson();

                        Session s = new Session(tp);
                        vsession.put(uid,s);

                }
                else
                {
                        //the action of the unsuccessful login
                        connection=false;
                }
        }

}

public void detectViolation()
{
        if(securityStatus==false)
        {

                if(connection==true)
                {
                        mode=false;
                        connection=false;
                }


                if(service==true)
                {
                        mode=false;
                        service=false;
                }

                if(connection==true&&service==true)
                {
                        mode=false;
                        connection=false;
                        service=false;

                }
        }
```

```
}

public void backToNormal()
{
        if(mode==false)
        {
                mode=true;
                connection=false;
                service=false;
        }
        else
        {
                new DocProvider(false,false,false,true);

        }
}

public void sendRequest(Request req)
{
        if(mode==true&&service==false)
        {
                String tmpuid =((Request)req.getOwner()).getUserid();
                if(vsession.containsKey(tmpuid)
                {
                        service = true;
                }
        }

}

public void recvResult()
{



}

public void close()
{

}


public void abNormalClose()
{
```

```
        }

}//end of class DocProvider
```

SecurityProfile.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class SecurityProfile
{
        private Person owner;
        private AccessRight right;

        public SecurityProfile(Person p, AccessRight ar)
        {
        owner=p;
                right=ar;

        }

        /**set the value of attribute owner*/
        public void setPerson(Person p)
        {
                owner=p;
        }

        public Person getPerson()
        {
                return owner;
        }

        /**set the value of attribute right*/
        public void setRight(AccessRight ac)
        {
                right=ac;
        }

        public AccessRight getRight()
        {
                return right;
        }
```

```
}//end of class SecurityProfile
```

AccessRight.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class AccessRight
{
        private boolean read;
        private boolean write;
        private boolean create;
        private boolean delete;

        private boolean addDoctor;
        private boolean removeDoctor;
        private boolean addFriend;
        private boolean removeFriend;

        public AccessRight(boolean r,boolean w,boolean c,boolean d,
                        boolean ad,boolean rd,boolean af,boolean rf)
        {
        read = r;
                write = w;
                create = c;
                delete = d;

                addDoctor = ad;
                removeDoctor= rd;
                addFriend = af;
                removeFriend = rf;

        }

        /**set the value of attribute read*/
        public void setRead(boolean b)
        {
                read=b;
        }

        public boolean getRead()
```

```
{
        return read;
}

/**set the value of attribute write*/
public void setWrite(boolean b)
{
        write=b;
}

public boolean getWrite()
{
        return write;
}

public void setCreate(boolean b)
{
        create=b;
}

public boolean getCreate()
{
        return create;
}

public void setDelete(boolean b)
{
        delete=b;
}

public boolean getDelete()
{
        return delete;
}

public void setAddDoctor(boolean b)
{
        addDoctor=b;
}

public boolean getAddDoctor()
{
        return addDoctor;
}

public void setRemoveDoctor(boolean b)
```

```
        {
                removeDoctor=b;
        }

        public boolean getRemoveDoctor()
        {
                return removeDoctor;
        }

        public void setAddFriend(boolean b)
        {
                addFriend=b;
        }

        public boolean getAddFriend()
        {
                return addFriend;
        }

        public void setRemoveFriend(boolean b)
        {
                removeFriend=b;
        }

        public boolean getRemoveFriend()
        {
                return removeFriend;
        }

}//end of this class AccessRight
```

Person.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Person
{
        protected String name;
        protected String userid;
        protected String password;
```

```
protected String address;
protected int ssn;

public Person(String u,String p, int s)
{

name=null;
        userid=u;
        password=p;

        address=null;
        ssn=s;
}

Person()
{

}

public void setName(String s)
{
        name=s;
}

public String getName()
{
        return name;
}

public void setUserid(String s)
{
        userid=s;
}

public String getUserid()
{
        return userid;
}

/**set the value of attribute pwd*/
public void setPassword(String s)
{
        password=s;
}

public String getPassword()
```

```
        {
                return password;
        }

        public void setAddress(String s)
        {
                address=s;
        }

        public String getAddress()
        {
                return address;
        }

        public void setSSN(int i)
        {
                ssn=i;
        }

        public int getSSN()
        {
                return ssn;
        }

}//end of class Person
```

Session.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Session
{
        Person owner;

        public Session(Person p)
        {
                owner=p;
        }

        public void sendRequest()
        {
```

```
        }

        public void logout()
        {

        }

}//end of class Session
```

Patient.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Patient extends Person
{
        private Hashtable vlist = new Hashtable();

        Patient(String u,String p, int s)
        {
                name=null;
                userid=u;
                password=p;

                address=null;
                ssn=s;

        }


        public void addDoctor(Doctor d)
        {
                vlist.put(d.getUserid(),d);

        }

        public void remDoctor(Doctor d)
        {
                vlist.remove(d.getUserid());

        }
```

```
	public void addFriend(Friend f)
	{
		vlist.put(f.getUserid(),f);

	}

	public void remFriend(Friend f)
	{

		vlist.remove(f.getUserid());
	}

}//end of class Patient
```

Friend.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Friend extends Person
{
	private String uid;

	Friend(String u,String p, int s)
	{
		name=null;
		userid=u;
		password=p;

		address=null;
		ssn=s;

	}
}//end of class Friend
```

Doctor.java

```
package vvot.test;

import java.util.*;
```

```
import java.lang.*;
import java.io.*;

public class Doctor extends Person
{
        Doctor(String u,String p, int s)
        {
                name=null;
                userid=u;
                password=p;

                address=null;
                ssn=s;

        }

}//end of class Doctor
```

Administrator.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Administrator extends Person
{
        private String uid;

        Administrator(String u,String p, int s)
        {
                name=null;
                userid=u;
                password=p;

                address=null;
                ssn=s;

        }

}//end of class Administrator
```

MedicalRecord.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class MedicalRecord
{
	private Person owner;
	private String medicalinfomation;

	public MedicalRecord(Person p,String info)
	{
		owner=p;
		medicalinfomation=info;
	}

	/**set the value of attribute owner*/
	public void setPerson(Person p)
	{
		owner=p;
	}

	public Person getPerson()
	{
		return owner;
	}

	/**set the value of attribute owner*/
	public void setMedicalInfomation(String str)
	{
		medicalinfomation=str;
	}

	public String getMedicalInfomation()
	{
		return medicalinfomation;
	}
}//end of class MedicalRecord
```

Request.java

```
package vvot.test;

import java.util.*;
import java.lang.*;
import java.io.*;

public class Request
{
        private Person owner;

        public MedicalRecord(Person p)
        {
                owner=p;
        }

        /**set the value of attribute owner*/
        public void setOwner(Person p)
        {
                owner=p;
        }

        public Person getOwner()
        {
                return owner;
        }

}//end of class Request
```

VITA

Surname: Ye Given Name: Hong

Place of Birth: Henan, China Date of Birth: November 01, 1968

Educational Institutions Attended:

University of Victoria, Victoria BC, Canada 2001 to 2003

Zhejiang University, Hangzhou, China 1986 to 1990

Degree Awarded:

B.Eng. Zhejiang University, Hangzhou, China 1990

Publications:

- Issa Traore, Demissie Ardeo and Hong Ye, *"An Integrated Framework for Formal Development of Open Distributed Systems"*, ACM Symposium on Applied Computing, SAC 2003, Melbourne, Florida USA
- Michael Liu, Hong Ye and Issa Traore, *"Using Formal Methods In Security Engineering: Case Study of a Patient Document Service"*, Technical Report No. ECE01-3, May 2001, University of Victoria

PARTIAL CORYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purpose may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

UML Based Testing of Object-Oriented Programs

Author 

Hong Ye

Aug. 13, 2003

Date